UNIT IV

PARALLEL AND DISTRIBUTED PROGRAMMING PARADIGMS

Parallel and distributed program is defined as a parallel program running on a set of computing engines or a distributed computing system. A distributed computing system is a set of computational engines connected by a network to achieve a common goal of running a job or an application. A computer cluster or network of workstations is an example of a distributed computing system. Parallel computing is the simultaneous use of more than one computational engine (not necessarily connected via a network) to run a job or an application. It has several advantages.

- From the users' perspective, it decreases application response time;
- from the distributed computing systems' standpoint, it increases throughput and resource utilization

the system should include the following

Partitioning

This is applicable to both computation and data as follows:

Computation partitioning This splits a given job or a program into smaller tasks. Partitioning greatly depends on correctly identifying portions of the job or program that can be performed concurrently.

Data partitioning This splits the input or intermediate data into smaller pieces. Similarly, upon identification of parallelism in the input data, it can also be divided into pieces to be processed on different workers.

Mapping This assigns the either smaller parts of a program or the smaller pieces of data to underlying resources. This process aims to appropriately assign such parts or pieces to be run simultaneously on different workers and is usually handled by resource allocators in the system.

Synchronization Because different workers may perform different tasks, synchronization and coordination among workers is necessary so that race conditions are prevented and data dependency among different workers is properly managed.

Communication Because data dependency is one of the main reasons for communication among workers, communication is always triggered when the intermediate data is sent to workers.

Scheduling For a job or program, when the number of computation parts (tasks) or data pieces is more than the number of available workers, a scheduler selects a sequence of tasks or data pieces to be assigned to the workers. It is worth noting that the resource allocator performs the actual mapping of the computation or data pieces to workers, while the scheduler only picks the next part from the queue of unassigned tasks based on a set of rules called the scheduling policy.

Motivation for Programming Paradigms

Simplicity of writing parallel programs is an important metric for parallel and distributed programming paradigms. Others are

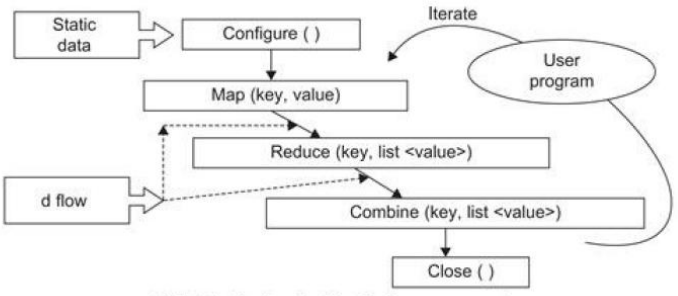
- to improve productivity of programmers
- to decrease programs' time to market
- to leverage underlying resources more efficiently
- to increase system throughput,
- to support higher levels of abstraction

The loose coupling of components in these paradigms makes them suitable for VM implementation and leads to much be□er fault tolerance and scalability.

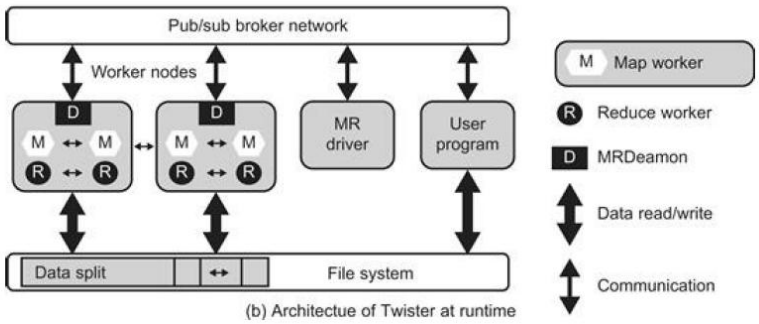
Twister

Twister is a lightweight MapReduce runtime we have developed by incorporating these enhancements

- Distinction on static and variable data
- Configurable long running (cacheable) map/reduce tasks
- Pub/sub messaging based communication/data transfers
- Efficient support for Iterative MapReduce computations (extremely faster thanHadoop or Dryad/DryadLINQ)
- Combine phase to collect all reduce outputs
- Data access via local disks
- Lightweight (~5600 lines of Java code)
- Support for typical MapReduce computations
- Tools to manage data
- Twister is much faster than traditional MapReduce.



(a) Twister for iterative MapReduce programming



(b) Architectue of Twister at runtime

Hadoop Library from Apache

Hadoop is an open source implementation of MapReduce coded in Java by Apache. This implementation uses the *Hadoop Distributed File System (HDFS)* as its underlying layer. It has two fundamental layers: the MapReduce engine and HDFS. The MapReduce engine is the computation engine running on top of HDFS as its data storage manager.

HDFS: It is a distributed file system that organizes files and stores their data on a distributed computing system.

HDFS Architecture

HDFS has a master/slave architecture containing a single NameNode as the master and a number of DataNodes as workers (slaves). To store a file in this architecture, HDFS splits the file into fixed-size blocks and stores them on workers. The mapping of blocks to DataNodes is determined by the NameNode. The NameNode (master) also manages the file system's metadata and namespace. Each DataNode, usually one per node in a cluster, manages the storage attached to the node. Each DataNode is responsible for storing and retrieving its file. Each DataNode is responsible for storing and retrieving its file blocks.

HDFS Features

Distributed file systems have special requirements, such as performance, scalability, concurrency control, fault tolerance, and security requirements, to operate efficiently. However, it does not need all the requirements for HDFS, beacuse it only executes specific types of applications. Two important characteristics of HDFS to distinguish it from other generic distributed file systems.

HDFS Fault Tolerance

Since Hadoop is designed to be deployed on low-cost hardware by default, a hardware failure in this system is considered to be common rather than an exception. Hadoop considers the following issues to fulfill reliability requirements of the file system:

Block replication

HDFS stores a file as a set of blocks and each block is replicated and distributed across the whole cluster. The replication factor is set by the user and is three by default.

Replica placement

The placement of replicas is another factor to fulfill the desired fault tolerance in HDFS. Although storing replicas on different nodes (DataNodes) located in different racks across the whole cluster provides more reliability, it is sometimes ignored as the cost of communication between two nodes in different racks is relatively high in comparison with that of different nodes located in the same rack. Therefore, sometimes HDFS compromises its reliability to achieve lower communication costs.

Heartbeat and Blockreport messages

Heartbeats and Blockreports are periodic messages sent to the NameNode by each DataNode in a cluster. Receipt of a Heartbeat implies that the DataNode is functioning properly, while each Blockreport contains a list of all blocks on a DataNode.

HDFS Operation

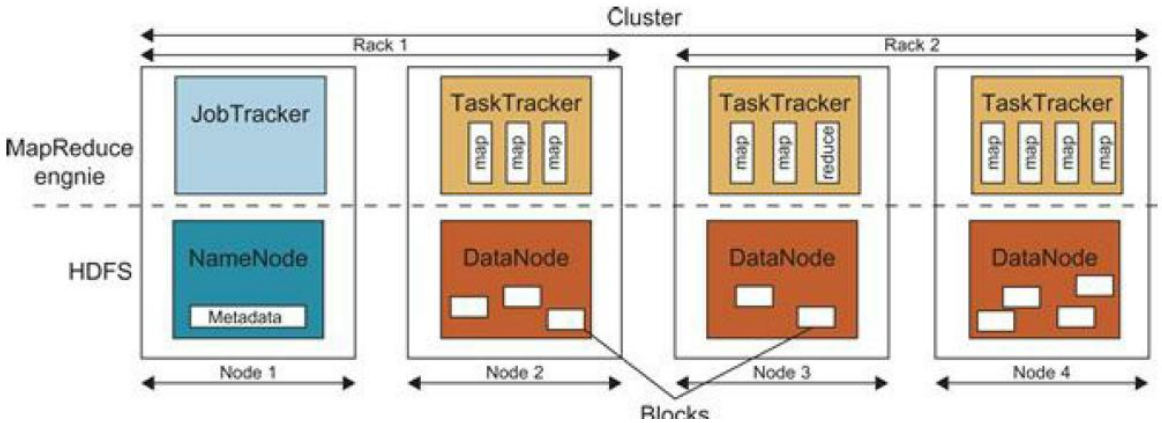
The control flow of the main operations of HDFS on files is the interaction between the user, the NameNode, and the DataNodes.

Reading a file To read a file in HDFS, a user sends an "open" request to the NameNode to get the location of file blocks. For each file block, the NameNode returns the address of a set of DataNodes containing replica information for the requested file. Upon receiving such information, the user calls the read function to connect to the closest DataNode containing the first block of the file. After the first block is streamed, the established connection is terminated and the same process is repeated for all blocks of the requested file.

Writing to a file To write a file in HDFS, a user sends a "create" request to the NameNode to create a new file in the file system namespace. If the file does not exist, the NameNode notifies the user and allows him to start writing data to the file by calling the write function. The first block of the file is written to an internal queue termed the data queue while a data streamer monitors its writing into a DataNode. Since each file block needs to be replicated by a predefined factor, the data streamer first sends a request to the NameNode to get a list of suitable DataNodes to store replicas of the first block.The procedure will repeat for all the blocks of a file.

Architecture of MapReduce in Hadoop

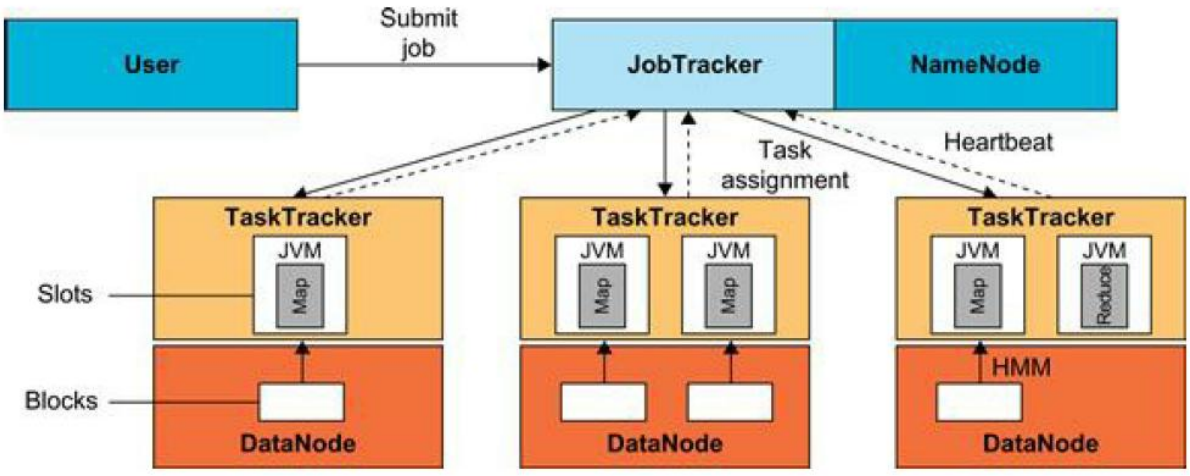
Figure shows the MapReduce engine architecture cooperating with HDFS. Similar to HDFS, the MapReduce engine also has a master/slave architecture consisting of a single JobTracker as the master and a number of TaskTrackers as the slaves (workers). The JobTracker manages the MapReduce job over a cluster and is responsible for monitoring jobs and assigning tasks to TaskTrackers. The TaskTracker manages the execution of the map and/or reduce tasks on a single computation node in the cluster.



Each TaskTracker node has a number of simultaneous execution slots, each executing either a map/reduce task. Slots are defined as the number of simultaneous threads supported by CPUs of the TaskTracker node. For example, a TaskTracker node with N CPUs, each supporting M threads, has M * N simultaneous execution slots.

Running a Job in Hadoop

Three components are contributing to run a job in this system: a user node, a JobTracker, and several TaskTrackers. The data flow starts by calling the runJob(conf) function inside a user program running on the user node, in which conf is an object containing some tuning parameters for the MapReduce framework and HDFS. The runJob(conf) function and conf are comparable to the MapReduce(Spec, &Results) function and Spec in the first implementation of MapReduce by Google. Figure depicts the data flow of running a MapReduce job in Hadoop.



Job Submission Each job is submitted from a user node to the JobTracker node that might be situated in a different node within the cluster through the following procedure:

- A user node asks for a new job ID from the JobTracker and computes input file splits.

- The user node copies some resources, such as the job’s JAR file, configuration file, and computed input splits, to the JobTracker’s file system.
- The user node submits the job to the JobTracker by calling the submitJob() function.

Task assignment The JobTracker creates one map task for each computed input split by the user node and assigns the map tasks to the execution slots of the TaskTrackers. The JobTracker considers the localization of the data when assigning the map tasks to the TaskTrackers. The JobTracker also creates reduce tasks and assigns them to the TaskTrackers. The number of reduce tasks is predetermined by the user, and there is no locality consideration in assigning them.

Task execution The control flow to execute a task (either map or reduce) starts inside the TaskTracker by copying the job JAR file to its file system. Instructions inside the job JAR file are executed after launching a Java Virtual Machine (JVM) to run its map or reduce task.

Task running check A task running check is performed by receiving periodic heartbeat messages to the JobTracker from the TaskTrackers. Each heartbeat notifies the JobTracker that the sending TaskTracker is alive, and whether the sending TaskTracker is ready to run a new task.

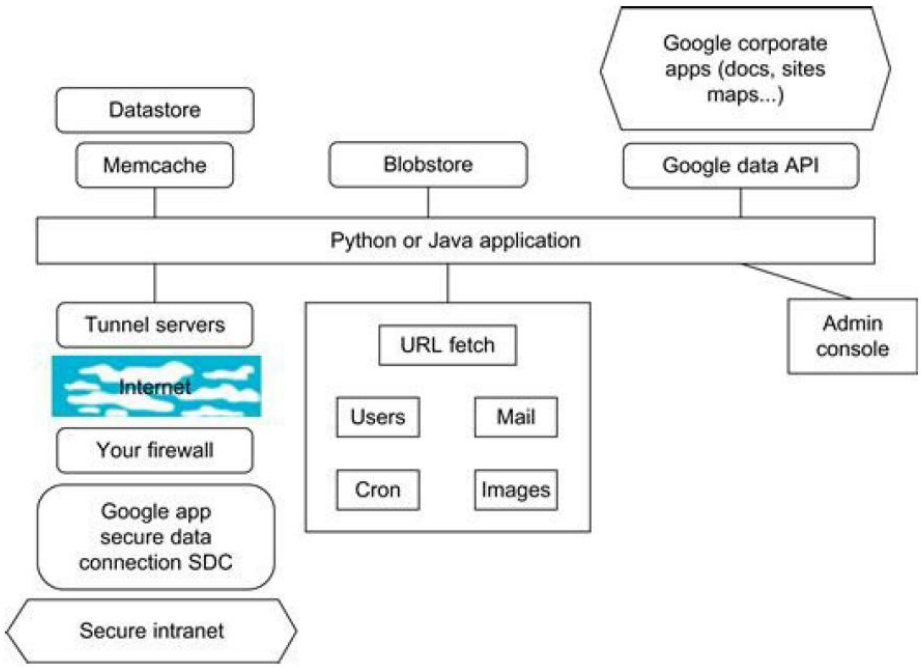
Mapping Applications

Mapping applications to different hardware and software in terms of six application architectures.

Category	Class	Description	Machine Architecture
1	Synchronous	The problem class can be implemented with instruction-level lockstep operation as in SIMD architectures.	SIMD
2	Loosely synchronous (BSP or bulk synchronous processing)	These problems exhibit iterative compute-communication stages with independent compute (map) operations for each CPU that are synchronized with a communication step. This problem class covers many successful MPI applications including partial differential equation solutions and particle dynamics applications.	MIMD on MPP (massively parallel processor)
3	Asynchronous	Illustrated by Compute Chess and Integer Programming; combinatorial search is often supported by dynamic threads. This is rarely important in scientific computing, but it is at the heart of operating systems and concurrency in consumer applications such as Microsoft Word.	Shared memory
4	Pleasingly parallel	Each component is independent. In 1988, Fox estimated this at 20 percent of the total number of applications, but that percentage has grown with the use of grids and data analysis applications including, for example, the Large Hadron Collider analysis for particle physics.	Grids moving to clouds
5	Metaproblems	These are coarse-grained (asynchronous or data flow) combinations of categories 1-4 and 6. This area has also grown in importance and is well supported by	Grids of clusters

Programming Support of Google App Engine

GAE programming model for two supported languages: Java and Python. A client environment includes an Eclipse plug-in for Java allows you to debug your GAE on your local machine. Google Web Toolkit is available for Java web application developers. Python is used with frameworks such as Django and CherryPy, but Google also has webapp Python environment.



There are several powerful constructs for storing and accessing data. The data store is a NOSQL data management system for entities. Java offers Java Data Object (JDO) and Java Persistence API (JPA) interfaces implemented by the Data Nucleus Access platform, while Python has a SQL-like query language called GQL. The performance of the data store can be enhanced by in-memory caching using the memcache, which can also be used independently of the data store.

Recently, Google added the blobstore which is suitable for large files as its size limit is 2 GB. There are several mechanisms for incorporating external resources. The Google SDC Secure Data Connection can tunnel through the Internet and link your intranet to an external GAE application. The URL Fetch operation provides the ability for applications to fetch resources and communicate with other hosts over the Internet using HTTP and HTTPS requests.

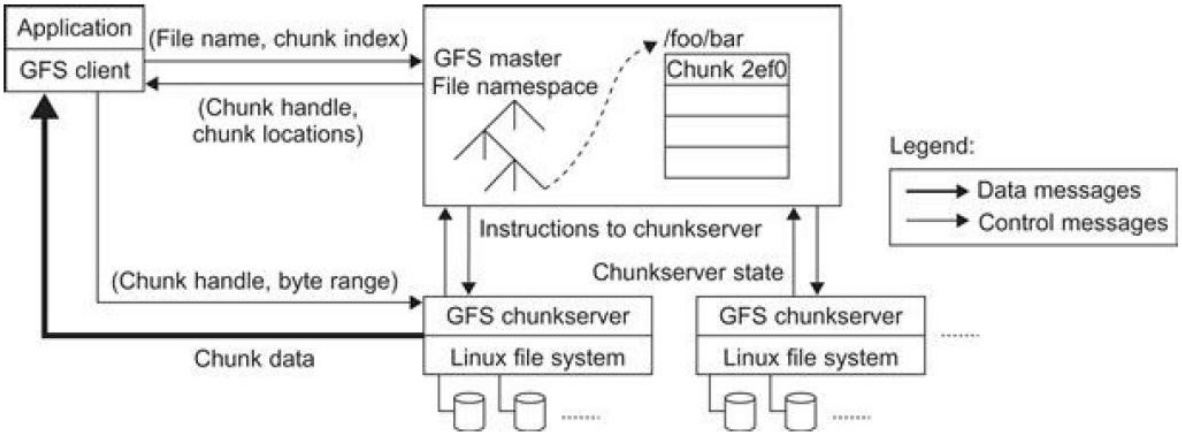
An application can use Google Accounts for user authentication. Google Accounts handles user account creation and sign-in, and a user that already has a Google account (such as a Gmail account) can use that account with your app. GAE provides the ability to manipulate image data using a dedicated Images service which can resize, rotate, flip, crop, and enhance

images. A GAE application is configured to consume resources up to certain limits or quotas. With quotas, GAE ensures that your application won't exceed your budget, and that other applications running on GAE won't impact the performance of your app. In particular, GAE use is free up to certain quotas.

Google File System (GFS)

GFS is a fundamental storage service for Google's search engine. GFS was designed for Google applications, and Google applications were built for GFS. There are several concerns in GFS. rate). As servers are composed of inexpensive commodity components, it is the norm rather than the exception that concurrent failures will occur all the time. Another concerns the file size in GFS. GFS typically will hold a large number of huge files, each 100 MB or larger, with files that are multiple GB in size quite common. Thus, Google has chosen its file data block size to be 64 MB instead of the 4 KB in typical traditional file systems. The I/O pattern in the Google application is also special. Files are typically written once, and the write operations are often the appending data blocks to the end of files. Multiple appending operations might be concurrent. The customized API can simplify the problem and focus on Google applications.

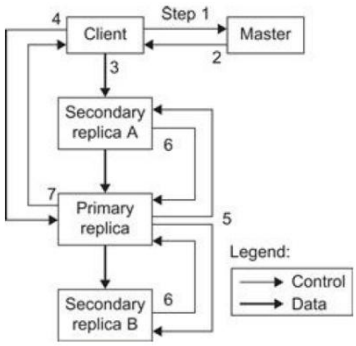
Figure shows the GFS architecture. It is quite obvious that there is a single master in the whole cluster. Other nodes act as the chunk servers for storing data, while the single master stores the metadata. The file system namespace and locking facilities are managed by the master. The master periodically communicates with the chunk servers to collect management information as well as give instructions to the chunk servers to do work such as load balancing or fail recovery.



The master has enough information to keep the whole cluster in a healthy state. Google uses a shadow master to replicate all the data on the master, and the design guarantees that all the data operations are performed directly between the client and the chunk server. The control messages are transferred between the master and the clients and they can be cached for future

use. With the current qualityof commodity servers, the single master can handle a cluster of more than 1,000 nodes.

Figure shows the data mutation (write, append operations) in GFS. Data blocks must be created for all replicas.



The goal is to minimize involvement of the master. The mutation takes the following steps:

1. The client asks the master which chunk server holds the current lease for the chunk and the locations of the other replicas. If no one has a lease, the master grants one to a replica it chooses (not shown).

2. The master replies with the identity of the primary and the locations of the other (secondary) replicas. The client caches this data for future mutations. It needs to contact the master again only when the primary becomes unreachable or replies that it no longer holds a lease.

3. The client pushes the data to all the replicas. Each chunk server will store the data in an internal LRU buffer cache until the data is used or aged out. By decoupling the data flow from the control flow, we can improve performance by scheduling the expensive data flow based on the network topology regardless of which chunk server is the primary.

4. Once all the replicas have acknowledged receiving the data, the client sends a write request to the primary. The request identifies the data pushed earlier to all the replicas. The primary assigns consecutive serial numbers to all the mutations it receives, possibly from multiple clients, which provides the necessary serialization. It applies the mutation to its own local state in serial order.

5. The primary forwards the write request to all secondary replicas. Each secondary replica applies mutations in the same serial number order assigned by the primary.

6. The secondaries all reply to the primary indicating that they have completed the operation.

7. The primary replies to the client. Any errors encountered at any replicas are reported to the client. In case of errors, the write corrects at the primary and an arbitrary subset of the secondary replicas. The client request is considered to have failed, and the modified region is left in an inconsistent state. Our client code handles such errors by retrying the failed mutation. It will

make a few attempts at steps 3 through 7 before falling back to a retry from the beginning of the write.

GFS was designed for high fault tolerance and adopted some methods to achieve this goal. Master and chunk servers can be restarted in a few seconds, and with such a fast recovery capability, the window of time in which the data is unavailable can be greatly reduced. As we mentioned before, each chunk is replicated in at least three places and can tolerate at least two data crashes for a single chunk of data. The shadow master handles the failure of the GFS master

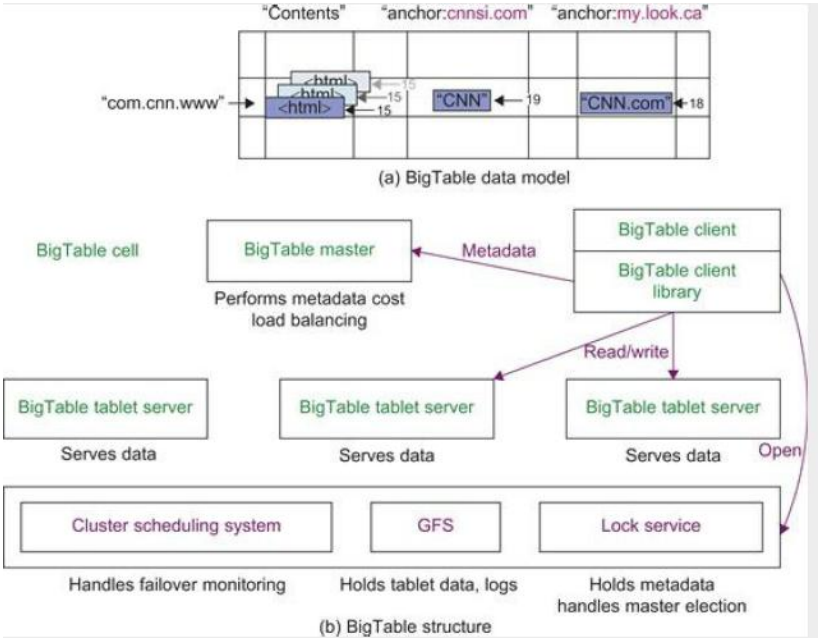
Big Table

BigTable was designed to provide a service for storing and retrieving structured and semistructured data. BigTable applications include storage of web pages, per-user data, and geographic locations. The database needs to support very high read/write rates and the scale might be millions of operations per second. Also, the database needs to support efficient scans over all or interesting subsets of data, as well as efficient joins of large one-to-one and one-to-many data sets. The application may need to examine data changes over time. The BigTable system is scalable, which means the system has thousands of servers, terabytes of in-memory data, petabytes of disk-based data, millions of reads/writes per second, and efficient scans. BigTable is used in many projects, including Google Search, Orkut, and Google Maps/Google Earth, among others.

The BigTable system is built on top of an existing Google cloud infrastructure. BigTable uses the following building blocks:

1. GFS: stores persistent state
2. Scheduler: schedules jobs involved in BigTable serving
3. Lock service: master election, location bootstrapping
4. MapReduce: often used to read/write BigTable data

BigTable provides a simplified data model called Web Table, compared to traditional database systems. Figure (a) shows the data model of a sample table. Web Table stores the data about a web page. Each web page can be accessed by the URL. The URL is considered the row index. The column provides different data related to the corresponding URL



The map is indexed by row key, column key, and timestamp−that is, (row:string, column: string, time:int64) maps to string (cell contents). Rows are ordered in lexicographic order by row key. The row range for a table is dynamically partitioned and each row range is called "Tablet". Syntax for columns is shown as a (family:qualifier) pair. Cells can store multiple versions of data with timestamps.

Figure (b) shows the BigTable system structure. A BigTable master manages and stores the metadata of the BigTable system. BigTable clients use the BigTable client programming library to communicate with the BigTable master and tablet servers. BigTable relies on a highly available and persistent distributed lock service called Chubby.

Programming on Amazon AWS

AWS platform has many features and offers many services

Features:

- Relational Database Service (RDS) with a messaging interface
- Elastic MapReduce capability
- NOSQL support in SimpleDB

Capabilities

- Auto-scaling enables you to automatically scale your Amazon EC2 capacity up or down according to conditions
- Elastic load balancing automatically distributes incoming application traffic across multiple Amazon EC2 instances

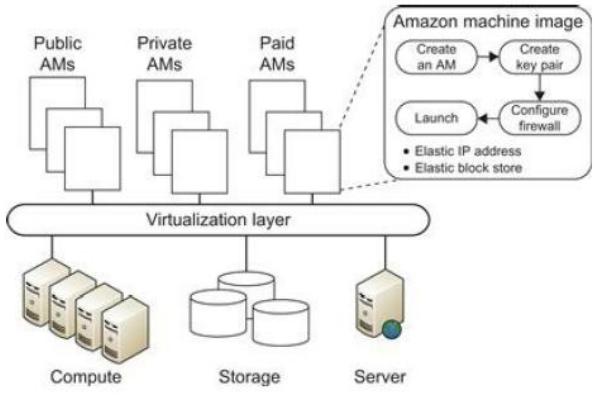
- CloudWatch is a web service that provides monitoring for AWS cloud resources, operational performance, andoverall demand patterns−including metrics such as CPU utilization, disk reads and writes, and network traffic.

A mazon provides several types of preinstalled VMs. Instances are often called Amazon Machine Images (AMIs) which are preconfigured with operating systems based on Linux or Windows, and additional software. Figure 6.24 shows an execution environment.A MIs are the templates for instances, which are running VMs. The AMIs are formed from the virtualized compute, storage, and server resource.

Private AMI: Images created by you, which are private by default. You can grant access to other users to launch your private images.

Public AMI: Images created by users and released to the AWS community, so anyone can launch instances based on them

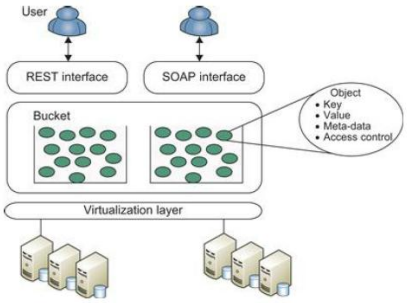
Paid QAMI: You can create images providing specific functions that can be launched by anyone willing to pay you per each hour of usage



Amazon Simple Storage Service (S3)

Amazon S3 provides a simple web services interface that can be used to store and retrieve any amount of data, at any time, from anywhere on the web. S3 provides the object-oriented storage service for users. Users can access their objects through Simple Object Access Protocol (SOAP) with either browsers or other client programs which support SOAP. SQS is responsible for ensuring a reliable message service between two processes.

Figure shows the S3 execution environment.



The fundamental operation unit of S3 is called an object. Each object is stored in a bucket and retrieved via a unique, developer-assigned key. In other words, the bucket is the container of the object. Besides unique key attributes, the object has other attributes such as values, metadata, and access control information. Through the key-value programming interface, users can write, read, and delete objects containing from 1 byte to 5 gigabytes of data each. There are two types of web service interface for the user to access the data stored in Amazon clouds. One is a REST (web 2.0) interface, and the other is a SOAP interface. Here are some key features of S3:

- Redundant through geographic dispersion.
- Designed to provide 99.99% durability and 99.99 %availability of objects over a given year with cheaper reduced redundancy storage (RRS).
- Authentication mechanisms to ensure that data is kept secure from unauthorized access. Objects can be made private or public, and rights can be granted to specific users.
- Per-object URLs and ACLs (access control lists).
- Default download protocol of HTTP

Amazon Elastic Block Store (EBS) and SimpleDB

The Elastic Block Store (EBS) provides the volume block interface for saving and restoring the virtual images of EC2 instances. The status of EC2 can now be saved in the EBS system after the machine is shut down.Users can use EBS to save persistent data and mount to the running instances of EC2. S3 is "Storage as a Service" with a messaging interface. Multiple volumes can be mounted to the same instance. These storage volumes behave like raw, unformatted block devices, with user-supplied device names and a block device interface.

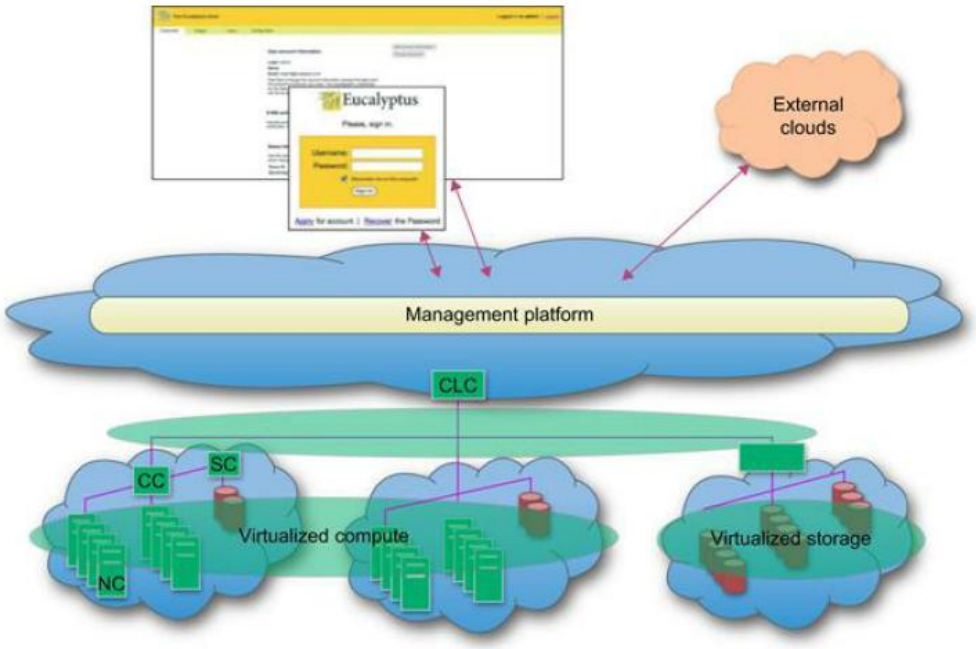
Amazon SimpleDB Service

SimpleDB provides a simplified data model based on the relational database data model. Structured data from users must be organized into domains. Each domain can be considered a table. The items are the rows in the table. A cell in the table is recognized as the value for a specific attribute (column name) of the corresponding row. it is possible to assign multiple values to a single cell in the table. This is not permitted in a traditional relational database.

SimpleDB, like Azure Table, could be called "LittleTable" as they are aimed at managing small amounts of information stored in a distributed table.

Cloud Software Environments

Eucalyptus is a product from Eucalyptus Systems developed as a research project at the University of California. Eucalyptus provides an AWS-compliant EC2-based web service interface for interacting with the cloud service. Figure shows the architecture based on the need to manage VM images. The system supports cloud programmers in VM image management as follows. Essentially, the system has been extended to support the development of both the computer cloud and storage cloud.



VM Image Management

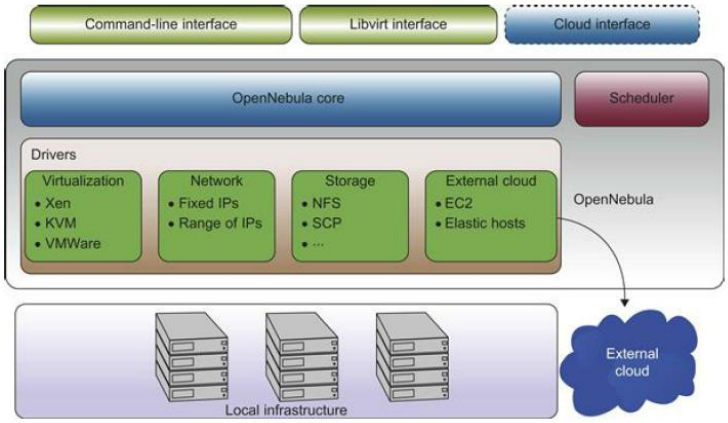
Eucalyptus takes many design queues from Amazon's EC2, and its image management system is no different. Eucalyptus stores images in Walrus, the block storage system that is analogous to the Amazon S3 service. As such, any user can bundle her own root file system, and upload and then register this image and link it with a particular kernel and ramdisk image. This image is uploaded into a user-defined bucket within Walrus, and can be retrieved anytime from any availability zone.

OpenNebula

It is an open source toolkit which allows users to transform existing infrastructure into an IaaS cloud. Figure shows the OpenNebula architecture and its main components. The architecture has been designed to be flexible and modular to allow integration with different

storage and network infrastructure configurations, and hypervisor technologies. Here, the core is a centralized component that manages the VM full life cycle, including setting up networks dynamically for groups of VMs and managing their storage requirements.

Another important component is the capacity manager or scheduler. It governs the functionality provided by the core. The default capacity scheduler is a requirement/rank matchmaker. The last main components are the access drivers. They provide an abstraction of the underlying infrastructure to expose the basic functionality of the monitoring, storage, and virtualization services available in the cluster.



Additionally, OpenNebula offers management interfaces to integrate the core’s functionality within other data-center management tools, such as accounting or monitoring frameworks. To this end, OpenNebula implements the libvirt API an open interface for VM management, as well as a command-line interface (CLI). A subset of this functionality is exposed to external users through a cloud interface. OpenNebula can support a hybrid cloud model by using cloud drivers to interface with external clouds when the local resources are insufficient. This lets organizations supplement their local infrastructure with computing capacity from a public cloud to meet peak demands, or implement HA strategies.

OpenStack

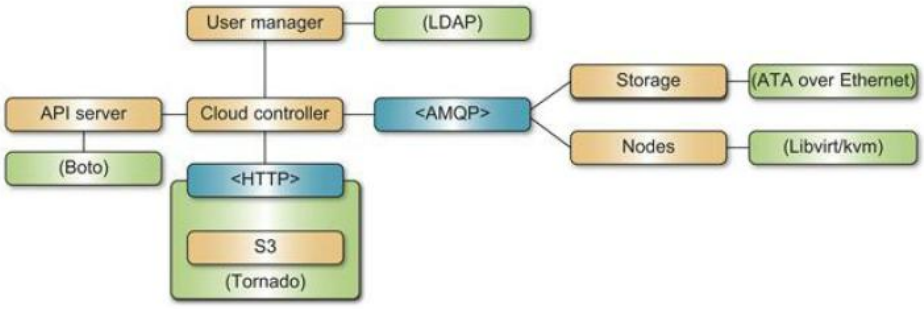
OpenStack focuses on the development of two aspects of cloud computing to address compute and storage aspects with the OpenStack Compute and OpenStack Storage solutions. “OpenStack Compute is the internal fabric of the cloud creating and managing large groups of virtual private servers” and “OpenStack Object Storage is software for creating redundant, scalable object storage using clusters of commodity servers to store terabytes or even petabytes of data.”

OpenStack Compute

OpenStack [103] is developing a cloud computing fabric controller, a component of an IaaS system, known as Nova. The architecture for Nova is built on the concepts of shared-nothing and messaging-based information exchange. Hence, most communication in Nova is facilitated by message queues. To prevent blocking components while waiting for a response from others, deferred objects are introduced. Such objects include callbacks that get triggered when a response is received. To achieve the shared-nothing paradigm, the overall system state is kept in a distributed data system. State updates are made consistent through atomic transactions. Nova is implemented in Python while utilizing a number of externally supported libraries and components.

Figure shows the main architecture of Open Stack Compute. In this architecture, the API Server receives HTTP requests from boto, converts the commands to and from the API format, and forwards the requests to the cloud controller.

The cloud controller maintains the global state of the system, ensures authorization while interacting with the User Manager via Lightweight Directory Access Protocol(LDAP), interacts with the S3 service, and manages nodes.



It includes the following types:

- *NetworkController* manages address and virtual LAN (VLAN) allocations
- *RoutingNode* governs the NAT (network address translation) conversion of public IPs to private IPs, and enforces firewall rules
- *AddressingNode* runs Dynamic Host Configuration Protocol (DHCP) services for private networks
- *TunnelingNode* provides VPN connectivity
- The network state (managed in the distributed object store) consists of the following:
- VLAN assignment to a project
- Private subnet assignment to a security group in a VLAN
- Private IP assignments to running instances
- Public IP allocations to a project

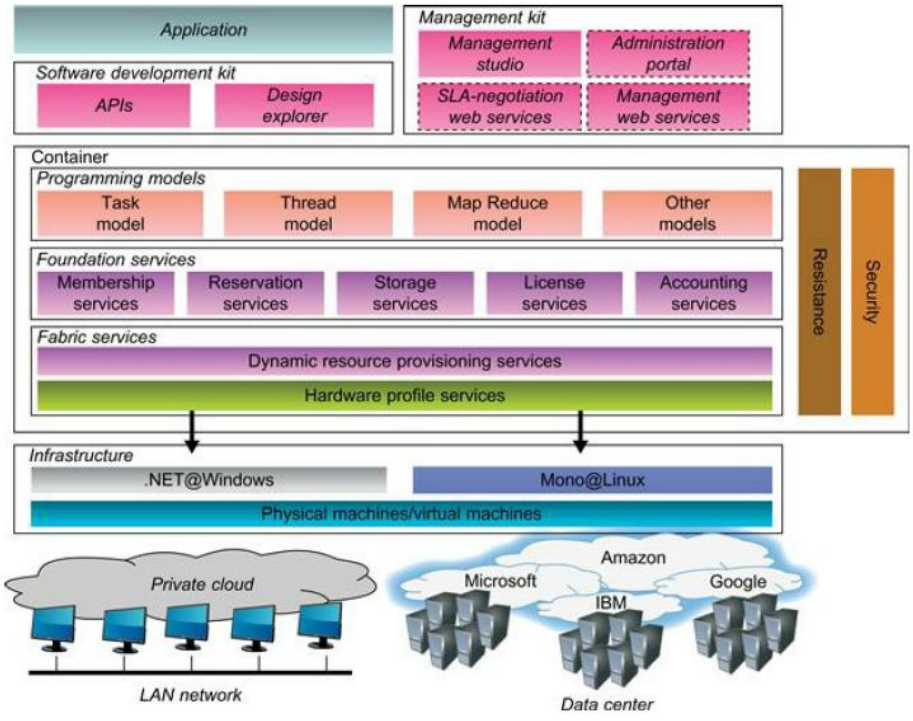
- Public IP associations to a private IP/running instance

OpenStack Storage

The OpenStack storage solution is built around a number of interacting components and concepts, including a proxy server, a ring, an object server, a container server, an account server, replication, updaters, and auditors. The role of the proxy server is to enable lookups to the accounts, containers, or objects in OpenStack storage rings and route the requests. Thus, any object is streamed to or from an object server directly through the proxy server to or from the user. A ring represents a mapping between the names of entities stored on disk and their physical locations. A ring includes the concept of using zones, devices, partitions, and replicas. Hence, it allows the system to deal with failures, and isolation of zones representing a drive, a server, a cabinet, a switch, or even a data center. Weights can be used to balance the distribution of partitions on drives across the cluster, allowing users to support heterogeneous storage resources. Objects are stored as binary files with metadata stored in the file's extended attributes.This requires that the underlying file system is built around object servers, which is often not the case for standard Linux installations.

Aneka

Aneka is a cloud application platform developed by Manjrasoft based in Melbourne, Australia. It is designed to support rapid development and deployment of parallel and distributed applications on private or public clouds. It provides a rich set of APIs for transparently exploiting distributed resources and expressing the business logic of applications by using preferred programming abstractions.



Aneka acts as a workload distribution and management platform for accelerating applications in both Linux and Microsoft.NET framework environments. Some of the key advantages of Aneka over other workload distribution solutions include:

· Support of multiple programming and application environments

· Simultaneous support of multiple runtime environments

· Rapid deployment tools and framework

· Ability to harness multiple virtual and/or physical machines for accelerating application provisioning based on users' Quality of Service/service-level agreement (QoS/SLA) requirements

· Built on top of the Microsoft .NET framework, with support for Linux environments through Mono.

Aneka offers three types of capabilities which are essential for building, accelerating, and managing clouds and their applications:

1. **Build** Aneka includes a new SDK which combines APIs and tools to enable users to rapidly develop applications. Aneka also allows users to build different runtime environments such as enterprise/private cloud by harnessing compute resources in network or enterprise data centers, Amazon EC2, and hybrid clouds by combining enterprise private clouds managed by Aneka with resources from Amazon EC2 or other enterprise clouds built and managed using XenServer.

2. **Accelerate** Aneka supports rapid development and deployment of applications in multiple runtime environments running different operating systems such as Windows or Linux/UNIX. Aneka uses physical machines as much as possible to achieve maximum utilization in local environments.

3. **Manage** Management tools and capabilities supported by Aneka include a GUI and APIs to set up, monitor, manage, and maintain remote and global Aneka compute clouds. Aneka also has an accounting mechanism and manages priorities and scalability based on SLA/QoS which enables dynamic provisioning.

Here are three important programming models supported by Aneka

- Thread programming model
- Task programming model
- MapReduce programming

Aneka Architecture

This environment is built by aggregating together physical and virtual nodes hosting the Aneka container. The container is a lightweight layer that interfaces with the hosting environment and manages the services deployed on a node.

The interaction with the hosting platform is mediated through the Platform Abstraction Layer (PAL), which hides in its implementation all the heterogeneity of the different operating systems.By means of the PAL it is possible to perform all the infrastructure-related tasks, such as performance and system monitoring.

The available services can be aggregated into three major categories:

Fabric Services: Fabric services implement the fundamental operations of the infrastructure of the cloud. These services include HA and failover for improved reliability, node membership and directory, resource provisioning, performance monitoring, and hardware profiling.

Foundation Services: Foundation services constitute the core functionalities of the Aneka middleware. They provide a basic set of capabilities that enhance application execution in the cloud. These services provide added value to the infrastructure and are of use to system administrators and developers.

Application Services: Application services deal directly with the execution of applications and are in charge of providing the appropriate runtime environment for each application model.

In addition, specific models can require additional services or a different type of support. Aneka provides support for the most well-known application programming patterns, such as distributed threads, bags of tasks, and MapReduce. Additional services can be designed and deployed in the system. This is how the infrastructure is enriched with additional features and capabilities. The SDK provides straightforward interfaces and ready-to-use components for rapid service prototyping.

Cloudsim

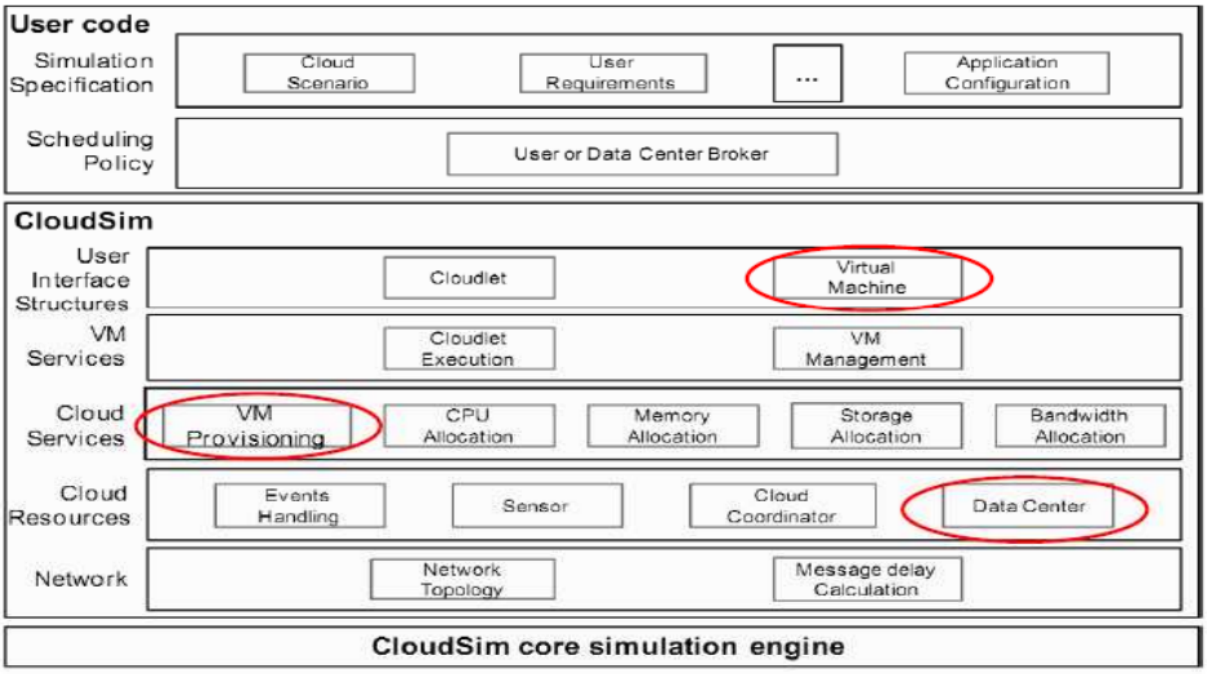
The primary objective of Cloudsim is to provide a generalized, and extensible simulation framework that enables seamless modeling, simulation, and experimentation of emerging Cloud computing infrastructures and application services. CloudSim is developed in the Cloud Computing and Distributed Systems (CLOUDS) Laboratory, at University of Melbourne.

- support for modeling and simulation of large scale Cloud computing data centers
- support for modeling and simulation of virtualized server hosts, with customizable policies for provisioning host resources to virtual machines
- support for modeling and simulation of energy-aware computational resources

- support for modeling and simulation of data center network topologies and message-passing applications
- support for dynamic insertion of simulation elements, stop and resume of simulation
- support for user-defined policies for allocation of hosts to virtual machines and policies for allocation of host resources to virtual machines

Architecture

The CloudSim layer provides support for modelling and simulation of cloud environments including dedicated management interfaces for memory, storage, bandwidth and VMs. The main components of the CloudSim framework includes



Regions: It models geographical regions in which cloud service providers allocate resources to their customers. In cloud analysis, there are six regions that correspond to six continents in the world.

Data centers: It models the infrastructure services provided by various cloud service providers. It encapsulates a set of computing hosts or servers that are either heterogeneous or homogeneous in nature, based on their hardware configurations.

Data centre characteristics: It models information regarding data centre resource configurations.

Hosts: It models physical resources (compute or storage).

The user base: It models a group of users considered as a single unit in the simulation, and its main responsibility is to generate traffic for the simulation.

Cloudlet: It specifies the set of user requests. It contains the application ID, name of the user base that is the originator to which the responses have to be routed back, as well as the size of the request execution commands, and input and output files. It models the cloud-based application services. CloudSim categorises the complexity of an application in terms of its computational requirements. Each application service has a pre-assigned instruction length and data transfer overhead that it needs to carry out during its life cycle.
Service broker: The service broker decides which data centre should be selected to provide the services to the requests from the user base.
VMM allocation policy: It models provisioning policies on how to allocate VMs to hosts.
VM scheduler: It models the time or space shared, scheduling a policy to allocate processor cores to VMs.