Unit - 4

Cloud programming and software environments

4.1. FEATURES OF CLOUD AND GRID:

4.1.1Cloud Capabilities and Platform Features

Commercial clouds need broad capabilities, as summarized in 4.1.(a) These capabilities offer cost-effective utility computing with the elasticity to scale up and down in power. However, as well as this key distinguishing feature, commercial clouds offer a growing number of additional capabilities commonly termed Platform as a Service(PaaS). 4.1.(b) lists some low-level infrastructure features. 4.1.(c) lists traditional programming environments for parallel and distributed systems that need to be supported in Cloud environments. 4.1.(d) presents features emphasized by clouds and by some grids.

4.1.(a)Important cloud platform capabilities

1.Physical or virtual computing platform

2.Massive data storage service, distributed file system

3.Massive data processing method and programming Model

4.Workflow and data query language support

5.Programming interface and service deployment

6.Runtime support

7.Support services

4.1.(b) Infrastructure Cloud Features:

- Accounting
- Appliances
- Authentication and authorization
- Data transport
- Operating systems
- Program library

- Registry
- Security
- Scheduling
- Gang scheduling
- Software as a Service (SaaS)
- Virtualization

4.1.(c) Traditional Features in Cluster, Grid, and Parallel Computing Environments:

- Cluster management
- Data management
- Grid programming environment
- OpenMP/threading
- Portals
- Scalable parallel computing environments
- Virtual organizations
- Workflow

4.1.(d)Platform Features Supported by Clouds and (Sometimes) Grids

- **Blob:** Basic storage concept typified by Azure Blob and Amazon S3
- **DPFS:** Support of file systems such as Google (MapReduce), HDFS (Hadoop), and Cosmos (Dryad) with compute-data affinity optimized for data processing
- **MapReduce:** Support MapReduce programming model including Hadoop on Linux, Dryad on Windows HPCS, and Twister on Windows and Linux. Include new associated languages such as Sawzall, Pregel, Pig Latin, and LINQ
- **Monitoring:** Many grid solutions such as Inca. Can be based on publish-subscribe.
- **Notification:** Basic function of publish-subscribe systems
- Programming model: Cloud programming models are built with other platform features and are related to familiar web and grid models
- **Queues:** Queuing system possibly based on publish-subscribe
- **SQL:** Relational database
- **Table:** Support of table data structures modeled on Apache Hbase or Amazon Simple DB/Azure Table. Part of NOSQL movement
- **Web role:** Used in Azure to describe important links to users and can be supported otherwise with a portal framework. This is the main purpose of GAE
- **Worker role:** Implicitly used in both Amazon and grids but was first introduced as a high-level construct by Azure

4.1.2.1 Workflow:

Workflow has spawned many projects in the United States and Europe. Pegasus, Taverna, and Kepler are popular, but no choice has gained wide acceptance. There are commercial systems such as Pipeline Pilot, AVS (dated), and the LIMS environments. A recent entry is Trident [2] from Microsoft Research which is built on top of Windows Workflow Foundation. If Trident runs on Azure or just any old Windows machine, it will run workflow proxy services on external (Linux) environments. Workflow links multiple cloud and non cloud services in real applications on demand.

4.1.2.2 Data Transport

The cost (in time and money) of data transport in (and to a lesser extent, out of) commercial clouds is often discussed as a difficulty in using clouds. If commercial clouds become an important component of the national cyber infrastructure we can expect that high-bandwidth links will be made available between clouds and TeraGrid. The special structure of cloud data with blocks (in Azure blobs) and tables could allow high-performance parallel algorithms, but initially, simple HTTP mechanisms are used to transport data [335] on academic systems/TeraGrid and commercial clouds.

4.1.2.3 Security, Privacy, and Availability:

The following techniques are related to security, privacy, and availability requirements for developing a healthy and dependable cloud programming environment. We summarize these techniques here.

Use virtual clustering to achieve dynamic resource provisioning with minimum overhead cost.

- Use stable and persistent data storage with fast queries for information retrieval.
- Use special APIs for authenticating users and sending email using commercial accounts.
- Cloud resources are accessed with security protocols such as HTTPS and SSL.
- Fine-grained access control is desired to protect data integrity and deter intruders or hackers.
- Shared data sets are protected from malicious alteration, deletion, or copyright violations.
- Features are included for availability enhancement and disaster recovery with life migration of VMs.
- Use a reputation system to protect data centers. This system only authorizes trusted clients and stops pirates.

4.1.3 Data Features and Databases

The programming features related to the program library, blobs, drives, DPFS, tables, and various types of databases including SQL, NOSQL, and nonrelational databases and special queuing services are given.

4.1.3.1 Program Library

Many efforts have been made to design a VM image library to manage images used in academic and commercial clouds.

4.1.3.2 Blobs and Drives

The basic storage concept in clouds is blobs for Azure and S3 for Amazon. These can be organized by containers in Azure. In addition to a service interface for blobs and S3, one can attach directly to compute instances such as Azure drives and the Elastic Block Store for Amazon. This concept is similar to shared file systems such as Lustre used in TeraGrid. However, the architecture ideas are similar between clouds and TeraGrid, and the Simple Cloud File Storage API could become important here.

4.1.3.3 DPFS(Distributed parallel file system)

DPFS file systems are precisely designed for efficient execution of data-intensive applications. However, the importance of DPFS for linkage with Amazon and Azure is not clear, as these clouds do not currently offer fine-grained support for compute-data affinity. We note here that Azure Affinity Groups are one interesting capability. We expect that initially blobs, drives, tables, and queues will be the areas where academic systems will most usefully provide a platform similar to Azure (and Amazon). Note the HDFS (Apache) and Sector (UIC) projects in this area.

4.1.3.4 SQL and Relational Databases

Both Amazon and Azure clouds offer relational databases and it is straightforward for academic systems to offer a similar capability unless there are issues of huge scale where, in fact, approaches based on tables and/or MapReduce might be more appropriate. Note that databases can be used to illustrate two approaches to deploying capabilities.

4.1.3.5 Table and NOSQL Non-relational Databases

A substantial number of important developments have occurred regarding simplified database structures4termed “NOSQL” typically emphasizing distribution and scalability. These are present in the three major clouds: BigTable in Google, SimpleDB in Amazon, and Azure Table for Azure. Tables are clearly important in science as illustrated by the VOTable standard in astronomy and the popularity of Excel. However, there does not appear to be substantial experience in using tables outside clouds.There are, of course, many important

uses of non-relational databases, especially in terms of triple stores for metadata storage and access. Recently, there has been interest in building scalable RDF triple stores based on MapReduce and tables or the Hadoop File System, with early success reported on very large stores. The current cloud tables fall into two groups: Azure Table and Amazon SimpleDB are quite similar and support lightweight storage for "document stores" while BigTable aims to manage large distributed data sets without size limitations.

4.1.3.6 Queuing Services

Both Amazon and Azure offer similar scalable, robust queuing services that are used to communicate between the components of an application. The messages are short (less than 8 KB) and have a Representational State Transfer (REST) service interface with "deliver at least once" semantics. They are controlled by timeouts for posting the length of time allowed for a client to process. One can build a similar approach, basing it on publish-subscribe systems such as ActiveMQ or NaradaBrokering with which we have substantial experience.

4.1.4 Programming and Runtime Support

Programming and runtime support are desired to facilitate parallel programming and provide runtime support of important functions in today's grids and clouds. Various MapReduce systems are reviewed in this section.

6.1.4.1 Worker and Web Roles

The roles introduced by Azure provide nontrivial processes and are automatically launched. Note that explicit scheduling is unnecessary in clouds for individual worker roles and for the "gang scheduling" supported transparently in MapReduce. Queues are a critical concept here, as they provide a natural way to manage task assignment in a fault 3 tolerant, distributed fashion. Web roles provide an interesting approach to portals. GAE is largely aimed at web applications, whereas science gateways are successful in TeraGrid.

What is MapReduce?

1. Simple data-parallel programming model
2. For large-scale data processing
3. Exploits large set of commodity computers
4. Executes process in distributed manner
5. Offers high availability
6. Pioneered by Google
7. Processes 20 petabytes of data per day

8. Popularized by open-source Hadoop project
9. Used at Yahoo!, Facebook, Amazon, …

What is MapReduce used for?

☐ **At Google:**

1. Index construction for Google Search
2. Article clustering for Google News
3. Statistical machine translation

☐ **At Yahoo!:**

1. "Web map" powering Yahoo! Search
2. Spam detection for Yahoo! Mail

☐ **At Facebook:**

1. Data mining
2. Ad optimization
3. Spam detection

Motivation: Large Scale Data Processing

- Many tasks composed of processing lots of data to produce lots of other data
- Want to use hundreds or thousands of CPUs... but this needs to be easy!

MapReduce provides

- User-defined functions
- Automatic parallelization and distribution
- Fault-tolerance
- I/O scheduling
- Status and monitoring

There has been substantial interest in "data parallel" languages largely aimed at loosely coupled computations which execute over different data samples. The language and runtime generate and provide efficient execution of "many task" problems that are well known as successful grid applications. However, MapReduce, summarized in has several advantages over traditional implementations for many task problems, as it supports dynamic execution, strong fault tolerance, and an easy-to-use high-level interface. The major open source/commercial MapReduce implementations are Hadoop and Dryad with execution possible with or without VMs.

Hadoop is currently offered by Amazon, and we expect Dryad to be available on Azure. A prototype Azure MapReduce was built at Indiana University, which we will discuss shortly. On FutureGrid, we already intend to support Hadoop, Dryad, and other

MapReduce approaches, including Twister support for iterative computations seen in many data-mining and linear algebra applications. Note that this approach has some similarities with Cloudera which offers a variety of Hadoop distributions including Amazon and Linux. MapReduce is closer to broad deployment than other cloud platform features, as there is quite a bit of experience with Hadoop and Dryad outside clouds.

Cloud Programming Models:

In many ways, most of the previous sections describe programming model features, but these are "macroscopic" constructs and do not address, for example, the coding (language and libraries). Both the GAE and Manjra soft Aneka environments represent programming models; both are applied to clouds, but are really not specific to this architecture. Iterative MapReduce is an interesting programming model that offers portability between cloud, HPC and cluster environments.

SaaS:

Services are used in a similar fashion in commercial clouds and most modern distributed systems. We expect users to package their programs wherever possible, so no special support is needed to enable SaaS. We already discussed in Section 4.1.3 why "Systems software as a service" was an interesting idea in the context of a database service. We desire a SaaS environment that provides many useful tools to develop cloud applications over large data sets. In addition to the technical features, such as MapReduce, BigTable, EC2, S3, Hadoop, AWS, GAE, and WebSphere2, we need protection features that may help us to achieve scalability, security, privacy, and availability.

	Google MapReduce [28]	Apache Hadoop [23]	Microsoft Dryad [26]	Twister [29]	Azure Twister [30]
Programming Model	MapReduce	MapReduce	DAG execution, extensible to MapReduce and other patterns	Iterative MapReduce	Currently just MapReduce; will extend to Iterative MapReduce
Data Handling	GFS (Google File System)	HDFS (Hadoop Distributed File System)	Shared directories and local disks	Local disks and data management tools	Azure blob storage
Scheduling	Data locality	Data locality; rack-aware, dynamic task scheduling using global queue	Data locality; network topology optimized at runtime; static task partitions	Data locality; static task partitions	Dynamic task scheduling through global queue
Failure Handling	Reexecution of failed tasks; duplicated execution of slow tasks	Reexecution of failed tasks; duplicated execution of slow tasks	Reexecution of failed tasks; duplicated execution of slow tasks	Reexecution of iterations	Reexecution of failed tasks; duplicated execution of slow tasks
HLL Support	Sawzall [31]	Pig Latin [32,33]	DryadLINQ [27]	Pregel [34] has related features	N/A
Environment	Linux cluster	Linux clusters, Amazon Elastic MapReduce on EC2	Windows HPCS cluster	Linux cluster, EC2	Windows Azure, Azure Local Development Fabric
Intermediate Data Transfer	File	File, HTTP	File, TCP pipes, shared-memory FIFOs	Publish-subscribe messaging	Files, TCP

4.2 PARALLEL AND DISTRIBUTED PROGRAMMING PARADIGMS:

The term carries the notion of two fundamental terms in computer science: distributed computing system and parallel computing. 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 to run a job or an application. For instance, parallel computing

may use either a distributed or a non-distributed computing system such as a multiprocessor platform. Running a parallel program on a distributed computing system (parallel and distributed programming) has several advantages for both users and distributed computing systems.

From the users' perspective, it decreases application response time; from the distributed computing systems standpoint, it increases throughput and resource utilization. Running a parallel program on a distributed computing system, however, could be a very complicated process.

4.2.1 Parallel Computing and Programming Paradigms

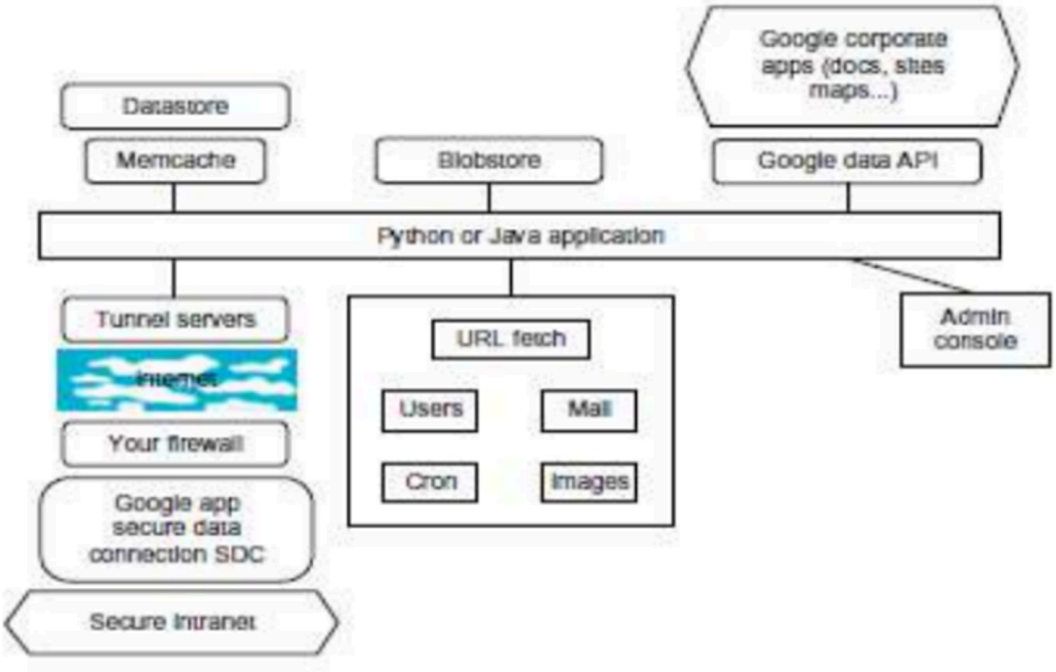
The system issues for running a typical parallel program in either a parallel or a distributed manner would include the following:

1. Partitioning: a). Computation partitioning b). Data partitioning
2. Mapping
3. Synchronization
4. Communication
5. Scheduling

4.3 PROGRAMMING SUPPORT OF GOOGLE APP ENGINE

4.3.1 Programming the Google App Engine

Figure summarizes some key features of GAE programming model for two supported languages: Java and Python. A client environment that includes an Eclipse plug-in for Java allows you to debug your GAE on your local machine. Also, the GWT Google Web Toolkit is available for Java web application developers. Developers can use this, or any other language using a JVM-based interpreter or compiler, such as JavaScript or Ruby. Python is often used with frameworks such as Django and CherryPy, but Google also supplies a built in webapp Python environment.



This diagram represents a high-level architecture of a Google App Engine-based application integrated with various components and services. Here's a breakdown of the components and their relationships:

1. Data Storage and Cache:

- Datastore: Manages structured data storage, suitable for querying and transactions.

- Mem cache: Provides high-performance caching to reduce load and improve performance.

2. Blob store: Handles storage and retrieval of large, unstructured binary objects like images and videos.

3. Application Layer:

- Built using Python or Java, serving as the core logic layer connecting various components.

- Integrates with:

Google Data API: For interacting with other Google services like Google Docs, Sites, and Maps.

Google Corporate Apps: Provides access to applications such as Docs, Sites, and Maps.

4. Functionality Modules:

- URL Fetch: Allows the application to make HTTP requests to external services.

- Users: Manages user authentication and authorization.

- Mail: Handles email communication.

- Cron: Manages scheduled tasks for the application.

- Images: Provides image processing services.

5. Admin Console:

- A management interface for overseeing application operations and configurations.

6. Secure Data Connections:

- The application connects to a secure intranet through:

    - Google App Secure Data Connection (SDC): Ensures encrypted communication.

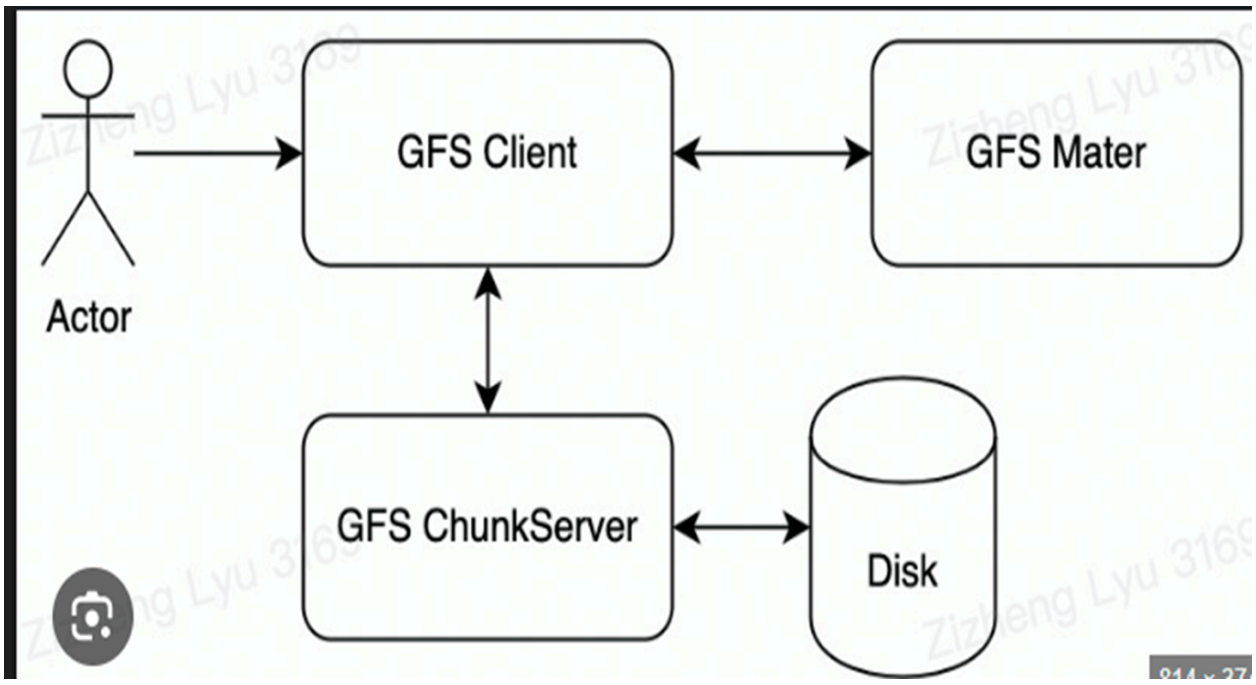
    - Your Firewall: Adds another layer of security.

    - Tunnel Servers: Establish secure network tunnels for data exchange.

7. Themes:

- The diagram includes a graphical representation of a secure connection or network link (symbolized by the blue, wavy line labeled "Themes").

4.3.2GFS(Google file system):



This image shows a high-level architectural diagram of the Google File System (GFS). It illustrates the interaction between the main components:

1. Actor:

Represents the user or application initiating file operations, such as reading or writing files.

2. GFS Client: Acts as an intermediary between the Actor and the GFS system. It communicates with the GFS Master and GFS Chunk servers for metadata and data access, respectively.

- The client sends file names and chunk indices to the GFS Master.

- The client receives metadata, such as chunk handles and their locations.

3. GFS Master: Maintains metadata, including the namespace (e.g., file paths and chunk mapping) and chunk states. It handles control operations, such as:

Providing chunk handles and locations to the GFS Client.

- Sending instructions to Chunk servers for chunk creation, replication, and deletion

4. GFS Chunk servers: Store chunks of files on local storage. They serve chunk data to the client based on the chunk handle and byte range specified.

5. Disk:

Physical storage where chunks are stored as files. Each chunk is replicated across multiple Chunk servers to ensure reliability and availability.

Flow of Interaction:

The Actor initiates a request (e.g., file read/write) via the GFS Client.

The GFS Client consults the GFS Master to get metadata about the requested file or chunk (e.g., chunk handle and locations).

The GFS Client communicates directly with the appropriate GFS Chunk Server to perform the data operation.

Chunks are retrieved or updated on the Disk managed by the Chunk Server. This architecture ensures scalability, fault tolerance, and efficient handling of large-scale distributed data.

Chubby, Google's Distributed Lock Service:

The structure of Chubby, Google's Distributed Lock Service, is designed to provide highly available and consistent lock management and metadata storage for distributed systems. Here's an overview of its structure:

1. Chubby Cell

- A Chubby Cell s the basic deployment unit of Chubby, consisting of:

  - Five Replica Nodes Distributed across different servers to ensure fault tolerance.

  One Master Node: Elected using the Paxos consensus algorithm, responsible for handling all client requests.

2. Components of a Chubby Cell

Master Node

- Coordinates all operations in the Chubby cell.

- Handles client requests for acquiring/releasing locks and reading/writing metadata.

- Manages session states and lease renewals.

- Propagates changes to replica nodes to maintain consistency.

Replica Nodes:

- Store a copy of the Chubby database, which includes metadata and lock information.

- Use Paxos consensus to replicate state changes (e.g., updates, lock acquisitions).

- Take over as master if the current master fails.

3. Chubby Client

- Each client interacts with the Chubby system to:

  - Acquire or release locks.

  - Read or write metadata stored in the Chubby namespace.

  - Register for change notifications on specific files or directories.

- Clients connect to the **Master Node** for metadata operations and synchronization.

4. File System-Like Namespace

- Chubby provides a hierarchical structure resembling a **file system**:

  - **Directories**: Used to group resources logically.

  - **Files**: Store small amounts of metadata or configuration data.

- Each file can act as a lock or store client-specific metadata.

5. Paxos Consensus Algorithm

- Used to maintain consistency among replicas.

- Ensures that updates to metadata or locks are replicated and agreed upon by a quorum (majority) of replicas.

- Guarantees that Chubby remains operational as long as a majority of replicas are functional.

6. Notifications and Watches

- Clients can set **watches** on files or directories in the Chubby namespace.

- When changes occur, the master sends **notifications** to registered clients.

7. Leases for Lock Management

- Locks are associated with leases, which have expiration times.

- This prevents stale locks if a client disconnects unexpectedly.

8. Security and Access Control

- Chubby uses strong authentication (e.g., Kerberos) to ensure secure access.

- Implements Access Control Lists (ACLs) to manage permissions for files and directories.

Workflow of Chubby

1. Lock Acquisition:

  - A client sends a lock request to the Chubby master.

  - The master checks lock availability and updates the lock state if granted.

  - Updates are replicated to other replicas using Paxos.

2. Metadata Access:

- Clients read/write metadata stored in the Chubby namespace through the master.

3. Failure Handling:

- If the master node fails, a new master is elected among the replicas using Paxos.

- Clients reconnect to the new master seamlessly.

This structure makes Chubby a highly reliable and scalable service for distributed synchronization and metadata storage in Google's infrastructure.

Chubby is intended to provide a coarse-grained locking service. It can store small files inside Chubby storage which provides a simple namespace as a file system tree. The files stored in Chubby are quite small compared to the huge files in GFS. The Chubby system can be quite reliable despite the failure of any member node. Figure shows the overall architecture of the Chubby system. Each Chubby cell has five replica nodes inside. Each replica node in the cell has the same file system namespace. Clients use the Chubby library to talk to the nodes in the cell. Client applications can perform various file operations on any node in the Chubby cell. Servers run the Paxos algorithm to make the whole file system reliable and consistent. Chubby has become Google's primary internal name service. GFS and Big Table use Chubby to elect a primary from redundant replicas.

Bigtable: Google's NoSQL System

Bigtable is a distributed, scalable, and high-performance NoSQL database developed by Google. It forms the backbone of many of Google's applications, such as Google Search, YouTube, Gmail, and Google Maps. Bigtable is specifically designed to handle large-scale workloads that involve vast amounts of structured and semi-structured data.

Key Features

1. Wide-Column Storage:

Data is stored in rows and columns, grouped into column families, making it ideal for semi-structured data like time-series or logs.

2. Scalability:

Bigtable scales horizontally to accommodate petabytes of data across thousands of machines, ensuring consistent performance as data grows.

3. Low Latency:

Optimized for real-time read and write operations with millisecond-level response times, it supports high-throughput applications.

4. Strong Consistency:

Unlike many NoSQL systems, Bigtable ensures strong consistency for read and write operations, making it reliable for critical applications.

5.Integration with Big Data Tools:

Bigtable seamlessly integrates with big data processing tools like Hadoop, Spark, and Google's analytics services like BigQuery.

Architecture

1. Row-Oriented Storage:

Data is indexed by unique row keys, and each row can have multiple column families. Each column family stores data in a compressed format for efficiency.

2. Column Families:

Data within a column family is stored together on disk, making read and write operations faster for related data.

3. Timestamps:

Each cell in Bigtable can hold multiple versions of data, indexed by timestamps. This feature is useful for time-series data.

4. Tablets:

Bigtable tables are divided into smaller units called "tablets," which are dynamically distributed across servers to balance load and ensure scalability.

5.Chubby Lock Service:

Google's distributed lock service, Chubby, is used for coordination and consistency across the distributed system.

How Bigtable Works

Data Storage:

Data is stored in rows and accessed using row keys. Each row contains column families, which in turn hold individual columns.

Data Distribution:

Tablets are distributed across multiple nodes for scalability and fault tolerance. If a node fails, the data is replicated and managed by the Bigtable master node.

Data Retrieval:

Efficient indexing using row keys ensures quick access to data. Applications can retrieve rows or specific subsets of data using filters.

Use Cases

1. Web Indexing:

Bigtable powers Google Search by storing and managing web indexing data.

2. Real-Time Analytics:

It is ideal for processing large-scale analytical workloads with low-latency requirements.

3. IoT Data Management:

Bigtable excels at handling time-series data generated by IoT devices.

4. Recommendation Systems:

It supports user personalization and recommendations in applications like YouTube and Gmail.

5. Financial Services:

Used for fraud detection and transaction analysis.

Advantages of Bigtable

Scalability: Can handle vast amounts of data efficiently.

High Availability: Data is distributed across servers to ensure fault tolerance.

Low Latency: Ideal for applications requiring real-time performance.

Data Versioning: Multiple versions of data are stored using timestamps.

Bigtable on Google Cloud

Google Cloud Bigtable is a managed version of Bigtable, providing developers with a hassle-free way to use Bigtable's capabilities on Google Cloud. It is particularly suited for:

Real-Time Applications: Low-latency data access for IoT and analytics.

Time-Series Databases: Storing logs or monitoring data.

Large-Scale Data Processing : Integration with Google Cloud services like Dataflow and Big Query.

Bigtable is a pioneering NoSQL system that addresses the challenges of managing massive datasets in distributed environments. Its robust architecture, scalability, and low-latency access make it an essential tool for applications requiring high performance and reliability.

4.4PROGRAMMING ON AMAZON AWS AND MICROSOFT AZURE:

Amazon (like Azure) offers a Relational Database Service (RDS) with a messaging interface. The Elastic MapReduce capability is equivalent to Hadoop running on the basic EC2 offering.

Amazon has NOSQL support in SimpleDB. However, Amazon does not directly support Big Table. Amazon offers the Simple Queue Service (SQS) and Simple Notification Service (SNS), which are the cloud implementations of services. Note that brokering systems run very efficiently in clouds and offer a striking model for controlling sensors and giving back-office support for a growing number of smartphones and tablets. We further note the auto-scaling and elastic load balancing services which support related capabilities. Auto-scaling enables you to automatically scale your Amazon EC2 capacity up or down according to conditions that you define. With autoscaling, you can ensure that the number of Amazon EC2 instances you're using scales up seamlessly during demand spikes to maintain performance, and scales down automatically during demand dulls to minimize cost.

Elastic load balancing automatically distributes incoming application traffic across multiple Amazon EC2 instances and allows you to avoid non-operating nodes and to equalize load on functioning images. Both auto-scaling and elastic load balancing are enabled by CloudWatch which monitors running instances. CloudWatch is a web service that provides monitoring for AWS cloud resources, starting with Amazon EC2. It provides customers with visibility into resource utilization, operational performance, and overall demand patterns including metrics such as CPU utilization, disk reads and writes, and network traffic.

Programming on Amazon EC2

Amazon was the first company to introduce VMs in application hosting. Customers can rent VMs instead of physical machines torun their own applications. By using VMs, customers can load any software of their choice. The elastic feature of such a service is that a customer can create, launch, and terminate server instances as needed, paying by the hour for active servers. Amazon provides several types of preinstalled VMs. Instances are often called Amazon Machine Images (AMIs) which are pre configured with operating systems based on Linux or Windows, and additional software. Table defines three types of AMI. Figure shows an execution environment. AMIs are the templates for instances, which are running VMs. The workflow to create a VM is Create an AMI → Create Key Pair → Configure Firewall → Launch This sequence is supported by public, private, and paid AMIs shown in Figure. The AMIs are formed from the virtualized compute, storage, and server resources shown at the bottom of Figure.

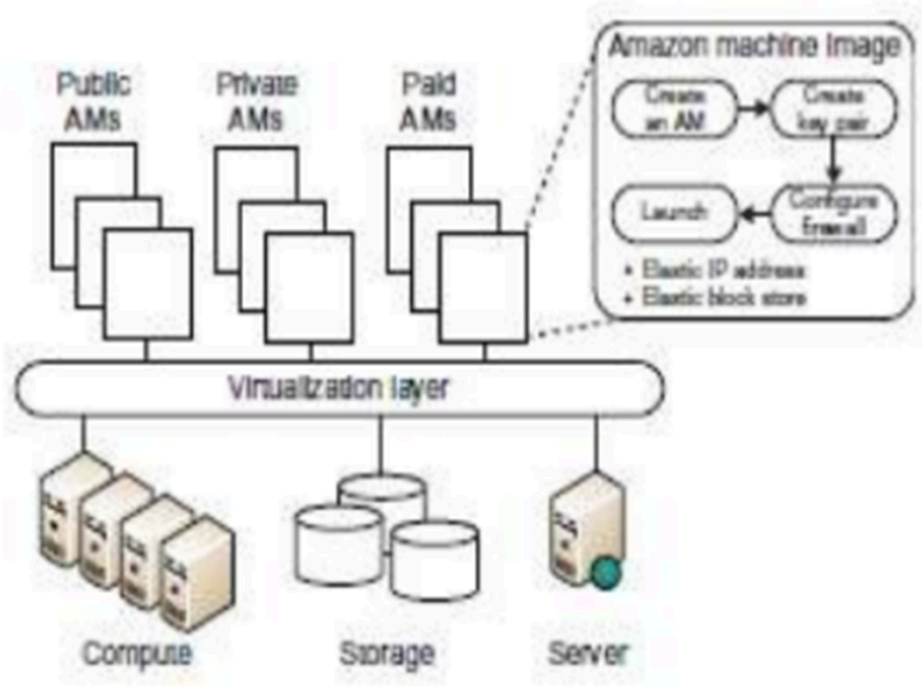
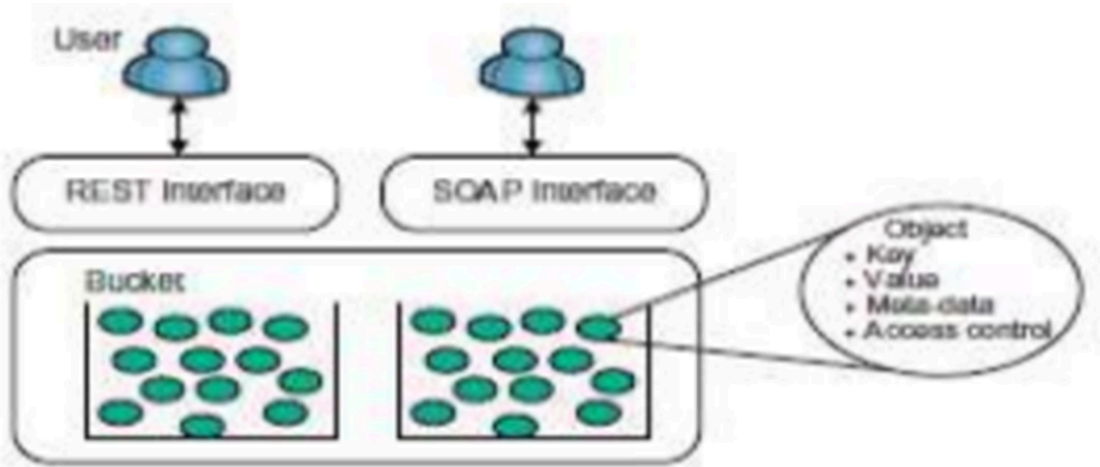


Fig: Types of AMI's



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, even if the receiver processes are not running. 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. From the programmer's perspective, the storage provided by S3 can be viewed as a very coarse-grained key-value pair.

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.

The key features of S3 are:

- Redundant through geographic dispersion.
- Designed to provide 99.999999999 percent durability and 99.99 percent 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 downloads protocol of HTTP. A BitTorrent protocol interface is provided to lower costs for high-scale distribution.
- $0.055(more than 5,000 TB) to 0.15 per GB per month storage (depending on total amount).
- First 1 GB per month input or output free and then $.08 to $0.15 per GB for transfers outside an S3 region.
- There is no data transfer charge for data transferred between Amazon EC2 and Amazon S3 within the same region or for data transferred between the Amazon EC2 Northern Virginia region and the Amazon S3 U.S. Standard region (as of October 6, 2010).

Amazon Elastic Block Store (EBS) and Simple DB

The Elastic Block Store (EBS) provides the volume block interface for saving and restoring the virtual images of EC2 instances. Traditional EC2 instances will be destroyed after use. 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. Note that S3 is <Storage as a Service= with a messaging interface. EBS is analogous to a distributed file system accessed by traditional OS disk access mechanisms. EBS allows you to create storage volumes from 1 GB to 1 TB that can be mounted as EC2 instances. 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. You can create a file system on top of Amazon EBS volumes, or use them in any other way you would use a block device (like a hard drive). Snapshots are provided so that the data can be saved incrementally. This can improve performance when saving and restoring data. In terms of pricing, Amazon provides a similar pay-per-use schema as EC2 and S3. The equivalent of EBS has been offered in open-source clouds such as Nimbus.

Amazon Simpledb Service

Simple DB 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. This is similar to a table in a relational database. However, it is possible to assign multiple values to a single cell in the table.

This is not permitted in a traditional relational database which wants to maintain data consistency. Many developers simply want to quickly store, access, and query the stored data. Simple DB removes the requirement to maintain database schemas with strong consistency. Simple DB, like Azure Table, could be called “Little Table”, as they are aimed at managing small amounts of information stored in a distributed table; one could say Big Table is aimed at basic big data, whereas Little Table is aimed at metadata.

Microsoft Azure Programming Support

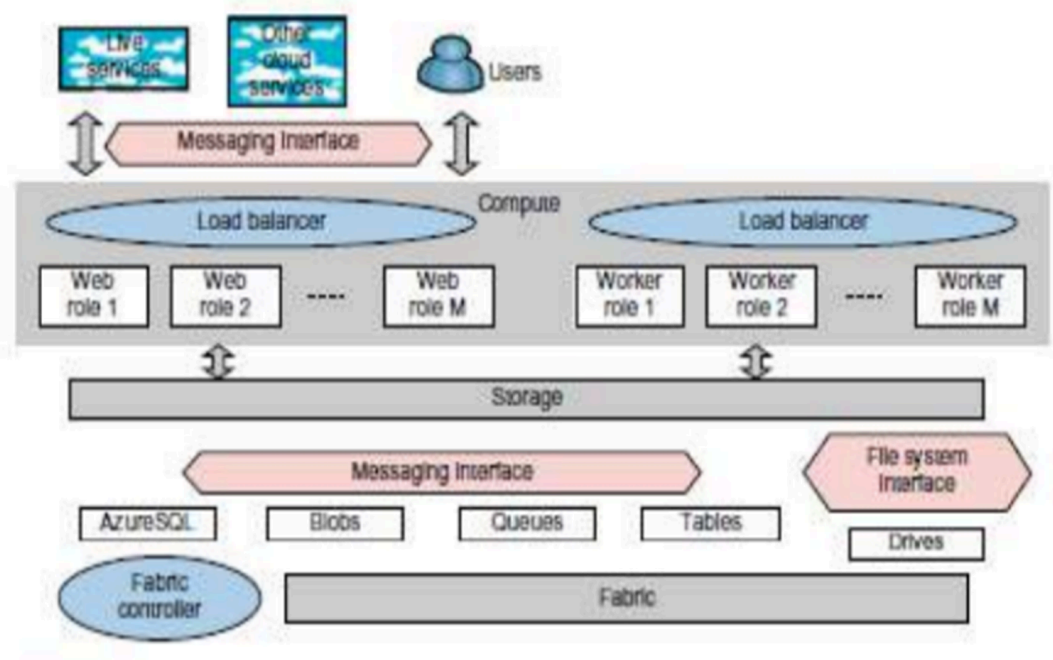
The key programming components, including the client development environment, SQL Azure, and the rich storage and programming subsystems, are shown in Figure. First, we have the underlying Azure fabric consisting of virtualized hardware together with a sophisticated control environment implementing dynamic assignment of resources and fault tolerance.

This implements domain name system (DNS) and monitoring capabilities. Automated service management allows service models to be defined by an XML template and multiple service copies to be instantiated on request. When the system is running, services are monitored and one can access event logs, trace/debug data, performance counters, IIS

web server logs, crash dumps, and other log files. This information can be saved in Azure storage. Note that there is no debugging capability for running cloud applications, but debugging is done from a trace. One can divide the basic features into storage and compute capabilities. The Azure application is linked to the Internet through a customized compute VM called a web role supporting basic Microsoft web hosting. Such configured VMs are often called appliances. The other important compute class is the worker role reflecting the importance in cloud computing of a pool of compute resources that are scheduled as needed.

The roles support HTTP(S) and TCP. Roles offer the following methods:

- The OnStart() method which is called by the Fabric on startup, and allows you to perform initialization tasks. It reports a Busy status to the load balancer until you return true.
- The OnStop() method which is called when the role is to be shut down and gives a graceful exit.
- The Run() method which contains the main logic. The Azure concept of roles is an interesting idea that we can expect to be expanded in terms of role types and use in other cloud environments.



SQL Azure

Azure offers a very rich set of storage capabilities, as shown in Figure 6.25. The SQL Azure service offers SQL Server as a service. All the storage modalities are accessed with REST interfaces except for the recently introduced Drives that are analogous to Amazon EBS, and offer a file system interface as a durable NTFS volume backed by blob storage. The REST interfaces are automatically associated with URLs and all storage is replicated three times for fault tolerance and is guaranteed to be consistent in access. The basic storage system is built from blobs which are analogous to S3 for Amazon. Blobs are arranged as a three-level hierarchy: Account → Containers → Page or Block Blobs. Containers are analogous to directories in traditional file systems with the account acting as the root. The block blob is used for streaming data and each such blob is made up as a sequence of blocks of up to 4 MB each, while each block has a 64-byte ID. Block blobs can be up to 200 GB in size. Page blobs are for random read/write access and consist of an array of pages with a maximum blob size of 1 TB. One can associate metadata with blobs as <name, value> pairs with up to 8 KB per blob.

Azure Tables

The Azure Table and Queue storage modes are aimed at much smaller data volumes. Queues provide reliable message delivery and are naturally used to support work spooling between web and worker roles. Queues consist of an unlimited number of messages which can be retrieved and processed at least once with an 8 KB limit on message size. Azures supports PUT, GET, and DELETE message operations as well as CREATE and DELETE for queues. Each account can have any number of Azure tables which consist of rows called entities and columns called properties. There is no limit to the number of entities in a table and the technology is designed to scale well to a large number of entities stored on distributed computers. All entities can have up to 255 general properties which are <name, type, value> triples. Two extra properties, Partition Key and Row Key, must be defined for each entity, but otherwise, there are no constraints on the names of properties .

4.5 EMERGING CLOUD SOFTWARE ENVIRONMENTS

Open Source Eucalyptus and Nimbus

Eucalyptus is a product from Eucalyptus Systems that was developed out of a research project at the University of California, Santa Barbara. Eucalyptus was initially aimed at bringing the cloud computing paradigm to academic supercomputers and clusters. Eucalyptus provides an AWS-compliant EC2-based web service interface for interacting with the cloud service. Additionally, Eucalyptus provides services, such as the AWS-compliant Walrus, and a user interface for managing users and images.

Eucalyptus Architecture

The Eucalyptus system is an open software environment. The architecture was presented in a Eucalyptus white paper. Figure 6.26 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. This allows users to create specialty virtual appliances and deploy them within Eucalyptus with ease.

Nimbus

Nimbus is a set of open source tools that together provide an IaaS cloud computing solution. Figure 6.27 shows the architecture of Nimbus, which allows a client to lease remote resources by deploying VMs on those resources and configuring them to represent the environment desired by the user. To this end, Nimbus provides a special web interface known as Nimbus Web. Its aim is to provide administrative and user functions in a friendly interface. Nimbus Web is centered on a Python Django web application that is intended to be deployable completely separate from the Nimbus service. As shown in Figure 6.27, a storage cloud implementation called Cumulus has been tightly integrated with the other central services, although it can also be used stand-alone. Cumulus is compatible with the Amazon S3 REST API , but extends its capabilities by including features such as quota management. On the other hand, the Nimbus cloud client uses the Java Jets3t library to interact with Cumulus. Nimbus supports two resource management strategies. The first is the default <resource pool= mode. In this mode, the service has direct control of a pool of VM manager nodes and it assumes it can start VMs. The other supported mode is called <pilot.= Here, the service makes requests to a cluster's Local Resource Management System (LRMS) to get a VM manager available to deploy VMs. Nimbus also provides an implementation of Amazon's EC2 interface that allows users to use clients developed for the real EC2 system against Nimbus-based clouds.

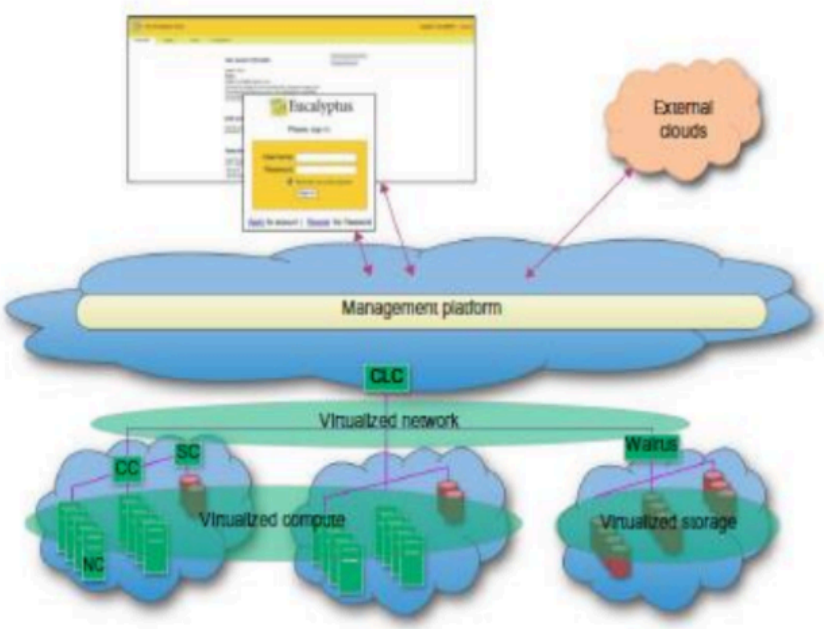


FIGURE 6.26
The Eucalyptus architecture for VM image management.

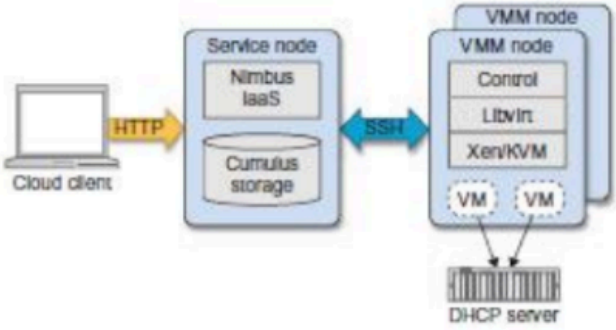


FIGURE 6.27
Nimbus cloud infrastructure.

OpenNebula, Sector/Sphere, and OpenStack

OpenNebula is an open source toolkit which allows users to transform existing infrastructure into an IaaS cloud with cloud-like interfaces. Figure 6.28 shows the OpenNebula architecture and its main components. The architecture of OpenNebula has been designed to be flexible and modular to allow integration with different storage and network infrastructure configurations, and hypervisor technolog-ies. 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, such as VM disk image deployment or on-the-fly software environment creation. 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. However, it is also possible to develop more complex scheduling policies, through a lease model and advance reservations. 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. Therefore, OpenNebula is not tied to any specific environment and can provide a uniform management layer regardless of the virtualization platform. 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 is able to adapt to organizations with changing resource needs, including addition or failure of physical resources. Some essential features to support changing environments are live migration and VM snapshots. Furthermore, when the local resources are insufficient, OpenNebula can support a hybrid cloud model by using cloud drivers to interface with external clouds. This lets organizations supplement their local infrastructure with computing capacity from a public cloud to meet peak demands, or implement HA strategies. OpenNebula currently includes an EC2 driver, which can submit requests to Amazon EC2 and Eucalyptus , as well as an ElasticHosts driver. Regarding storage, an Image Repository allows users to easily specify disk images from a catalog without worrying about low-level disk configuration attributes or block device mapping. Also, image access control is applied to the images registered in the repository, hence simplifying multiuser environments and image sharing. Nevertheless, users can also set up their own images.

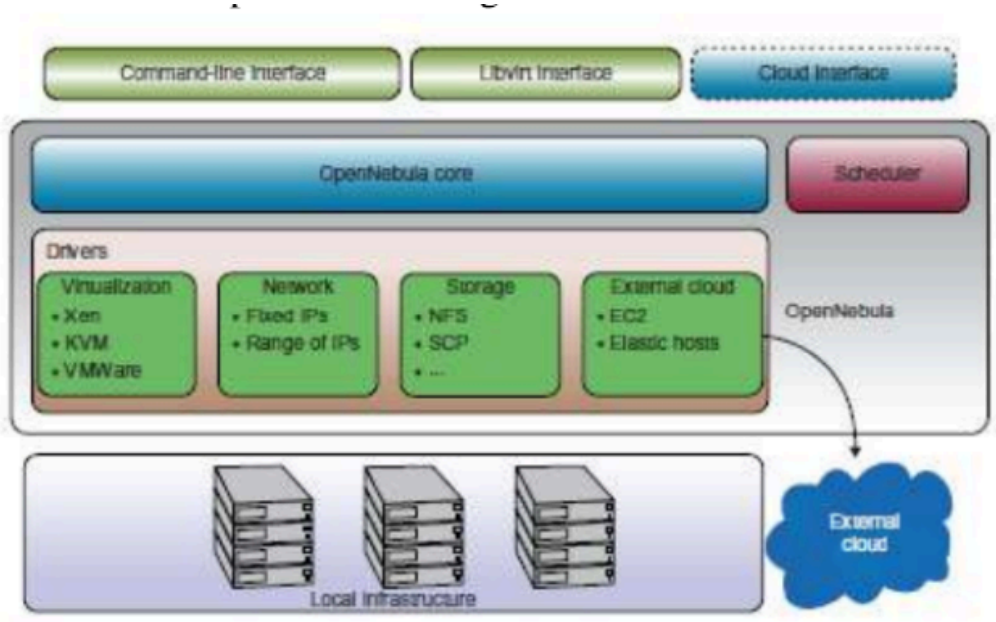


FIGURE 6.28

OpenNebula architecture and its main components.

Sector/Sphere is a software platform that supports very large distributed data storage and simplified distributed data processing over large clusters of commodity computers, either within a data center or across multiple data centers. The system consists of the Sector distributed file system and the Sphere parallel data processing framework. Sector is a distributed file system (DFS) that can be deployed over a wide area and allows users to manage large data sets from any location with a high speed network connection. The fault tolerance is implemented by replicating data in the file system and managing the replicas. Since Sector is aware of the network topology when it places replicas, it also provides better reliability, availability, and access throughout. The communication is performed using User Datagram Protocol (UDP) for message passing and user-defined type (UDT) for data transfer. Obviously, UDP is faster than TCP for message passing because it does not require connection setup, but it could become a problem if Sector is used over the Internet. Meanwhile, UDT is a reliable UDP-based application-level data transport protocol which has been specifically designed to enable high-speed data transfer over wide area high-speed networks. Finally, the Sector client provides a programming API, tools, and a FUSE user space file system module.

On the other hand, Sphere is a parallel data processing engine designed to work with data managed by Sector. This coupling allows the system to make accurate decisions about job scheduling and data location. Sphere provides a programming framework that developers can use to process data stored in Sector. Thus, it allows UDFs to run on all input data segments in parallel. Such data segments are processed at their storage locations whenever possible (data locality). Failed data segments may be restarted on other nodes to achieve fault tolerance. In a Sphere application, both inputs and outputs are Sector files. Multiple Sphere processing segments can be combined to support more complicated applications, with inputs/outputs exchanged/shared via the Sector file system. The Sector/Sphere platform is supported by the architecture shown in Figure 6.29, which is composed of four components. The first component is the security server, which is responsible for authenticating master servers, slave nodes, and users. We also have the master servers that can be considered the infrastructure core. The master server maintains file system metadata, schedules jobs, and responds to users' requests. Sector supports multiple active masters that can join and leave at runtime and can manage the requests. Another component is the slave nodes, where data is stored and processed. The slave nodes can be located within a single data center or across multiple data centers with high-speed network connections. The last component is the client component. This provides tools and programming APIs for accessing and processing Sector data. Finally, it is worthy to mention that as part of this platform, a new component has been developed. It is called Space and it consists of a framework to support column-based distributed data tables. Therefore, tables are stored by columns

and are segmented onto multiple slave nodes. Tables are independent and no relationships between them are supported. A reduced set of SQL operations is supported, including, but not limited to, table creation and modification, key-value updates and lookups, and select UDF operations.

OpenStack

OpenStack was introduced by Rackspace and NASA in July 2010. The project is building an open source community spanning technologists, developers, researchers, and industry to share resources and technologies with the goal of creating a massively scalable and secure cloud infrastructure. In the tradition of other open source projects, the software is open source and limited to just open source APIs such as Amazon. Currently, 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.= Recently, an image repository was prototyped. The image repository contains an image registration and discovery service and an image delivery service. Together they deliver images to the compute service while obtaining them from the storage service. This development gives an indication that the project is striving to integrate more services into its portfolio.

OpenStack Compute

As part of its computing support efforts, OpenStack 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. This is very similar to established concepts from parallel computing, such as <futures,= which have been used in the grid community by projects such as the CoG Kit. 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 it implemented in Python while utilizing a number of externally supported libraries and components. This includes boto, an Amazon API provided in Python, and Tornado, a fast HTTP server used to implement the S3 capabilities in OpenStack. Figure 6.30 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, as well as storage workers through a queue. Additionally, Nova integrates networking components to manage private networks, public IP addressing, virtual private network (VPN) connectivity, and firewall rules. 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. Separate rings for accounts, containers, and objects exist. 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. According to the documentation, <the Object Server is a very simple blob storage server that can store, retrieve and delete objects stored on local devices.= 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. To list objects, a container server can be utilized. Listing of containers is handled by the

account server. The first release of OpenStack <Austin= Compute and Object Storage was October 22, 2010. This system has a strong developer community.

Manjrasoft Aneka Cloud and Appliances

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. System administrators can leverage a collection of tools to monitor and control the deployed infrastructure. It can be deployed on a public cloud such as Amazon EC2 accessible through the Internet to its subscribers, or a private cloud constituted by a set of nodes with restricted access as shown in Figure 6.31.

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 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. Whenever users set QoS parameters such as deadlines, and if the enterprise resources are insufficient to meet the deadline, Aneka supports

dynamic leasing of extra capabilities from public clouds such as EC2 to complete the task within the deadline (see Figure 6.32).