Module:5 Cloud Programming and Software Environments: Features of Cloud and Grid Platforms, Parallel and Distributed Computing Paradigms, Programming Support for Google App Engine, Programming on Amazon AWS and Microsoft, Emerging Cloud Software Environments.

6.1 Features of Cloud and Grid Platforms

Cloud Computing Features

- **On-demand self-service** – Users can access resources (like storage or computing power) as needed.
- **Scalability** – Resources can be increased or decreased easily.
- **Resource pooling** – Multiple users share resources securely.
- **Broad network access** – Services are available over the internet on any device.
- **Pay-per-use model** – You only pay for what you use.
- **Automatic updates** – Cloud services are updated automatically by the provider.
- **High availability** – Cloud services offer strong uptime and data backup.

Grid Computing Features

- **Resource sharing** – Connects many computers to share resources like CPU or storage.
- **Distributed computing** – Tasks are divided and processed across multiple systems.
- **High performance** – Suitable for complex computations and scientific tasks.
- **Scalability** – More computers can be added to improve performance.
- **Heterogeneous resources** – Works with different hardware and operating systems.
- **Decentralized** – No single point of control, resources are managed locally.
- **Collaboration** – Often used in academic or research collaborations.

Cloud Capabilities and Platform Features

Capability	Description
Physical or virtual computing platform	The cloud environment includes physical and virtual platforms, with virtual platforms offering isolated environments for diverse applications and users.
Massive data storage service, distributed filesystem	Cloud data storage services and distributed file systems provide large disk capacity and interfaces for storing and retrieving data, similar to local file systems.
Massive database storage service	Distributed file systems and cloud database storage services provide structured and scalable data storage, similar to traditional DBMS, enabling developers to store data efficiently.
Massive data processing method and programming model	Cloud infrastructure enables programmers to utilize thousands of computing nodes efficiently without managing complex issues like network failures or scalability, allowing seamless access to computing resources.
Workflow and data query language support	Cloud computing provides workflow and data query languages that abstract the infrastructure, enabling better application logic, similar to how SQL simplifies database management.
Programming interface and service deployment	Cloud applications use web interfaces and APIs (e.g., J2EE, PHP, ASP, Rails) to provide functionality, while Ajax enhances user experience, and cloud providers offer programming interfaces to access massive storage efficiently
Runtime support	Cloud runtime support ensures transparent execution for users and applications, offering distributed monitoring, task scheduling, and locking services, which are essential for running cloud applications efficiently.
Support services	Clouds provide data and computing services, including rich data services and parallel execution models like MapReduce for efficient processing.

Traditional Features Common to Grids and Clouds

In this section, we concentrate on features related to workflow, data transport, security, and availability concerns that are common to today's computing grids and clouds

- **Workflow**-Workflow systems like Pegasus, Taverna, and Kepler have emerged but lack universal adoption. Commercial options include Pipeline Pilot, AVS, and LIMS. Microsoft's **Trident**, built on **Windows Workflow Foundation**, runs on **Azure or Windows** and integrates **cloud and non-cloud services** via Linux proxy services.
- **Data Transport**-Data transport in commercial clouds is costly, but **TeraGrid integration** could improve bandwidth. **Cloud data structures** support parallel algorithms, though **HTTP-based transfers** are currently used
- **Security, Privacy and Availability**-Cloud programming ensures **security, privacy, and availability** through **virtual clustering**, **secure APIs**, **HTTPS/SSL protocols**, **fine-grained access control**, **shared data protection**, **VM live migration**, and a **reputation system** for trusted access.

Below are the traditional features common to Grids and Clouds

- **Resource Sharing** – Both allow sharing of computing, storage, and network resources.
- **Scalability** – Can grow or shrink in resources as demand changes.
- **Virtualization** – Use of virtual machines to manage and deploy applications flexibly.
- **High Performance Computing (HPC)** – Support for intensive computing tasks and parallel processing.
- **Service-Oriented Architecture (SOA)** – Both use web services for accessing and managing resources.
- **Distributed Computing** – Tasks and data are spread across multiple systems.
- **Fault Tolerance** – Mechanisms to recover from system or hardware failures.
- **Multi-tenancy** – Multiple users or organizations can use the same infrastructure **Security and Authentication** securely.
- – Ensure secure access to shared resources.
- **Job Scheduling** – Manage and queue multiple jobs across available resources.

Data Features and Databases

Program Library

- VM image libraries manage cloud images.
- Used in both academic and commercial clouds.
- Cloud environments support IaaS for easy deployment/configuration.

Blobs and Drives

- Azure: uses blobs, organized via containers.
- Amazon: uses S3 for object storage and EBS for block storage.
- Azure Drives and Amazon EBS can attach directly to VMs.
- Cloud storage is fault-tolerant; similar in idea to Lustre on TeraGrid.
- Simple Cloud File Storage API could help unify interfaces.

DPFS (Data Processing File Systems)

- Includes GFS (Google), HDFS (Hadoop), Cosmos (Dryad).
- Optimized for compute-data affinity in data processing.
- DPFS is application-centric; blobs/drives are repository-centric.
- Azure lacks strong compute-data affinity support; Azure Affinity Groups are a start.

SQL and Relational Databases

- Both Amazon and Azure support relational databases.
- Academic clouds can support similar features unless at huge scale.
- Example: FutureGrid uses Oracle, SAS, and Hadoop for medical data analysis.
- Cloud DBs run as independent services (SQL-as-a-Service).
- Simplifies management, reduces VM image complexity.

Table and NoSQL Databases

- NoSQL = schema-free, scalable, distributed data storage.
- Examples:
    - Google: BigTable
    - Amazon: SimpleDB
    - Azure: Azure Table
- Useful for scientific data (e.g., VOTable in astronomy).
- Tools: HBase (BigTable-style), CouchDB (document store), M/DB (SimpleDB-like).
- Simple Cloud APIs aim to standardize across platforms.

Queuing Services

- Azure and Amazon provide robust, REST-based queue services.
- Messages are small (<8KB), with "deliver at least once" semantics.
- Academic alternatives: ActiveMQ, NaradaBrokering.
- Used to connect application components.

Programming and Runtime Support

Programming and runtime support are desired to facilitate parallel programming and provide run time support of important functions in today's grids and clouds.

- **Worker & Web Roles:** Azure's worker roles allow automatic process scheduling, while web roles support portals and applications like Google App Engine (GAE).
- **MapReduce:** A framework for parallel data processing with advantages like dynamic execution and fault tolerance. Popular implementations include Hadoop (Amazon) and Dryad (expected on Azure).

  MapReduce is a programming model used to **process large amounts of data** across many computers (nodes) in a distributed system.

- It breaks the task into two main steps:
- **Map:**
  - Takes input data and **breaks it into key-value pairs**.
  - Each piece is processed independently in parallel.
- **Reduce:**
  - Takes all the output from the Map step, **groups by key**, and combines values to produce final results.

- **Cloud Programming Models:** Includes models like Google App Engine (GAE) and Aneka, supporting cloud, HPC, and clusters. Iterative MapReduce offers portability across platforms.
- **Software as a Service (SaaS):** Users can package applications without needing specialized support. Cloud services like MapReduce, BigTable, EC2, S3, and Hadoop ensure scalability, security, and availability.

6.2 PARALLEL AND DISTRIBUTED PROGRAMMING PARADIGMS

- parallel and distributed program 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.

Ex: Cluster or a Network

Running a parallel program on a distributed computing system (parallel and distributed programming) has several advantages for both users and distributed computing systems.

From User Perspective the advantages are

- decreases application response time

From distributed computing Perspective the advantages are

- increases throughput and resource utilization

Running a parallel program on a distributed computing system, however, could be a very complicated process.

Key Issues in Running Parallel/Distributed Programs:

Partitioning

- **Computation Partitioning**
    - ✓ Splits a job/program into smaller concurrent tasks.
    - ✓ Requires identifying parts of the program that can run in parallel.
    - ✓ Tasks may process different data or the same data copy.
- **Data Partitioning**
    - ✓ Splits input or intermediate data into smaller chunks.
    - ✓ Chunks can be processed in parallel by different program parts or program copies.

Mapping

- ✓ Assigns smaller tasks or data pieces to computing resources.
- ✓ Ensures efficient use of workers and concurrency.
- ✓ Typically handled by system resource allocators.

Synchronization

- ➢ Coordinates workers to prevent **race conditions** and manage **data dependencies**.
- ➢ Ensures correct sequencing when tasks share resources or depend on others' output.

Communication

- ➢ Involves transferring intermediate data between workers.

Scheduling

Single Job/Program:

- ➢ Selects the order of task/data assignment when workers are fewer than tasks.

Multiple Jobs/Programs:

- Chooses which jobs/programs run based on system resource availability.

Scheduler decides the order; **resource allocator** performs actual assignment.

Motivation for Programming Paradigms

- Parallel and distributed programming models help simplify the complexities of handling data flow in large-scale systems. These models provide an abstraction layer, allowing programmers to focus on their logic instead of implementation details.

Key advantages of these models include:

- Boosting programmer productivity by reducing the need for specialized knowledge.
- Accelerating time to market by streamlining development.
- Optimizing resource utilization for efficient system performance.
- Enhancing system throughput to handle large-scale workloads.
- Supporting higher levels of abstraction for ease of programming.

Popular models like MapReduce, Hadoop, and Dryad were initially designed for information retrieval but have proven useful for various applications. Their loose coupling makes them suitable for virtual machine (VM) environments, improving fault tolerance and scalability over traditional models.

MapReduce, Twister, and Iterative MapReduce

- ✓ MapReduce is a software framework for parallel and distributed computing on large datasets.
- ✓ It abstracts the data flow in distributed systems to simplify programming.
- ✓ Provides two main user-defined functions: Map and Reduce.
- ✓ Users override these functions to process and manage data flow in their programs.
- ✓ The Map function processes input data and produces intermediate key-value pairs.
- ✓ The Reduce function aggregates intermediate data to produce final output.
- ✓ The framework handles data distribution, parallel execution, and fault tolerance behind the scenes.

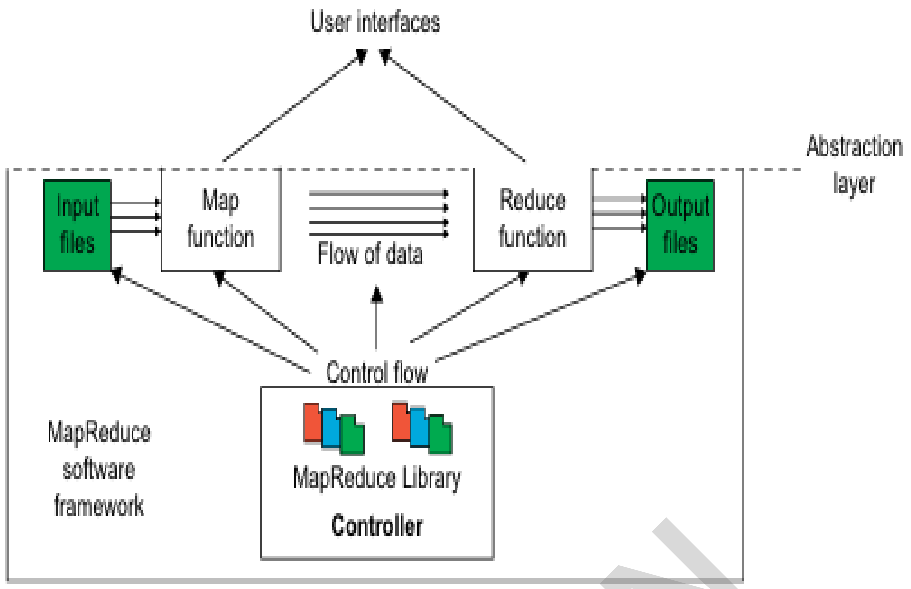


FIGURE 6.1

MapReduce framework: Input data flows through the Map and Reduce functions to generate the output result under the control flow using MapReduce software library. Special user interfaces are used to access the Map and Reduce resources.

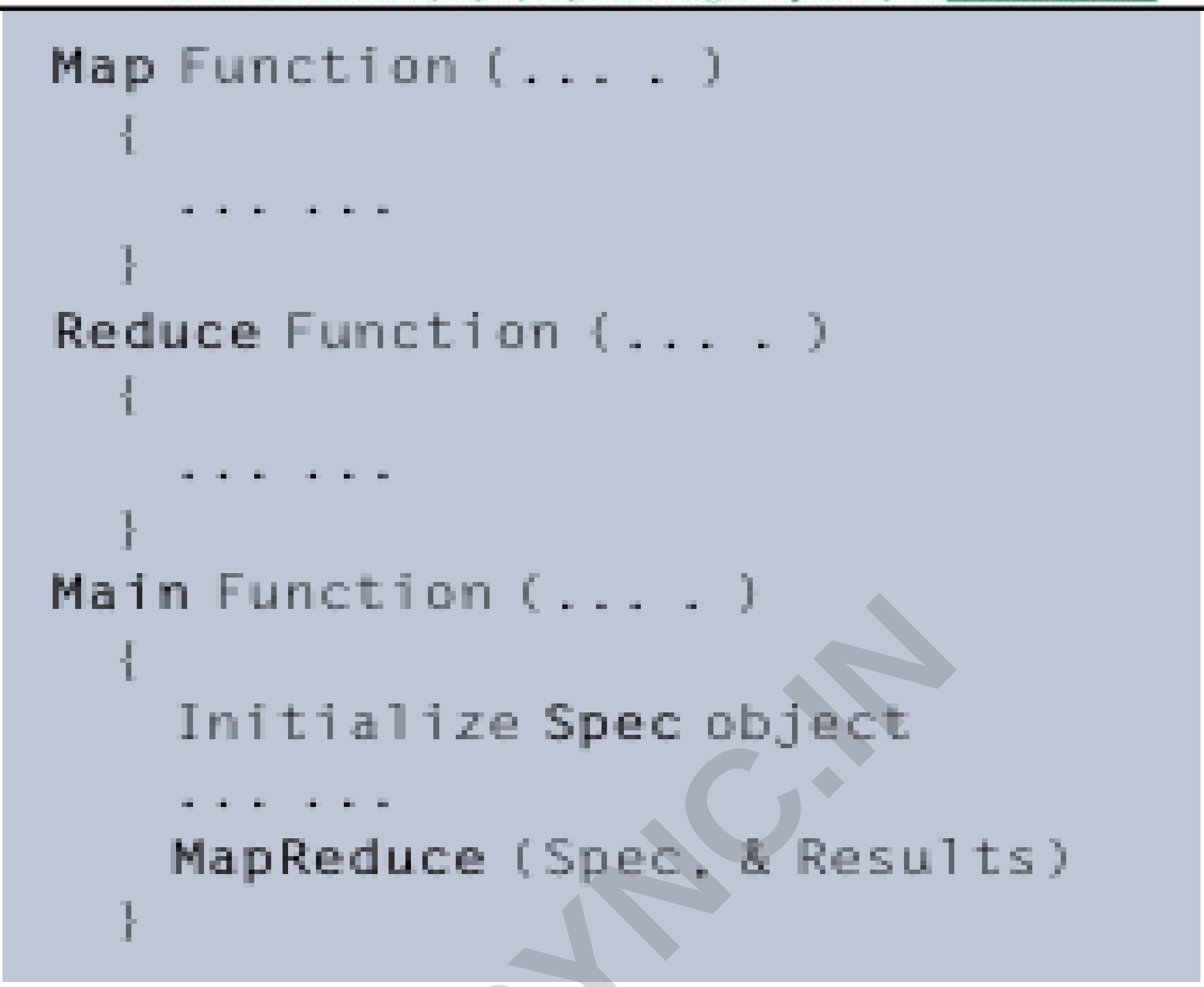
Formal Definition of MapReduce

- ✓ **MapReduce framework** provides an **abstraction layer** for data and control flow in distributed computing.
- ✓ It **hides complex implementation details** like data partitioning, mapping, synchronization, communication, and scheduling.
- ✓ The framework exposes **two user-defined interfaces**: **Map** and **Reduce** functions.
- ✓ Users call the built-in **MapReduce(Spec, &Results)** function to execute their program.
- ✓ A **Spec object** must be initialized and populated with:
- ✓ Names of input and output files.
- ✓ Names of user-defined Map and Reduce functions.
- ✓ Optional tuning parameters.

The user program includes three main components:

- ✓ **Map Function** – defines how input data is processed into intermediate key-value pairs.
- ✓ **Reduce Function** – defines how intermediate data is aggregated into final results.

Main Function – sets up the Spec object and invokes the MapReduce() function



```
Map Function (.... )
  {
     ... ...
  }
Reduce Function (.... )
  {
     ... ...
  }
Main Function (.... )
  {
     Initialize Spec object
     ... ...
     MapReduce (Spec, & Results)
  }
```

MapReduce Logical Data Flow

Input to Map Function:

Format: **(key, value)** pair.

Example: **(line offset, line content)** from an input file.

Output of Map Function:

Emits **intermediate (key, value)** pairs.

One input may produce zero or more intermediate pairs.

Example: For word count, output could be **(word, 1)** for each word.

Processing Intermediate Data:

Values with the same key are **grouped** into a structure like **(key, [values])**.

Input to Reduce Function:

Format: **(key, [set of values])**.

The Reduce function processes each group to compute a result.

Output of Reduce Function:

Produces final **(key, value)** pairs as output.

Example: In word count, **(people, 2)** indicates the word "people" appeared twice

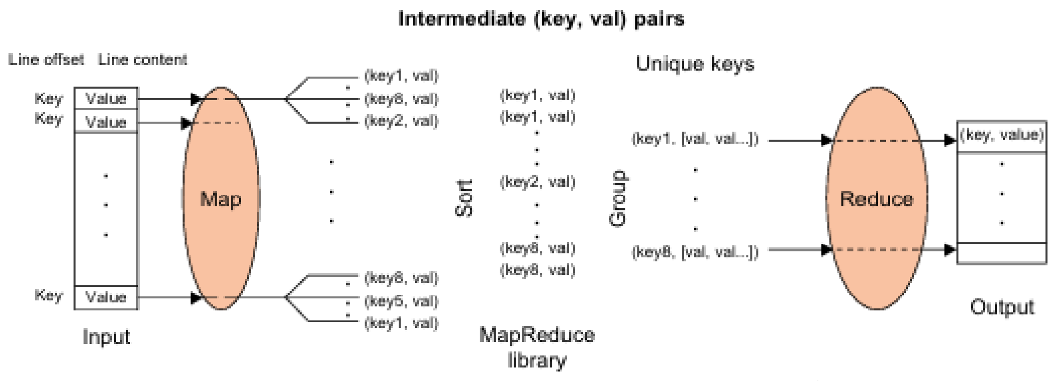
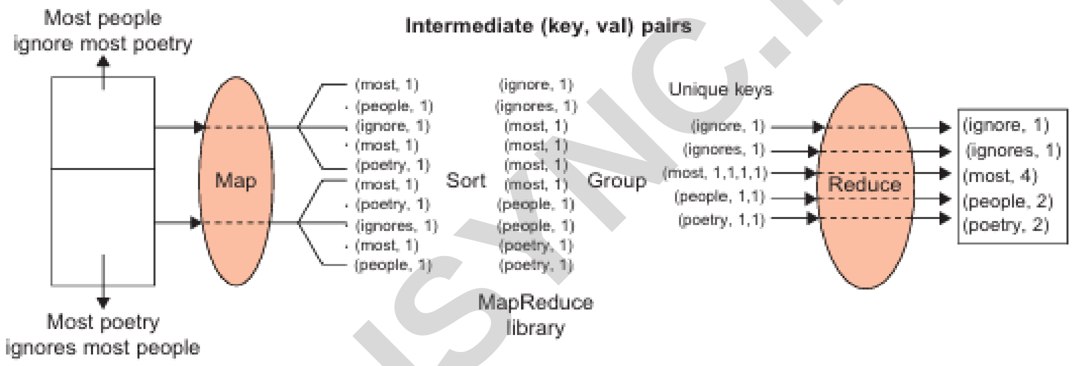


FIGURE 6.2

MapReduce logical data flow in 5 processing stages over successive (key, value) pairs.



Formal Notation of MapReduce Data Flow

The Map function is applied in parallel to every input (key, value) pair, and produces new set of intermediate (key, value) pairs as follows

$$(key_1, val_1) \xrightarrow{\text{Map Function}} \text{List}\,(key_2, val_2)$$

Then the MapReduce library collects all the produced intermediate (key, value) pairs from all input (key, value) pairs, and sorts them based on the "key" part. It then groups the values of all occur rences of the same key. Finally, the Reduce function is applied in parallel to each group producing the collection of values as output as follows

$$(key_2, \text{List}\,(val_2)) \xrightarrow{\text{Reduce Function}} \text{List}\,(val_2)$$

Strategy to Solve MapReduce Problems

MapReduce Basics: Finding unique keys is crucial for solving MapReduce problems, as intermediate data is grouped by key.

- Problem Examples & Solutions:

1. Word Frequency Counting: Each word is the unique key, and its occurrences form the value.
2. Word Size Grouping: Each word is the unique key, with its length as the value.
3. Anagram Counting: The key is the alphabetically sorted sequence of letters, and the value is the number of occurrences.

Example: Word Count Problem

Input lines:

“most people ignore most poetry”

“most poetry ignores most people”

Map output (key = word, value = 1):

Example: **(most, 1)**, **(people, 1)**, **(ignore, 1)**

After Sort and Group:

Example: **(most, [1, 1, 1, 1])**, **(people, [1, 1])**

Reduce output:

Example: **(most, 4)**, **(people, 2)**

MapReduce Actual Data and Control Flow

- Data Partitioning: Input data is split into smaller chunks, matching the number of map tasks.
- Computation Partitioning: Programs are written using Map and Reduce functions, and copies are distributed across computing nodes.
- Master-Worker Model: One instance acts as the master, assigning tasks to worker nodes.
- Map Phase: Each map worker processes its input split, producing intermediate key-value pairs.
- Combiner Function (Optional): Reduces communication costs by locally merging data before sending it to reduce tasks.
- Partitioning Function: Ensures that identical keys are processed together by grouping data accordingly.

- Synchronization: Reduce tasks begin only after all map tasks finish.
- Communication & Data Transfer: Reduce workers retrieve assigned data using remote procedure calls.
- Sorting & Grouping: Reduce workers arrange and group data for efficient processing.
- Reduce Function: Processes grouped data to generate final results.

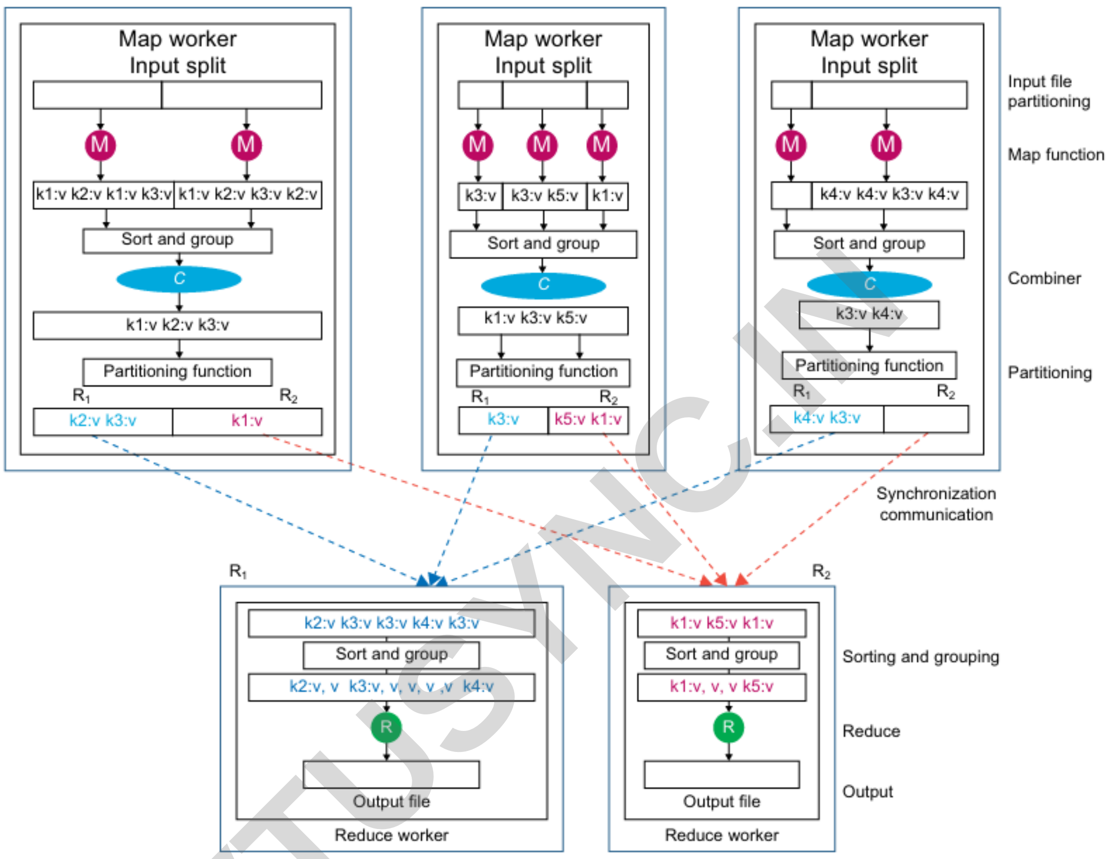


FIGURE 6.5

Data flow implementation of many functions in the Map workers and in the Reduce workers through multiple sequences of partitioning, combining, synchronization and communication, sorting and grouping, and reduce operations.

Compute-Data Affinity

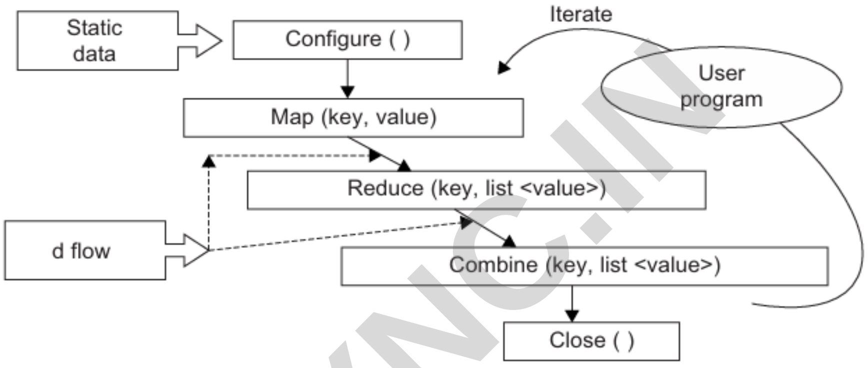
- **Origin & Implementation:** Google first proposed and implemented MapReduce in C, integrating it with Google File System (GFS).
- **GFS & Data Storage:** GFS stores files as fixed-size blocks (chunks) distributed across cluster nodes.
- **MapReduce Process:** The framework splits input data into blocks, performing Map tasks in parallel.
- **Optimized Data Processing:** Instead of moving large data sets, MapReduce sends computation to the data location, improving efficiency.

- **Block Size:** Both GFS and MapReduce use a default block size of 64 MB.

Twister and Iterative MapReduce

It is important to understand the performance of different runtimes and, in particular, to compare MPI and MapReduce

- MapReduce reads and writes via files, whereas MPI transfers information directly between nodes over the network
- MPI does not transfer all data from node to node, but just the amount needed to data flow.



(a) Twister for iterative MapReduce programming

- MPI and MapReduce differ significantly in performance due to communication overhead and load imbalance.
- MPI minimizes data transfer by sending only essential updates (δ flow), while MapReduce writes and reads full data via files, leading to higher overhead.
- To improve efficiency, streaming data between steps and using long-running threads for δ flow can enhance performance, though at the cost of fault tolerance.
- Twister, a modified MapReduce paradigm, optimizes iterative computations and performs better than traditional MapReduce, particularly in tasks like K-means clustering.

Hadoop Library from Apache

Hadoop is an open source implementation of MapReduce coded and released in Java (rather than C) by Apache. The Hadoop implementation of MapReduce uses the Hadoop Distributed File System (HDFS) as its underlying layer rather than GFS. The Hadoop core is divided into two fundamental layers: the MapReduce engine and HDFS. The MapReduce engine is the computation engine run ning on top of HDFS as its data storage manager. The following two sections cover the details of these two fundamental layers.

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

HDFS Architecture:

HDFS follows a **master/slave architecture**:

- ✓ **NameNode** (master): Manages metadata and file namespace.
- ✓ **DataNodes** (slaves): Store and retrieve file blocks.

Files are split into **fixed-size blocks** (e.g., 64MB) and distributed across DataNodes.

The **NameNode** keeps track of block locations on DataNodes.

HDFS Features:

- ❑ Designed for high performance, scalability, and fault tolerance.
- ❑ Not a general-purpose file system; it lacks some features like built-in security.
- ❑ HDFS Fault Tolerance
- ❑ Block Replication: Each file block is stored in multiple copies (default: 3 replicas).
- ❑ Replica Placement:
  - ✓ 1st copy on the original node.
  - ✓ 2nd copy on another node in the same rack.
  - ✓ 3rd copy on a node in a different rack.
- ❑ Heartbeats & Block Reports:
  - ✓ DataNodes send heartbeats to indicate they are alive.
  - ✓ Block reports list all blocks stored on a DataNode.

HDFS Operation:

Reading a File:

- ✓ User requests file info from the NameNode.
- ✓ NameNode returns addresses of DataNodes with block replicas.
- ✓ User reads each block from the nearest DataNode.

Writing a File:

- ✓ User sends a "create" request to the NameNode.
- ✓ Data is written in blocks to a data queue.
- ✓ A data streamer sends blocks to the first DataNode.
- ✓ The first DataNode forwards the block to the next DataNode(s) for replication.
- ✓ Process repeats until all blocks are stored and replicated.

Architecture of MapReduce in Hadoop

The topmost layer of Hadoop is the MapReduce engine that manages the data flow and control flow of MapReduce jobs over distributed computing systems.

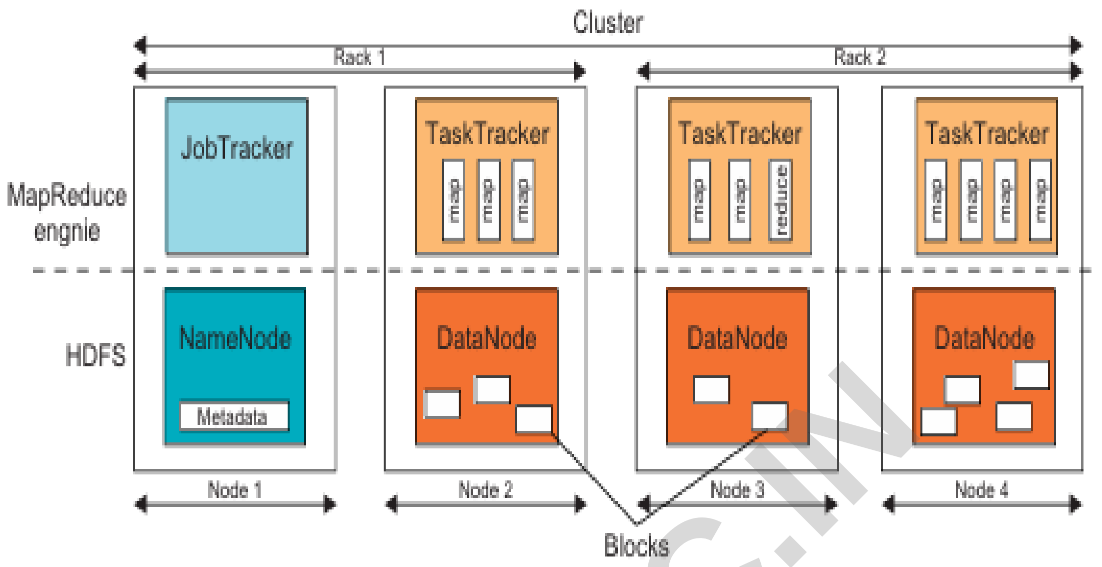


FIGURE 6.11

HDFS and MapReduce architecture in Hadoop where boxes with different shadings refer to different functional nodes applied to different blocks of data.

Figure 6.11 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 manages multiple **execution slots** based on CPU threads (**M * N slots**).
- Each data block is processed by **one map task**, ensuring a direct one-to-one mapping between map tasks and data blocks.

Running a Job in Hadoop

- Job Execution Components: A user node, a JobTracker, and multiple TaskTrackers coordinate a MapReduce job.
- Job Submission: The user node requests a job ID, prepares input file splits, and submits the job to the JobTracker.

- Task Assignment: The JobTracker assigns map tasks based on data locality and reduce tasks without locality constraints.
- Task Execution: The TaskTracker runs tasks by copying the job's JAR file and executing it in a Java Virtual Machine (JVM).
- Task Monitoring: Heartbeat messages from TaskTrackers inform the JobTracker about their status and readiness for new tasks

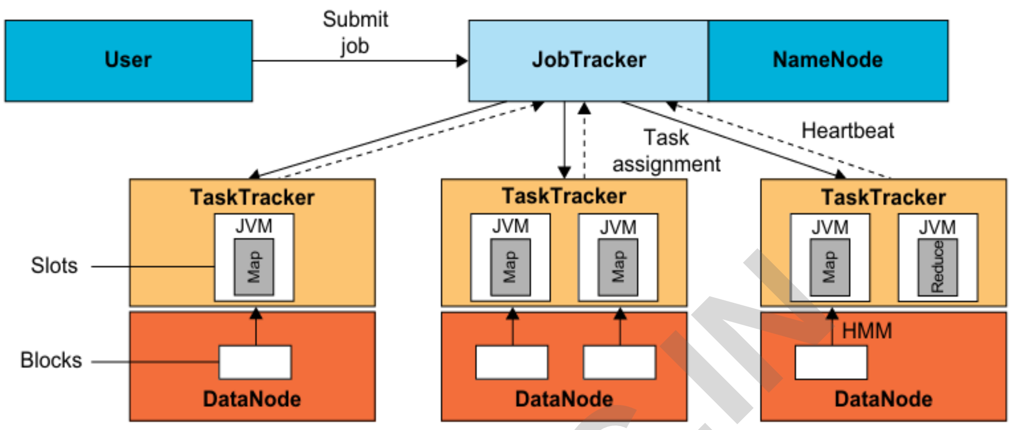


FIGURE 6.12

Data flow in running a MapReduce job at various task trackers using the Hadoop library.

Dryad and DryadLINQ from Microsoft

Two runtime software environments are reviewed in this section for parallel and distributed computing, namely the Dryad and DryadLINQ, both developed by Microsoft.

Dryad

- Flexibility Over MapReduce: Dryad allows users to define custom data flows using directed acyclic graphs (DAGs), unlike the fixed structure of MapReduce.
- DAG-Based Execution: Vertices represent computation engines, while edges are communication channels. The job manager assigns tasks and monitors execution.
- Job Manager & Name Server: The job manager builds, deploys, and schedules jobs, while the name server provides information about available computing resources.
- 2D Pipe System: Unlike traditional UNIX pipes (1D), Dryad's 2D distributed pipes enable large-scale parallel processing across multiple nodes.
- Fault Tolerance: Handles vertex failures by reassigning jobs and channel failures by recreating communication links.
- Broad Applicability: Supports scripting languages, MapReduce programming, and SQL integration, making it a versatile framework.

DryadLINQ from Microsoft

Built on **Microsoft's Dryad** execution framework, combining **Dryad** (distributed execution engine) with **.NET LINQ** (database-style query language).

Goal: Enables large-scale distributed computing for ordinary programmers.

Execution Flow (Key Steps):

- A **.NET application** creates a **DryadLINQ expression object**.
- Calling **ToDryadTable** triggers parallel execution.
- **DryadLINQ compiles the LINQ expression** into a distributed **Dryad execution plan**.
- A **Dryad job manager** oversees execution and monitors progress.
- The **job graph** is created and **vertices are scheduled**.
- **Vertices execute their assigned computations**.
- **Results are written to output tables**.
- The **job manager terminates**, returning control to **DryadLINQ**.
- Users **access the results via a Dryad Table iterator**.

Sawzall and Pig Latin High-Level Languages

 Sawzall Overview: A high-level scripting language built on Google's MapReduce framework for parallel data processing.

- Purpose & Development: Created by Rob Pike to process Google's log files, transforming batch processing into interactive sessions.
- Distributed Data Processing: Handles large-scale datasets, offering fault tolerance and efficiency.
- Execution Process:
  - Data partitioning & local processing with filtering.
  - Aggregation for final result computation.
  - Sawzall runtime engine translates scripts into MapReduce programs, distributing tasks across multiple nodes.
- Scalability & Reliability: Leverages cluster computing and redundant servers to ensure performance.
- Open Source Project: Recently released for public use.

 Pig Latin Overview: A high-level **data flow language** developed by **Yahoo!**, implemented in **Apache Pig** on **Hadoop**.

- **Comparison with Other Languages:**

- **DryadLINQ** is based on **SQL**.
- **Pig Latin & Sawzall** stem from **NoSQL**, though **Pig Latin supports SQL-like features** (e.g., **Join**, which Sawzall lacks).

- **Parallelism Automation:** Users focus on **data manipulation** while parallel execution is handled automatically.
- **Historical Influence:** Similar parallelism concepts appeared in **High-Performance Fortran**.
- **Language Features:** Discussions cover **data types** and **operators** used in Pig Latin.

Mapping Applications to Parallel and Distributed Systems

Fox's Application Architectures: Initially, five categories were defined for mapping applications to hardware/software, mainly focusing on simulations.

Category Breakdown:

1. SIMD (Single Instruction, Multiple Data): Parallel execution with lock-step operations (historically relevant but less used now).
2. SPMD (Single Program, Multiple Data) on MIMD machines: Each unit runs the same program asynchronously, commonly used for dynamic irregular problems.
3. Asynchronous Parallelism: Objects interact asynchronously, used in OS threads, event-driven simulations, and gaming applications (less common in large-scale parallelism).
4. Disconnected Parallel Components: Simple tasks that require minimal inter-node communication, suitable for grids/clouds.
5. Coarse-Grained Workflow Linkage: Breaks problems into atomic components, useful in metaproblem approaches like workflow modeling.

- Emerging Category (MapReduce++): A sixth category was introduced for data-intensive computing, divided into:
- Map-only applications (similar to category 4).
- Classic MapReduce (file-to-file operations).
- Extended MapReduce (supporting additional synchronous data processing).

PROGRAMMING SUPPORT OF GOOGLE APP ENGINE

Supported Languages

- Java: Comes with Eclipse plug-in and GWT (Google Web Toolkit).
- Python: Supports Django, CherryPy, and Google's built-in webapp environment.
- Other Languages: JVM-based interpreters enable JavaScript and Ruby compatibility.

Data Management

- NoSQL Data Store: Schema-less, entity-based system (max size: 1MB per entity).
- Java Interfaces: JDO & JPA via Data Nucleus Access platform.
- Python Interface: SQL-like GQL.
- Transactions: Strong consistency with optimistic concurrency control.
- Memcache: Speeds up data retrieval; works independently or with the datastore.
- Blobstore: Supports large files up to 2GB.

External Connectivity & Communication

- Secure Data Connection (SDC): Allows tunneling between intranet and GAE.
- URL Fetch Service: Fetches web resources via HTTP/HTTPS.
- Google Data API: Enables interaction with services like Maps, Docs, YouTube, Calendar.
- Google Accounts Authentication: Simplifies user login using existing Google accounts.
- Email Mechanism: Built-in system for sending emails from GAE applications.

Task & Resource Management

- Cron Service: Automates scheduled tasks (daily, hourly, etc.).
- Task Queues: Allows asynchronous background processing.
- Resource Quotas: Controls resource consumption, ensuring budget adherence.
- Free Tier: Usage is free up to certain quota limits.

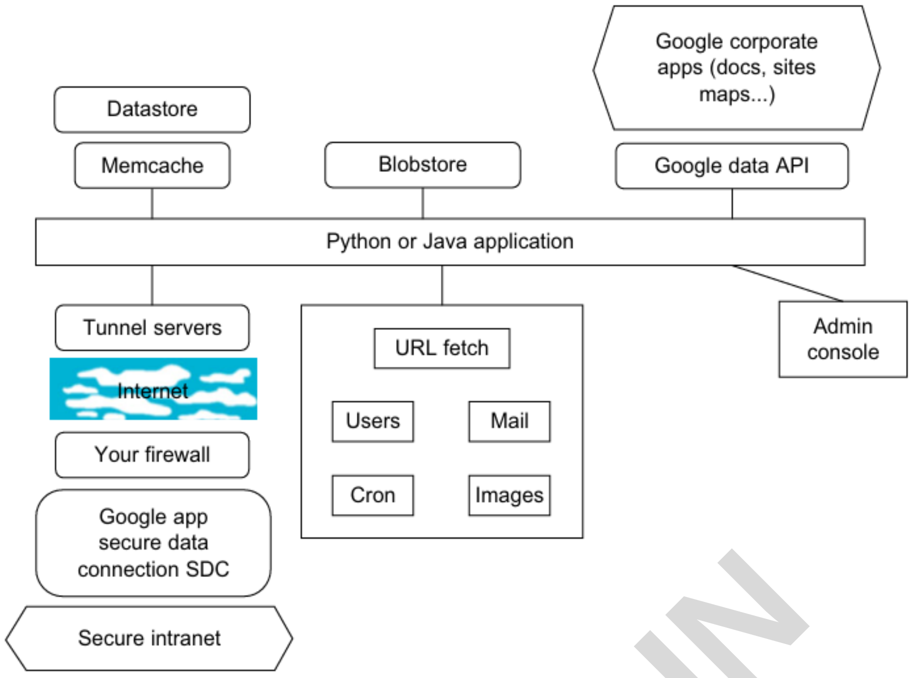


FIGURE 6.17

Programming environment for Google AppEngine.

Google File System

- Fundamental storage service for Google's search engine.
- Designed for massive data storage on commodity hardware.
- Optimized for large file sizes (100MB to several GB).
- Not a traditional POSIX-compliant file system but offers a customized API for Google applications.

Key Features

- 64MB block size (much larger than traditional 4KB blocks).
- Optimized for sequential writes & streaming reads (minimal random access).
- Reliability through replication (each chunk stored on at least three chunk servers).
- Single master manages metadata and coordinates file access.
- Shadow master provides backup to prevent single-point failure.
- No caching (as large-scale sequential reads/writes do not benefit from locality).
- Supports snapshot & record append operations.

Architecture & Data Flow

- Single master oversees chunk servers, which store the actual data.
- The master handles namespace & locking, ensuring system integrity.
- The master periodically interacts with chunk servers for load balancing and failure recovery.
- Clients communicate directly with chunk servers for data operations.

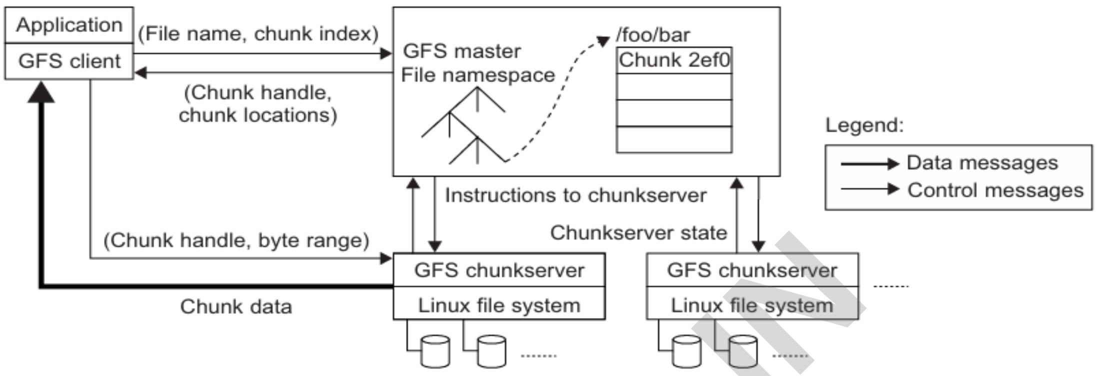


FIGURE 6.18

Architecture of Google File System (GFS).

Data Mutation Process (Writes & Appends)

1. Client queries master for chunk location & replicas.
2. Master designates the primary chunk server and provides replica locations.
3. Client pushes data to all replicas.
4. Primary assigns serial numbers to mutations and applies changes.
5. Primary forwards mutations to secondary replicas.
6. Secondaries acknowledge completion.
7. Primary informs client of success or errors.

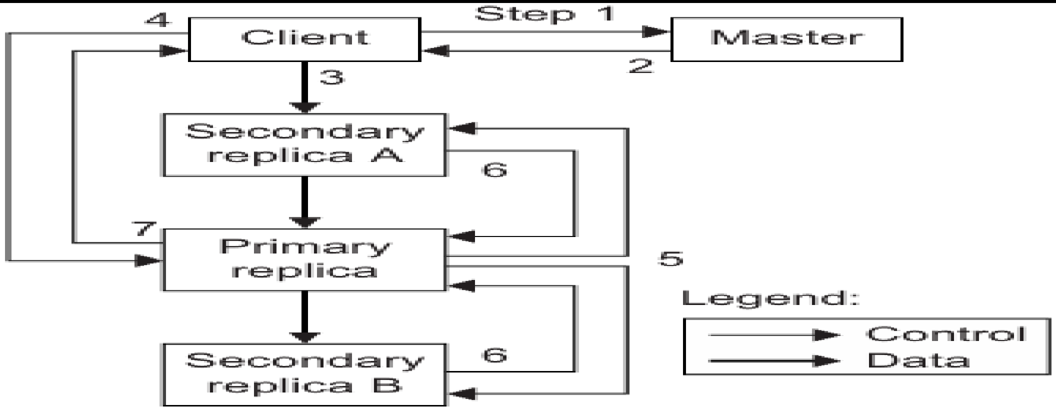


FIGURE 6.19

Data mutation sequence in GFS.

Fault Tolerance & Performance Optimization

- Fast recovery: Master & chunk servers restart in seconds.
- Replication ensures resilience: Each chunk is stored in at least three locations.
- Checksum validation for data integrity.
- Designed for large-scale distributed computing with high availability (HA).
- Optimized for frequent append operations, which is crucial for applications like web crawlers.

Google File System revolutionized big data storage, making large-scale processing possible on commodity hardware.

BigTable, Google's NOSQL System

- Designed for **storing and retrieving structured & semi-structured data**.
- Used for **web pages, per-user data, and geographic locations** in services like Google Search, Google Earth, and Orkut.
- Handles massive **data scales** with billions of URLs, petabytes of storage, and millions of queries per second.

Motivation for BigTable

- Commercial databases couldn't handle such large-scale data.
- Performance optimization was critical for Google's growing data needs.
- Low-level storage optimizations improved speed and efficiency

Key Features

- Distributed multilevel map structure for fault-tolerant and persistent data storage.

- Scalability: Supports thousands of servers, petabytes of data, and millions of read/write operations per second.
- Self-managing system: Dynamic addition/removal of servers with automatic load balancing.
- Efficient data access: Supports high read/write rates, large dataset scans, and historical data examination

BigTable's Building Blocks

1. Google File System (GFS) – Stores persistent data.
2. Scheduler – Manages job scheduling.
3. Lock Service (Chubby) – Handles master election and location bootstrapping.
4. MapReduce – Used for reading and writing BigTable data.

BigTable plays a crucial role in Google's cloud infrastructure, offering a highly scalable and efficient data management system tailored for its massive data needs

Chubby, Google's Distributed Lock Service

- ➢ Chubby Overview: A coarse-grained locking service that provides a simple namespace for storing small files.
- ➢ Reliability: Uses the Paxos agreement protocol, ensuring consistency even if member nodes fail.
- ➢ Architecture: Each Chubby cell has five servers, all sharing the same file system namespace.
- ➢ Client Interaction: Clients communicate with Chubby servers via the Chubby library to perform file operations.
- ➢ Primary Use: Serves as Google's internal name service, helping GFS and BigTable elect primary replicas

PROGRAMMING ON AMAZON AWS AND MICROSOFT AZURE

AWS platform and its updated service offerings

- ✓ EC2 – Runs virtual servers (computers in the cloud).
- ✓ S3 – Stores data and files in the cloud.
- ✓ SimpleDB – Stores structured NoSQL data.
- ✓ RDS – Manages SQL databases (like MySQL, PostgreSQL).
- ✓ Elastic MapReduce (EMR) – Runs big data jobs using Hadoop on EC2.
- ✓ No BigTable – AWS does not support Google BigTable.
- ✓ SQS – Sends and receives messages between services.

- ✓ SNS – Sends notifications to users or services.
- ✓ Auto-Scaling – Adds or removes EC2 instances automatically based on demand.
- ✓ Elastic Load Balancing – Distributes traffic across multiple EC2 instances.
- ✓ CloudWatch – Monitors EC2 performance (CPU, disk, network).

Programming on Amazon EC2

- ✓ Amazon introduced VM-based hosting – Customers rent virtual machines (VMs) instead of physical servers.
- ✓ Elastic service – Users can start or stop VMs anytime and pay only for usage.
- ✓ Amazon Machine Images (AMIs) – Preconfigured VM templates with OS and software.
- ✓ VM setup steps – Create AMI → Create Key Pair → Configure Firewall → Launch.
- ✓ AMI types:
- ✓ Private AMI – Created by you, used only by you (or shared with selected users).
- ✓ Public AMI – Shared by users for public use, free to launch.
- ✓ Paid AMI – Created by users, available for a fee per hour (plus AWS cost).

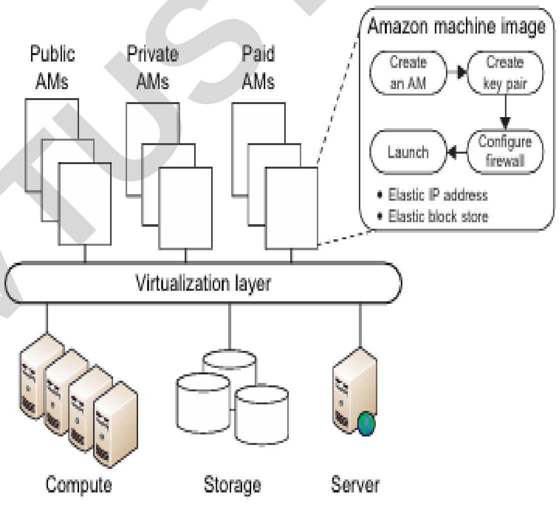


FIGURE 6.23

Amazon EC2 execution environment.

Amazon Simple Storage Service (S3)

- ✓ **Stores any amount of data**, accessible anytime from anywhere.
- ✓ **Uses objects** stored in **buckets**, accessed via a unique **key**.

- ✓ **Key-value storage** model; supports objects from **1 byte to 5 GB**.
- ✓ Access via **REST** or **SOAP** web interfaces.
- ✓ Supports **metadata**, **access control**, and **per-object URLs/ACLs**.
- ✓ Provides **very high durability (99.999999999%)** and **availability (99.99%)**.
- ✓ Supports **public/private access** and **authenticated security**.
- ✓ Offers **BitTorrent support** for large-scale downloads.
- ✓ **Pricing**: Starts from **$0.055 to $0.15 per GB/month**, first **1 GB free**.
- ✓ **No charge** for data transfers between S3 and EC2 in the same region.

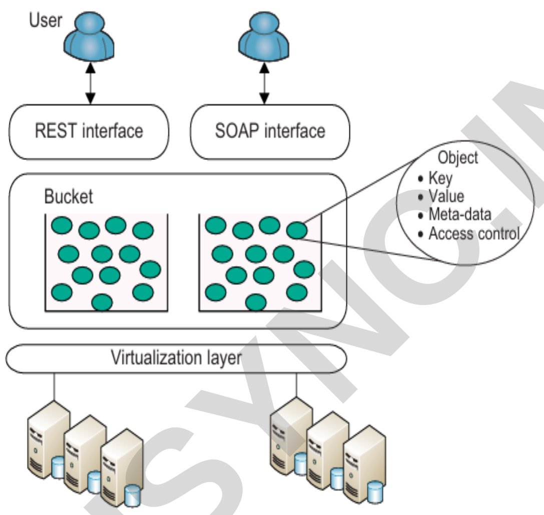


FIGURE 6.24

Amazon S3 execution environment.

Amazon Elastic Block Store (EBS) and SimpleDB

Amazon Elastic Block Store (EBS)

- Provides persistent block storage for EC2 instances.
- Acts like a distributed file system, supporting OS disk access.
- Storage volumes range from **1GB to 1TB**; multiple volumes can be attached to one instance.
- Can be formatted into a file system or used as raw block storage.
- Snapshots allow **incremental backups** for improved performance.
- **Pricing:** $0.10 per GB/month + $0.10 per million I/O requests.
- Open-source alternative: **Nimbus**.

Amazon SimpleDB

- Schema-free **database service** for structured data storage.

- Organizes data into **domains (like tables)** containing items (rows).
- Allows **multiple values per cell**, unlike relational databases.
- Removes need for **strong schema enforcement**.
- **Pricing:** $0.140 per machine hour; first **25 hours free** per month.
- Designed for **metadata management**, compared to BigTable which handles big data.
- Related to Amazon Dynamo, an early research system.

Microsoft Azure Programming Support

- ❑ **Azure Fabric** – Manages cloud servers and keeps apps running smoothly.
- ❑ **XML Templates** – Used to set up and run cloud services automatically.
- ❑ **No live debugging** – You check logs and traces instead.
- ❑ **Storage options** – Includes Blobs (files), Tables (data), Queues (messages), and Drives (disks).
- ❑ **SQL Azure** – Cloud-based SQL database.
- ❑ **Web Role** – Runs websites and handles web traffic.
- ❑ **Worker Role** – Runs background tasks or heavy processing.
- ❑ **Main role methods**:
  - ✓ OnStart() – Runs when the app starts.
  - ✓ Run() – Runs the main app code.
  - ✓ OnStop() – Runs when the app shuts down.
- ❑ **Roles can be load balanced** – Work is shared across servers for better performance.
- ❑ **Appliances** – Pre-made virtual machines for specific tasks.

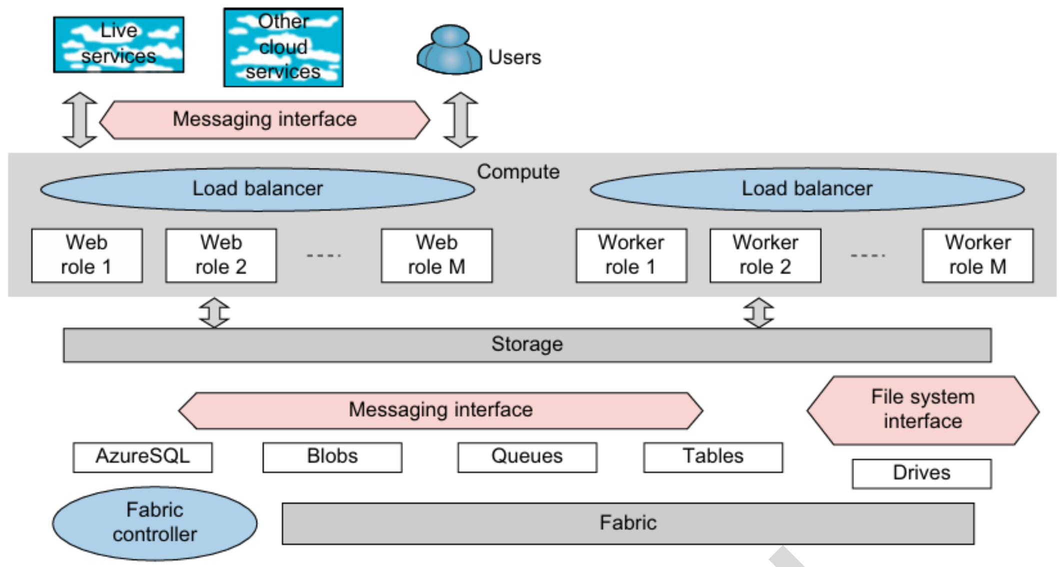


FIGURE 6.25

Features of the Azure cloud platform.

SQLAzure

- **SQLAzure** provides **SQL Server as a service** with REST API access.
- Offers **Drives**, similar to **Amazon EBS**, providing durable NTFS volumes backed by **blob storage**.
- **REST interfaces** automatically associate storage with **URLs**.
- **Data replication** occurs **three times** for **fault tolerance** and **consistency**.
- **Storage system** is based on **blobs**, similar to **Amazon S3**.
- **Blobs Hierarchy**: **Account → Containers → Block or Page Blobs**.
    - **Containers** act like directories.
    - **Block blobs**: Optimized for **streaming data**, consist of **blocks (max 4MB each)** and support up to **200GB** in size.
    - **Page blobs**: Used for **random read/write**, with a maximum blob size of **1TB**.
- **Metadata** can be stored as **pairs** (up to **8KB per blob**).

Azure Tables

Azure Queue Storage

- Handles reliable message delivery for web and worker roles.
- Supports unlimited messages, each up to 8KB in size.

- Provides PUT, GET, DELETE message operations & CREATE, DELETE for queues.

Azure Table Storage

- A scalable NoSQL storage system for structured data.
- Organizes data into entities (rows) and properties (columns).
- No limit on number of entities; designed for distributed computing.
- Each entity can have up to 255 properties formatted as .
- Requires PartitionKey (grouping entities) and RowKey (unique identifier).
- PartitionKey usage enhances search performance.
- Maximum storage per entity: 1MB (larger values stored in blob storage).
- Supports ADO.NET and LINQ for querying.

EMERGING CLOUD SOFTWARE ENVIRONMENTS

Open Source Eucalyptus

Eucalyptus is an open-source cloud computing platform, originally developed at UC Santa Barbara.

- Provides AWS-compliant services, including an EC2-based web interface for cloud management.
- Includes Walrus, a block storage system similar to Amazon S3, for storing VM images.
- Supports both computer cloud and storage cloud environments.
- Allows users to create, upload, and register VM images, linking them with a kernel and ramdisk.
- Images are stored in Walrus buckets and can be retrieved from any availability zone.
- Enables the creation of specialized virtual appliances for easy deployment.
- Available in commercial proprietary and open-source versions.

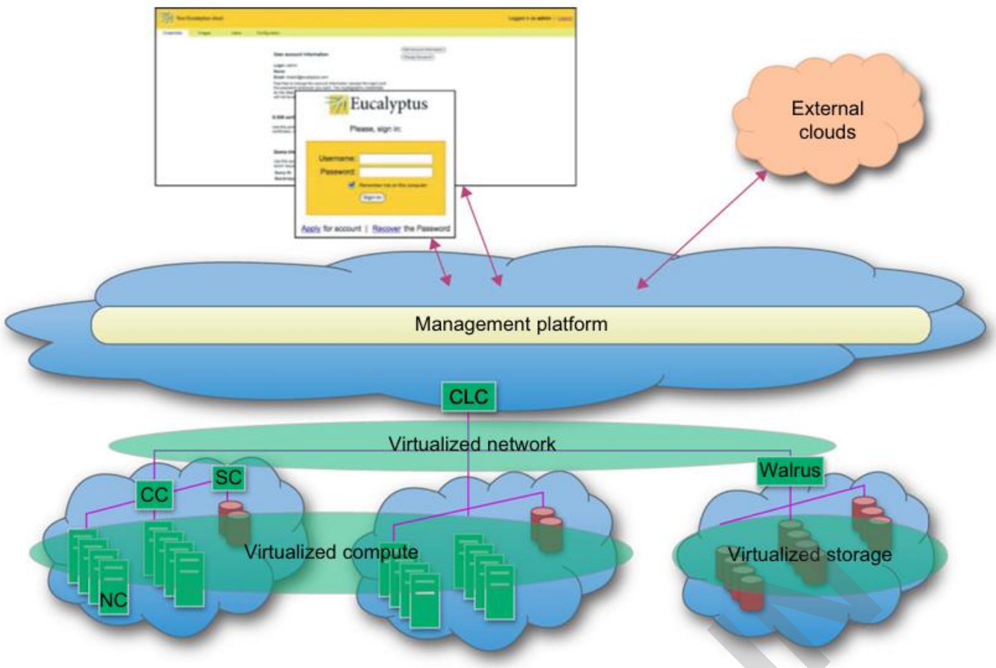


FIGURE 6.26
The Eucalyptus architecture for VM image management.

Nimbus

- Nimbus is an open-source IaaS cloud computing solution.
- Allows clients to lease remote resources by deploying VMs.
- Features Nimbus Web, a Python Django-based user-friendly interface.
- Includes Cumulus, a storage cloud system compatible with Amazon S3 REST API, supporting quota management.
- Works with boto, s2cmd, and Jets3t for cloud storage interactions.
- Supports two resource management modes:
  - Resource Pool Mode: Direct control of VM manager nodes.
  - Pilot Mode: Uses LRMS to request VM managers.
- Provides Amazon EC2-compatible API for cloud deployments.

(Courtesy of Eucalyptus LLC [81])

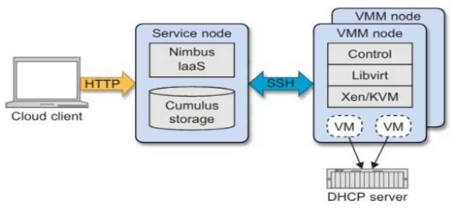


FIGURE 6.27
Nimbus cloud infrastructure.

OpenNebula, Sector/Sphere, and OpenStack

- OpenNebula is an open-source IaaS cloud toolkit for transforming existing infrastructure into a cloud.
- Flexible and modular architecture supports various storage, network, and hypervisor configurations.
- Core manages the full VM lifecycle, including network setup and storage management.
- Capacity manager (scheduler) supports basic and advanced scheduling policies.
- Access drivers abstract infrastructure and expose functionalities for monitoring, storage, and virtualization.
- Provides management interfaces, including libvirt API, CLI, and cloud interface.
- Supports live migration and VM snapshots for dynamic resource management.
- Enables hybrid cloud setups by integrating with Amazon EC2, Eucalyptus, and ElasticHosts.
- Includes an Image Repository for simplified disk image selection and multiuser environments.

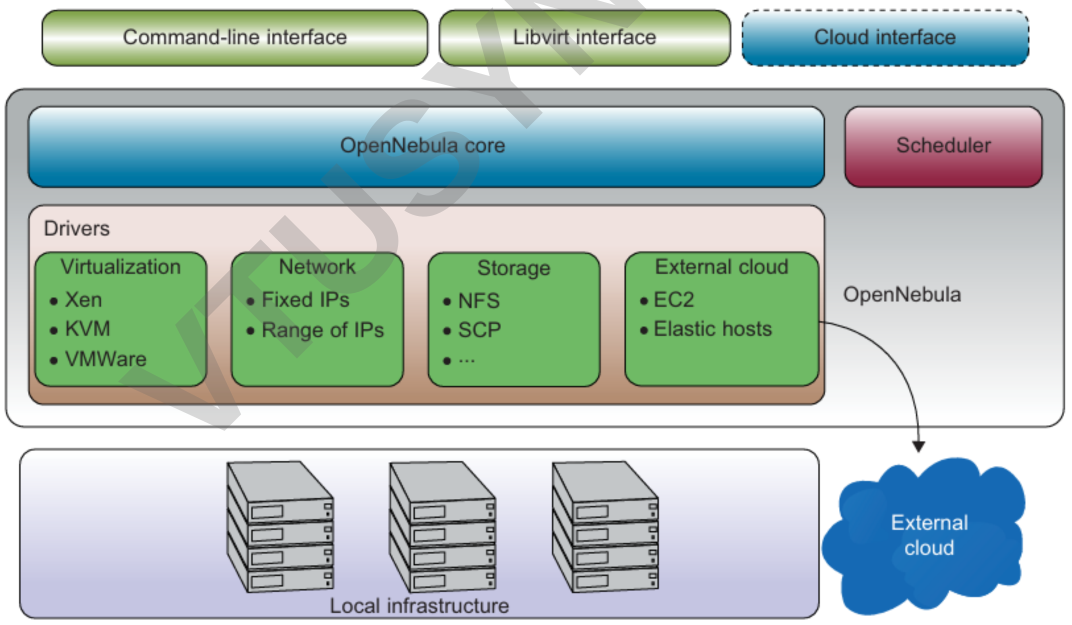


FIGURE 6.28

OpenNebula architecture and its main components.

Sector/Sphere

 Sector:

A distributed file system (DFS) for managing large datasets over wide-area networks.

- Provides fault tolerance through replication.
- Uses UDP for fast message passing and UDT for high-speed data transfer.
- Includes a programming API and FUSE user space file system module.

Sphere: A parallel data processing engine integrated with Sector.

- Supports job scheduling and data locality for efficient processing.
- Allows User Defined Functions (UDFs) to run in parallel across data segments.
- Provides fault tolerance by restarting failed data segments on other nodes.

Architecture Components:

- Security Server: Manages authentication.
- Master Servers: Handle metadata, job scheduling, and user requests.
- Slave Nodes: Store and process data in distributed environments.
- Client Component: Offers APIs and tools for accessing Sector data.
- Space Component: Supports column-based distributed data tables.
- Tables are stored by columns with no relationships between them.
- Implements basic SQL operations like table creation, modifications, and key-value updates.

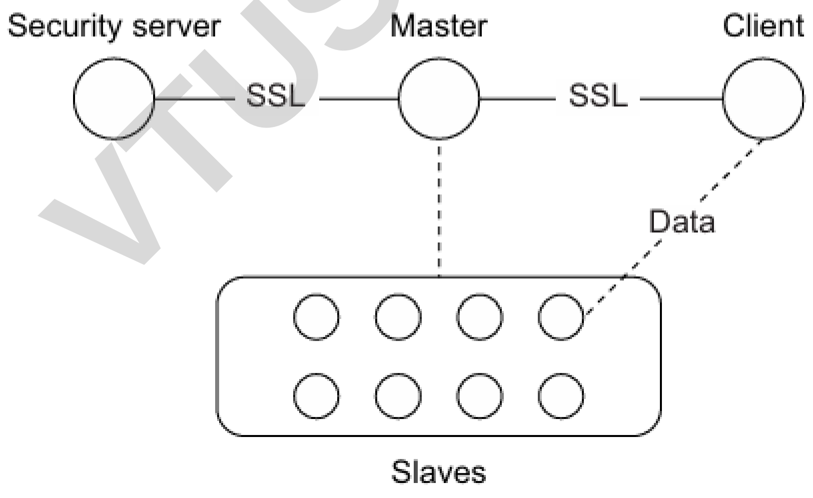


FIGURE 6.29

The Sector/Sphere system architecture.

OpenStack

- Founded – Introduced by Rackspace and NASA in July 2010.

- Purpose – Open-source cloud computing framework for scalable and secure infrastructure.
- Community – Involves technologists, developers, researchers, and industries sharing technologies.
- Key Components:
  - *Compute:* Manages large groups of virtual private servers.
  - *Storage:* Offers scalable object storage using clusters of commodity servers.
- Recent Development:
  - *Image repository prototyped.*
  - *Image registration & discovery service* – Locates and manages stored images.
  - *Image delivery service* – Transfers images to compute services from storage.
- Future Direction – Expanding service integration and enhancing its ecosystem.

OpenStack Compute

- OpenStack Nova is a cloud fabric controller within the OpenStack IaaS ecosystem, designed for efficient computing support.
- It follows a shared-nothing architecture, relying on message queues for communication and using deferred objects to prevent blocking.
- The system maintains state through a distributed data system with atomic transactions for consistency.
- Built in Python, Nova integrates boto (Amazon API) and Tornado (fast HTTP server for S3).
- The API Server processes requests and forwards them to the Cloud Controller, which handles system state, authorization (via LDAP), and node management.
- Nova also includes network components such as NetworkController (VLAN allocation), RoutingNode (NAT conversion & firewall rules), AddressingNode (DHCP services), and TunnelingNode (VPN connectivity), with network state stored in a distributed object store managing IP and subnet assignments. This ensures scalability, security, and efficient cloud resource management

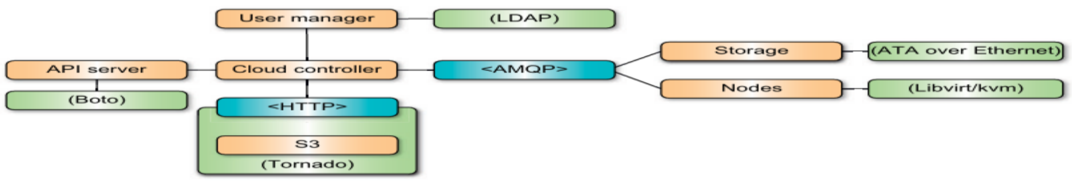


FIGURE 6.30
OpenStack Nova system architecture. The AMQP (Advanced Messaging Queuing Protocol) was described in Section 5.2.

OpenStack Storage

OpenStack's storage solution consists of several interconnected components.

- The **proxy server** routes requests and facilitates lookups for accounts, containers, and objects.
- **Rings** map entity names to physical locations while using **zones, devices, partitions, and replicas** for fault tolerance and resource isolation.
- **Object Server** stores, retrieves, and deletes binary objects with metadata, but requires specific file system support.
- **Container Server** lists objects, while **Account Server** manages container listings.
- The system ensures resilience through **replication, updaters, and auditors** while balancing partitions across heterogeneous storage clusters.
- OpenStack's first release, **Austin**, launched on **October 22, 2010**, supported by an active developer community.

Manjrasoft Aneka Cloud and Appliances

Developed by Manjrasoft (Melbourne, Australia), Aneka is a cloud application platform for deploying **parallel & distributed applications** on private or public clouds.

- **Deployment Options** – Can be hosted on **Amazon EC2** (public cloud) or a **private cloud** with restricted access.
- **Key Advantages:**
  - Supports **multiple programming & application environments**.
  - Allows **simultaneous execution** across multiple runtime environments.
  - Provides **rapid deployment tools & frameworks**.
  - Enables **dynamic resource allocation** based on **QoS/SLA requirements**.
  - Built on **Microsoft .NET**, with **Linux support via Mono**.
- **Three Core Capabilities:**

1. **Build** – Includes an SDK with APIs & tools for developing applications. Supports enterprise/private clouds, Amazon EC2, and hybrid clouds.

2. **Accelerate** – Allows rapid deployment across multiple runtime environments (Windows/Linux/UNIX), optimizing local resources & leasing extra capacity from public clouds when needed.

3. **Manage** – Offers tools (GUI & APIs) for **monitoring, managing, and scaling** Aneka compute clouds dynamically based on **SLA/QoS requirements**.

- **Programming Models Supported:**

1. **Thread Programming Model** – Best for utilizing **multicore nodes** in cloud computing.
2. **Task Programming Model** – Ideal for prototyping **independent task-based applications**.
3. **MapReduce Programming Model** – Efficient for large-scale **data processing & computation**.

Aneka Architecture

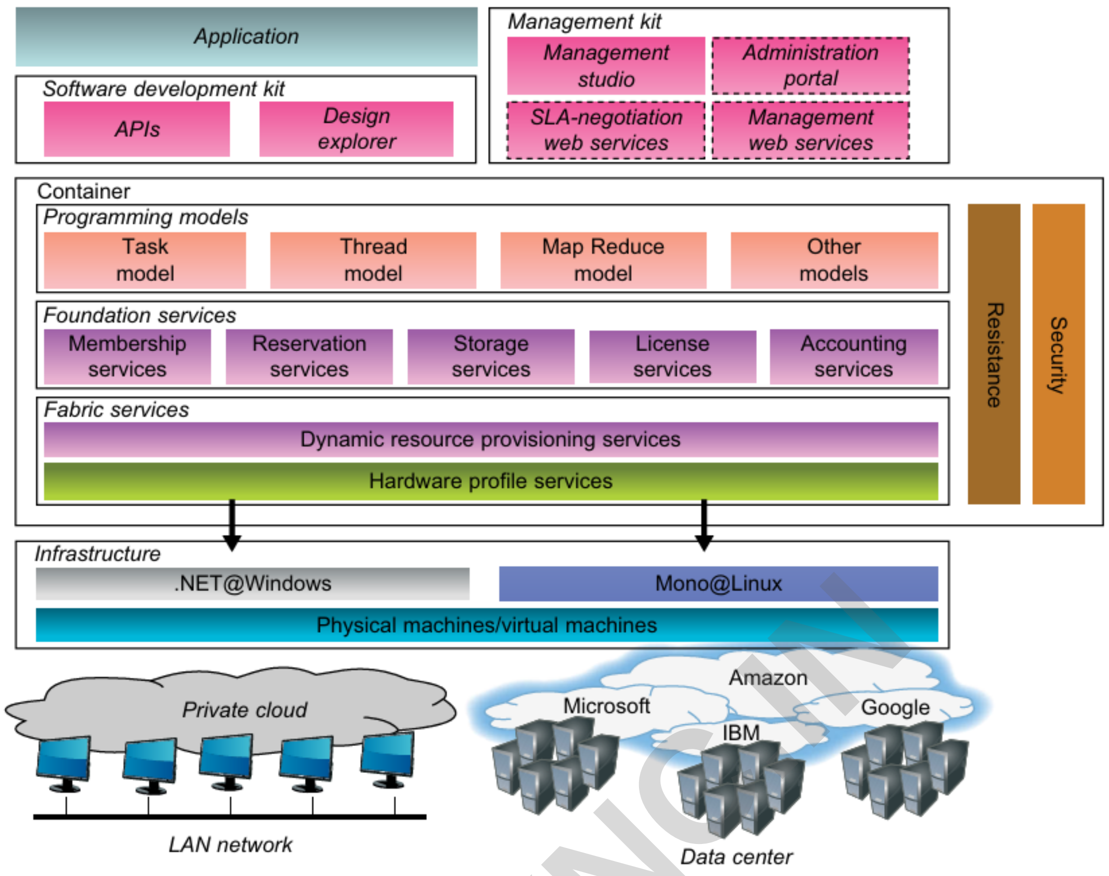


FIGURE 6.31

Architecture and components of Aneka.

Distributed Runtime Environment:

- Aggregates **physical & virtual nodes** hosting the Aneka container.
- **Container** – Manages services deployed on a node, interfacing with the hosting environment.
- Uses **Platform Abstraction Layer (PAL)** to hide OS heterogeneity & perform infrastructure tasks (monitoring & performance management).

- **Three Major Service Categories:**

1. **Fabric Services:**
   - Provide **HA (high availability) & failover**, node membership, resource provisioning, monitoring, and hardware profiling.
2. **Foundation Services:**
   - Core middleware functionalities like **storage management, resource reservation, reporting, accounting, billing, licensing, and services monitoring**.
3. **Application Services:**

- Manage application execution, **elastic scalability, data transfer, performance monitoring, and billing**.
- Supports **different application models & distributed programming patterns**.

- **Application Execution Model:**
    - Managed via **scheduling & execution services**.
    - Supports **distributed threads, bags of tasks, and MapReduce** programming models.
- **Extensibility & Customization:**
- Additional services can be designed & deployed dynamically.
- SDK offers **ready-to-use tools** for rapid service prototyping.
- Uses **Spring framework** for seamless integration of new services.

Virtual Appliances

- ✓ **Virtual appliances** -Ready-to-use VM images with all needed software pre-installed.
- ✓ Include OS, libraries, app code, and auto-setup scripts.
- ✓ **Run out-of-the-box** — no extra setup needed.
- ✓ Useful for **grid/cloud systems** — easier and faster to deploy apps.
- ✓ Help avoid software compatibility problems with host systems.
- ✓ Supported by tools like **Xen, VMware, KVM, VirtualBox**.
- ✓ **VM conversion tools** allow use across different platforms.
- ✓ Can be managed like a **local network**, even if spread across different networks.
- ✓ Example: **Grid Appliance** project — easy setup for distributed computing.