3.Explain about Virtualization structure and show how virtualization is achieved in CPU,memory and I/O devices.

4.Explain in detail about Virtual clusters.

5.Explain how resource management is done in cloud and virtualization is implemented in data centers ?.

UNIT IV

Hypervisor	A **hypervisor** or virtual machine monitor (VMM) is a piece of computer software, firmware or hardware that creates and runs virtual machines. A computer on which a **hypervisor** is running one or more virtual machines is defined as a host machine. Each virtual machine is called a guest machine.
Hadoop development	Apache **Hadoop** is an open-source software **framework** written in Java for distributed storage and distributed processing of very large data sets on computer clusters built from commodity hardware.
Map Reduce Computation	MapReduce is designed to continue to work in the face of system failures. When a job is running, MapReduce monitors progress of each of the servers participating in the job. If one of them is slow in returning an answer or fails before completing its work, MapReduce automatically starts another instance of that task on another server that has a copy of the data.
HDFS	Hadoop File System was developed using distributed file system design. It is run on commodity hardware. Unlike other distributed systems, HDFS is highly fault tolerant and designed using low-cost hardware. HDFS holds very large amount of data and provides easier access. To store such huge data, the files are stored across multiple machines.

OPEN SOURCE GRID MIDDLEWARE PACKAGES

As reviewed in Berman, Fox, and Hey , many software, middleware, and programming environments have been developed for grid computing over past 15 years. Below we assess their relative strength and limitations based on recently reported applications. We first introduce some grid standards and popular APIs. Then we present the desired software support and middleware developed for grid computing includes four grid middleware packages.

Grid Software Support and Middleware Packages

BOINC Berkeley Open Infrastructure for Network Computing.

UNICORE Middleware developed by the German grid computing community.

Globus (GT4) A middleware library jointly developed by Argonne National Lab., Univ. of Chicago, and USC Information Science Institute, funded by DARPA, NSF, and NIH. CGSP in

ChinaGrid

The CGSP (ChinaGrid Support Platform) is a middleware library developed by 20 top universities in China as part of the ChinaGrid Project .

Condor-G Originally developed at the Univ. of Wisconsin for general distributed computing, and later extended to Condor-G for grid job management. .

Sun Grid Engine (SGE)

Developed by Sun Microsystems for business grid applications. Applied to private grids and local clusters within enterprises or campuses.

Grid Standards and APIs

Grid standards have been developed over the years. The Open Grid Forum (formally Global Grid Forum) and Object Management Group are two well-formed organizations behind those standards. We have already introduced the OGSA (Open Grid Services Architecture) in standards including the GLUE for resource representation, SAGA (Simple API for Grid Applications), GSI (Grid Security Infrastructure), OGSI (Open Grid Service Infrastructure), and WSRE (Web Service Resource Framework).

The grid standards have guided the development of several middleware libraries and API tools for grid computing. They are applied in both research grids and production grids today. Research grids tested include the EGEE, France Grilles, D-Grid (German), CNGrid (China), TeraGrid (USA), etc. Production grids built with the standards include the EGEE, INFN grid (Italian), NorduGrid, Sun Grid, Techila, and Xgrid . We review next the software environments and middleware implementations based on these standards.

Software Support and Middleware

Grid middleware is specifically designed a layer between hardware and the software. The middleware products enable the sharing of heterogeneous resources and managing virtual organizations created around the grid. Middleware glues the allocated resources with specific user applications. Popular grid middleware tools include the Globus Toolkits (USA), gLight, UNICORE (German), BOINC (Berkeley), CGSP (China), Condor-G, and Sun Grid Engine, etc. summarizes the grid software support and middleware packages developed for grid systems since 1995. In subsequent sections, we will describe the features in Condor-G, SGE, GT4, and CGSP.

THE GLOBUS TOOLKIT ARCHITECTURE (GT4)

The Globus Toolkit, started in 1995 with funding from DARPA, is an open middleware library for the grid computing communities. These open source software libraries support many operational grids and their applications on an international basis. The toolkit addresses common problems and issues related to grid resource discovery, management, communication, security, fault detection, and portability. The software itself provides a variety of components and capabilities. The library includes a rich set of service implementations. The implemented software supports grid infrastructure management, provides tools for building new web services in Java, C, and Python, builds a powerful standard-based security infrastructure and client APIs (in different languages), and offers comprehensive command-line programs for accessing various grid services. The Globus Toolkit was initially motivated by a desire to remove obstacles that prevent seamless collaboration, and thus sharing of resources and services, in scientific and engineering applications. The shared resources can be computers, storage, data, services, networks, science instruments (e.g., sensors), and so on. The Globus library version GT4. Globus Tookit GT4 supports distributed and cluster computing services. Courtesy of I. Foster

The GT4 Library

GT4 offers the middle-level core services in grid applications. The high-level services and tools, such as MPI, Condor-G, and Nirod/G, are developed by third parties for general-purpose distributed computing applications. The local services, such as LSF, TCP, Linux, and Condor, are at the bottom level and are fundamental tools supplied by other developers summarizes GT4's core grid services by module name. Essentially, these functional modules help users to discover available resources, move data between sites, manage user credentials, and so on. As a de facto standard in grid middleware, GT4 is based on industry-standard web service technologies.

Functional Modules in Globus GT4 Library

Service Functionality Module Name Functional Description

Global Resource Allocation Manager GRAM Grid Resource Access and Management (HTTP-based)

Communication Nexus Unicast and multicast communication

Grid Security Infrastructure GSI Authentication and related security services

Monitory and Discovery Service MDS Distributed access to structure and state information

Health and Status HBM Heartbeat monitoring of system components

Global Access of Secondary Storage GASS Grid access of data in remote secondary storage

Grid File Transfer GridFTP Inter-node fast file transfer

Nexus is used for collective communications and HBM for heartbeat monitoring of resource nodes. GridFTP is for speeding up internode file transfers. The module GASS is used for global access of secondary storage. More details of the functional modules of Globus GT4 and their applications are available at www.globus.org/toolkit/.

Globus Job Workflow

The typical job workflow when using the Globus tools. A typical job execution sequence proceeds as follows: The user delegates his credentials to a delegation service. The user submits a job request to GRAM with the delegation identifier as a parameter. GRAM parses the request, retrieves the user proxy certificate from the delegation service, and then acts on behalf of the user. GRAM sends a transfer request to the RFT (Reliable File Transfer), which applies GridFTP to bring in the necessary files. GRAM invokes a local scheduler via a GRAM adapter and the SEG (Scheduler Event Generator) initiates a set of user jobs. The local scheduler reports the job state to the SEG. Once the job is complete, GRAM uses RFT and GridFTP to stage out the resultant files. The grid monitors the progress of these operations and sends the user a notification when they succeed, fail, or are delayed.

Globus job workflow among interactive functional modules.

Client-Globus Interactions

GT4 service programs are designed to support user applications .There are strong interactions between provider programs and user code. GT4 makes heavy use of industry-standard web service protocols and mechanisms in service description, discovery, access, authentication, authorization, and the like. GT4 makes extensive use of Java, C, and Python to write user code. Web service mechanisms define specific interfaces for grid computing. Web services provide flexible, extensible, and widely adopted XML-based interfaces.

Client and GT4 server interactions; vertical boxes correspond to service programs and horizontal boxes represent the user codes. Courtesy of Foster and Kesselman GT4 components do not, in general, address end-user needs directly. Instead, GT4 provides a set of infrastructure services for accessing, monitoring, managing, and controlling access to infrastructure elements. The server code in the vertical boxes in corresponds to 15 grid services that are in heavy use in the GT4 library. These demand computational, communication, data, and storage resources. We must enable a range of end-user tools that provide the higher-level capabilities needed in specific user applications. Wherever possible, GT4 implements standards to facilitate construction of operable and reusable user code. Developers can use these services and libraries to build simple and complex systems quickly.

A high-security subsystem addresses message protection, authentication, delegation, and authorization. Comprising both a set of service implementations (server programs at the bottom of Figure 7.21) and associated client libraries at the top, GT4 provides both web services and non-WS applications. The horizontal boxes in the client domain denote custom applications and/or third-party tools that access GT4 services. The toolkit programs provide a set of useful infrastructure services. Globus container serving as a runtime environment for implementing web services in a grid platform. Courtesy of Foster and Kesselman Three containers are used to host user-developed services written in Java, Python, and C, respectively. These containers provide implementations of security, management, discovery, state management, and other mechanisms frequently required when building services. They extend open source service hosting environments with support for a range of useful web service specifications, including WSRF, WS-Notification, and WS-Security. A set of client libraries allow client programs in Java, C, and Python to invoke operations on both GT4 and user-developed services. In many cases, multiple interfaces provide different levels of control: For example, in the case of GridFTP, there is not only a simple command-line client (globus-url-copy) but also control and data channel libraries for use in programs—and the XIO library allowing for the integration of alternative transports. The use of uniform abstractions and mechanisms means clients can interact with different services in similar ways, which facilitates construction of complex, interoperable systems and encourages code reuse Parallel Computing and Programming Paradigms

Consider a distributed computing system consisting of a set of networked nodes or workers. The system issues for running a typical parallel program in either a parallel or a distributed manner would include the following :

Partitioning This is applicable to both computation and data as follows:

Computation partitioning This splits a given job or a program into smaller tasks. Partitioning greatly depends on correctly identifying portions of the job or program that can be performed concurrently. In other words, upon identifying parallelism in the structure of the program, it can be divided into parts to be run on different workers. Different parts may process different data or a copy of the same data.

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

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

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

Multiple accesses to a shared resource by different workers may raise race conditions, whereas data dependency happens when a worker needs the processed data of other workers.

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

Scheduling For a job or program, when the number of computation parts (tasks) or data pieces is more than the number of available workers, a scheduler selects a sequence of tasks or data pieces to be assigned to the workers. It is worth noting that the resource allocator performs the actual mapping of the computation or data pieces to workers, while the scheduler only picks the next part from the queue of unassigned tasks based on a set of rules called the scheduling policy. For multiple jobs or programs, a scheduler selects a sequence of jobs or programs to be run on the distributed computing system. In this case, scheduling is also necessary when system resources are not sufficient to simultaneously run multiple jobs or programs.

Motivation for Programmig Paradigms

Because handling the whole data flow of parallel and distributed programming is very time-consuming and requires specialized knowledge of programming, dealing with these issues may affect the productivity of the programmer and may even result in affecting the program's time to market. Furthermore, it may detract the programmer from concentrating on the logic of

the program itself. Therefore, parallel and distributed programming paradigms or models are offered to abstract many parts of the data flow from users.

In other words, these models aim to provide users with an abstraction layer to hide implementation details of the data flow which users formerly ought to write codes for. Therefore, simplicity of writing parallel programs is an important metric for parallel and distributed programming paradigms. Other motivations behind parallel and distributed programming models are (1) to improve productivity of programmers, (2) to decrease programs' time to market, (3) to leverage underlying resources more efficiently, (4) to increase system throughput, and (5) to support higher levels of abstraction .

MapReduce, Hadoop, and Dryad are three of the most recently proposed parallel and distributed programming models. They were developed for information retrieval applications but have been shown to be applicable for a variety of important applications . Further, the loose coupling of components in these paradigms makes them suitable for VM implementation and leads to much better fault tolerance and scalability for some applications than traditional parallel computing models such as MPI .

MapReduce, Twister, and Iterative MapReduce

MapReduce, is a software framework which supports parallel and distributed computing on large data sets . This software framework abstracts the data flow of running a parallel program on a distributed computing system by providing users with two interfaces in the form of two functions:

Map and Reduce. Users can override these two functions to interact with and manipulate the data flow of running their programs illustrates the logical data flow from the Map to the Reduce function in MapReduce frameworks. In this framework, the ¯value part of the data, (key, value), is the actual data, and the ¯key part is only used by the MapReduce controller to control the data flow .

Formal Definition of MapReduce

The MapReduce software framework provides an abstraction layer with the data flow and flow of control to users, and hides the implementation of all data flow steps such as data partitioning, mapping, synchronization, communication, and scheduling. Here, although the data flow in such frameworks is predefined, the abstraction layer provides two well-defined interfaces in the form of two functions: Map and Reduce . These two main functions can be overridden by the user to achieve specific objectives the MapReduce framework with data flow and control flow.Therefore, the user overrides the Map and Reduce functions first and then invokes the provided MapReduce (Spec, & Results) function from the library to start the flow of data. The MapReduce function, MapReduce (Spec, & Results), takes an important parameter which is a specification object, the Spec. This object is first initialized inside the user's program, and then the user writes code to fill it with the names of input and output files, as well as other optional tuning parameters. This object is also filled with the name of the Map and Reduce functions to identify these user-defined functions to the MapReduce library.

The overall structure of a user's program containing the Map, Reduce, and the Main functions is given below. The Map and Reduce are two major subroutines. They will be called to implement the desired function performed in the main program.

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

}

MapReduce Logical Data Flow

The input data to both the Map and the Reduce functions has a particular structure. This also pertains for the output data. The input data to the Map function is in the form of a (key, value) pair. For example, the key is the line offset within the input file and the value is the content of the line. The output data from the Map function is structured as (key, value) pairs called intermediate (key, value) pairs. In other words, the user-defined Map function processes each input (key, value) pair and produces a number of (zero, one, or more) intermediate (key, value) pairs. Here, the goal is to process all input (key, value) pairs to the Map function in parallel .

MapReduce logical data flow in 5 processing stages over successive (key, value) pairs. In turn, the Reduce function receives the intermediate (key, value) pairs in the form of a group of intermediate values associated with one intermediate key, (key, [set of values]). In fact, the MapReduce framework forms these groups by first sorting the intermediate (key, value) pairs and then grouping values with the same key. It should be noted that the data is sorted to simplify the grouping process. The Reduce function processes each (key, [set of values]) group and produces a set of (key, value) pairs as output. To clarify the data flow in a sample MapReduce application, one of the well-known

MapReduce problems, namely word count, to count the number of occurrences of each word in a collection of documents is presented here demonstrates the data flow of the word-count problem for a simple input file containing only two lines as follows: (1) ¯most people ignore most poetry and (2) ¯most poetry ignores most people. In this case, the Map function simultaneously produces a number of intermediate (key, value) pairs for each line of content so that each word is the intermediate key with 1 as its intermediate value; for example, (ignore, 1). Then the MapReduce library collects all the generated intermediate (key, value) pairs and sorts them to group the 1's for identical words; for example, (people, [1,1]). Groups are then sent to the Reduce function in parallel so that it can sum up the 1 values for each word and generate the actual number of occurrence for each word in the file; for example, (people, 2).

The data flow of a word-count problem using the MapReduce functions (Map, Sort, Group and Reduce) in a cascade operations.

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:

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 occurrences of the same key. Finally, the Reduce function is applied in parallel to each group producing the collection of values as output as illustrated here:

Strategy to Solve MapReduce Problems

As mentioned earlier, after grouping all the intermediate data, the values of all occurrences of the same key are sorted and grouped together. As a result, after grouping, each key becomes unique in all intermediate data. Therefore, finding unique keys is the starting point to solving a typical MapReduce problem. Then the intermediate (key, value) pairs as the output of the Map function will be automatically found. The following three examples explain how to define keys and values in such problems:

Problem 1: Counting the number of occurrences of each word in a collection ofdocuments

Solution: unique ¯key : each word, intermediate ¯value : number of occurrences

Problem 2: Counting the number of occurrences of words having the same size, or the same number of letters, in a collection of documents

Solution: unique ¯key : each word, intermediate ¯value : size of the word

Problem 3: Counting the number of occurrences of anagrams in a collection of documents. Anagrams are words with the same set of letters but in a different order (e.g., the words ¯listen and ¯silent).

Solution: unique ¯key : alphabetically sorted sequence of letters for each word (e.g., eilnst),

intermediate ¯value : number of occurrences

MapReduce Actual Data and Control Flow

The main responsibility of the MapReduce framework is to efficiently run a user's program on a distributed computing system. Therefore, the MapReduce framework meticulously handles all partitioning, mapping, synchronization, communication, and scheduling details of such data flows . We summarize this in the following distinct steps:

Data partitioning The MapReduce library splits the input data (files), already stored in GFS, into M pieces that also correspond to the number of map tasks.

Computation partitioning This is implicitly handled (in the MapReduce framework) byobliging users to write their programs in the form of the Map and Reduce functions. therefore, the MapReduce library only generates copies of a user program (e.g., by a fork system call) containing the Map and the Reduce functions, distributes them, and starts them up on a number of available computation engines.

Determining the master and workers The MapReduce architecture is based on a master-worker model. Therefore, one of the copies of the user program becomes the master and the rest become workers. The master picks idle workers, and assigns the map and reduce tasks to them. A map/reduce worker is typically a computation engine such as a cluster node to run map/reduce tasks by executing Map/Reduce functions. Steps 4–7 describe the map workers.

Reading the input data (data distribution) Each map worker reads its corresponding portion of the input data, namely the input data split, and sends it to its Map function. Although a map worker may run more than one Map function, which means it has been assigned more than one input data split, each worker is usually assigned one input split only.

Map function Each Map function receives the input data split as a set of (key, value) pairs to process and produce the intermediated (key, value) pairs.

Combiner function This is an optional local function within the map worker which applies to intermediate (key, value) pairs. The user can invoke the Combiner function inside the user program. The Combiner function runs the same code written by users for the Reduce function as its functionality is identical to it. The Combiner function merges the local data of each map worker before sending it over the network to effectively reduce its communication costs. As mentioned in our discussion of logical data flow, the MapReduce framework sorts and groups the data before it is processed by the Reduce function. Similarly, the MapReduce framework will also sort and group the local data on each map worker if the user invokes the Combiner function.

Partitioning function As mentioned in our discussion of the MapReduce data flow, the intermediate (key, value) pairs with identical keys are grouped together because all values inside each group should be processed by only one Reduce function to generate the final result. However, in real implementations, since there are M map and R reduce tasks, intermediate (key, value) pairs with the same key might be produced by different map tasks, although they should be grouped and processed together by one Reduce function only.

Therefore, the intermediate (key, value) pairs produced by each map worker are partitioned into R regions, equal to the number of reduce tasks, by the Partitioning function to guarantee that all (key, value) pairs with identical keys are stored in the same region. As a result, since reduce worker i reads the data of region i of all map workers, all (key, value) pairs with the same key will be gathered by reduce worker I accordingly . To implement this technique, a Partitioning function could simply be a hash function (e.g., Hash(key) mod R) that forwards the data into particular regions. It is also worth noting that the locations of the buffered data in these Rpartitions are sent to the master for later forwarding of data to the reduce workers shows the data flow implementation of all data flow steps. The following are two networking steps:

Synchronization MapReduce applies a simple synchronization policy to coordinate map workers with reduce workers, in which the communication between them starts when all map tasks finish.

Communication Reduce worker i, already notified of the location of region i of all mapworkers, uses a remote procedure call to read the data from the respective region of all map workers. Since all reduce workers read the data from all map workers, all-to-all communication among all map and reduce workers, which incurs network congestion, occurs in the network. This issue is one of the major bottlenecks in increasing the performance of such systems . A data transfer module was proposed to schedule data transfers independently .Steps 10 and 11 correspond to the reduce worker domain:

Sorting and Grouping When the process of reading the input data is finalized by a reduce worker, the data is initially buffered in the local disk of the reduce worker. Then the reduce worker groups intermediate (key, value) pairs by sorting the data based on their keys, followed by grouping all occurrences of identical keys. Note that the buffered data is sorted and grouped because the number of unique keys produced by a map worker may be more than R regions in which more than one key exists in each region of a map worker .

Reduce function The reduce worker iterates over the grouped (key, value) pairs, and for each unique key, it sends the key and corresponding values to the Reduce function.

Then this function processes its input data and stores the output results in predetermined files in the user’s program.

Use of MapReduce partitioning function to link the Map and Reduce workers.

To better clarify the interrelated data control and control flow in the MapReduce framework, shows the exact order of processing control in such a system contrasting with dataflow. 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.

Control flow implementation of the MapReduce functionalities in Map workers and Reduce workers (running user programs) from input files to the output files under the control of the master user program. Courtesy of Yahoo! Pig Tutorial

Compute-Data Affinity

The MapReduce software framework was first proposed and implemented by Google. The first implementation was coded in C. The implementation takes advantage of GFS as the underlying layer. MapReduce could perfectly adapt itself to GFS. GFS is a distributed file system where files are divided into fixed-size blocks (chunks) and blocks are distributed and stored on cluster nodes. As stated earlier, the MapReduce library splits the input data (files) into fixed-size blocks, and ideally performs the Map function in parallel on each block. In this case, as GFS has already stored files as a set of blocks, the

MapReduce framework just needs to send a copy of the user's program containing the Map function to the nodes' already stored data blocks. This is the notion of sending computation toward data rather than sending data toward computation. Note that the default GFS block size is 64 MB which is identical to that of the MapReduce framework.

Twister and Iterative MapReduce

It is important to understand the performance of different runtimes and, in particular, to compare MPI and MapReduce . The two major sources of parallel overhead are load imbalance and communication (which is equivalent to synchronization overhead as communication synchronizes parallel units [threads or processes] in Categories 2 and 6). The communication overhead in MapReduce can be quite high, for two reasons:

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 update information. We can call the MPI flow ä flow and the MapReduce flow full data flow.

The same phenomenon is seen in all ¯classic parallel loosely synchronous applications which typically exhibit an iteration structure over compute phases followed by communication phases. We can address the performance issues with two important changes:

Stream information between steps without writing intermediate steps to disk.

Use long-running threads or processors to communicate the ä (between iterations) flow.

These changes will lead to major performance increases at the cost of poorer fault tolerance and ease to support dynamic changes such as the number of available nodes.

This concept has been investigated in several projects while the direct idea of using MPI for MapReduce applications is investigated in . The Twister programming paradigm and its implementation architecture at run time are illustrated whose performance results for K means are shown in Figure 6.8 [55,56], where Twister is much faster than traditional MapReduce. Twister distinguishes the static data which is never reloaded from the dynamic ä flow that is communicated.

Twister: An iterative MapReduce programming paradigm for repeated MapReduce execution

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 running 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 has a master/slave architecture containing a single NameNode as the master and a number of DataNodes as workers (slaves). To store a file in this architecture, HDFS splits the file into fixed-size blocks (e.g., 64 MB) and stores them on workers (DataNodes). The mapping of blocks to DataNodes is determined by the NameNode. The NameNode (master) also manages the file system’s metadata and namespace. In such systems, the namespace is the area maintaining the metadata, and metadata refers to all the information stored by a file system that is needed for overall management of all files. For example, NameNode in the metadata stores all information regarding the location of input splits/blocks in all DataNodes. Each DataNode, usually one per node in a cluster, manages the storage attached to the node. Each DataNode is responsible for storing and retrieving its file blocks .

HDFS Features: Distributed file systems have special requirements, such as performance, scalability, concurrency control, fault tolerance, and security requirements , to operate efficiently. However, because HDFS is not a general-purpose file system, as it only executes specific types of applications, it does not need all the requirements of a general distributed file system. For example, security has never been supported for HDFS systems. The following discussion highlights two important characteristics of HDFS to distinguish it from other generic distributed file systems .

HDFS Fault Tolerance: One of the main aspects of HDFS is its fault tolerance characteristic. Since Hadoop is designed to be deployed on low-cost hardware by default, a hardware failure in this system is considered to be common rather than an exception. Therefore, Hadoop considers the following issues to fulfill reliability requirements of the file system :

Block replication To reliably store data in HDFS, file blocks are replicated in this system. In other words, HDFS stores a file as a set of blocks and each block is replicated and distributed across the whole cluster. The replication factor is set by the user and is three by default.

Replica placement The placement of replicas is another factor to fulfill the desired fault tolerance in HDFS. Although storing replicas on different nodes (DataNodes) located in different racks across the whole cluster provides more reliability, it is sometimes ignored as the cost of communication between two nodes in different racks is relatively high in comparison with that of different nodes located in the same rack. Therefore, sometimes HDFS compromises its reliability to achieve lower communication costs. For example, for the default replication factor of three, HDFS stores one replica in the same node the original data is stored, one replica on a different node but in the same rack, and one replica on a different node in a different rack to provide three copies of the data .

Heartbeat and Blockreport messages Heartbeats and Blockreports are periodic messages sent to the NameNode by each DataNode in a cluster. Receipt of a Heartbeat implies that the DataNode is functioning properly, while each Blockreport contains a list of all blocks on a DataNode. The NameNode receives such messages because it is the sole decision maker of all replicas in the system.

HDFS High-Throughput Access to Large Data Sets (Files): Because HDFS is primarilydesigned for batch processing rather than interactive processing, data access throughput in HDFS is more important than latency. Also, because applications run on HDFS typically have large data sets, individual files are broken into large blocks (e.g., 64 MB) to allow HDFS to decrease the amount of metadata storage required per file. This provides two advantages: The list of blocks per file will shrink as the size of individual blocks increases, and by keeping large amounts of data sequentially within a block, HDFS provides fast streaming reads of data.

HDFS Operation: The control flow of HDFS operations such as write and read can properly highlight roles of the NameNode and DataNodes in the managing operations. In this section, the control flow of the main operations of HDFS on files is further described to manifest the interaction between the user, the NameNode, and the DataNodes in such systems .

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

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

The steamer then stores the block in the first allocated DataNode. Afterward, the block is forwarded to the second DataNode by the first DataNode. The process continues until all allocated DataNodes receive a replica of the first block from the previous DataNode.

Once this replication process is finalized, the same process starts for the second block and continues until all blocks of the file are stored and replicated on the file system.

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

Each TaskTracker node has a number of simultaneous execution slots, each executing either a map or a reduce task. Slots are defined as the number of simultaneous threads supported by CPUs of the TaskTracker node. For example, a TaskTracker node with N CPUs, each supporting M threads, has M * N simultaneous execution slots . It is worth noting that each data block is processed by one map task running on a single slot. Therefore, there is a one-to-one correspondence between map tasks in a TaskTracker and data blocks in the respective DataNode.

Running a Job in Hadoop

Three components contribute in running a job in this system: a user node, a Job Tracker, and several Task Trackers. The data flow starts by calling the run Job(conf) function inside a user program running on the user node, in which conf is an object containing some tuning parameters for the Map Reduce framework and HDFS. The run Job(conf) function and conf are comparable to the Map Reduce(Spec, &Results) function and Spec in the first implementation of Map Reduce by Google, depicts the data flow of running a MapReduce job in Hadoop . Data flow in running a Map Reduce job at various task trackers using the Hadoop library. Job Submission Each job is submitted from a user node to the JobTracker node that might be situated in a different node within the cluster through the followingprocedure:

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

The user node copies some resources, such as the job's JAR file, configuration file, and computed input splits, to the JobTracker's file system.

The user node submits the job to the JobTracker by calling the submitJob() function.

Task assignment The JobTracker creates one map task for each computed input split by the user node and assigns the map tasks to the execution slots of the TaskTrackers.

The JobTracker considers the localization of the data when assigning the map tasks to the TaskTrackers. The JobTracker also creates reduce tasks and assigns them to the TaskTrackers. The number of reduce tasks is predetermined by the user, and there is no locality consideration in assigning them.

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

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