1. Explain the Globus Toolkit Architecture (GT4)

The Globus Toolkit 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.

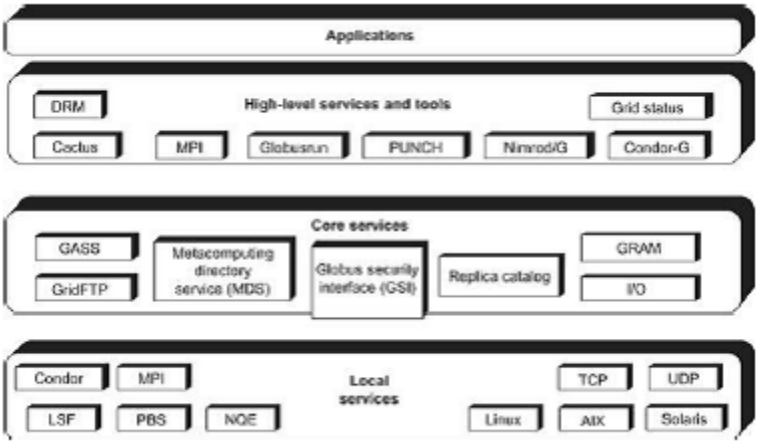


Figure: Globus Toolkit GT4 supports distributed and cluster computing services

The GT4 Library

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.

Globus Job Workflow

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, which applies GridFTP to bring in the necessary files.

GRAM invokes a local scheduler via a GRAM adapter and the SEG 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.

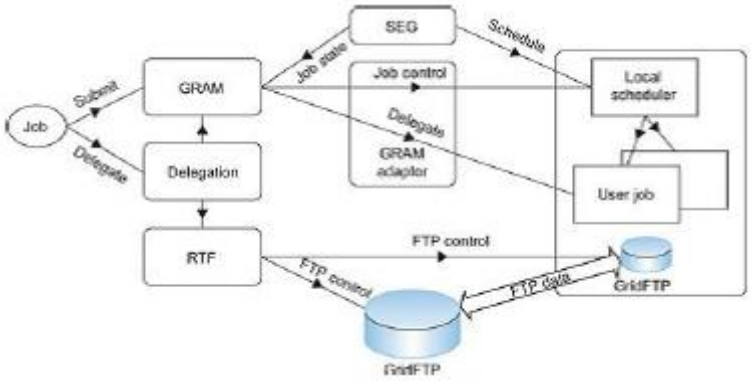


Figure: Globus job workflow among interactive functional modules.

Client-Globus Interactions

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.

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. Developers can use these services and libraries to build simple and complex systems quickly.

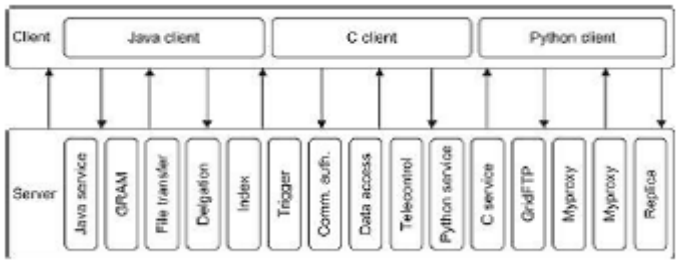
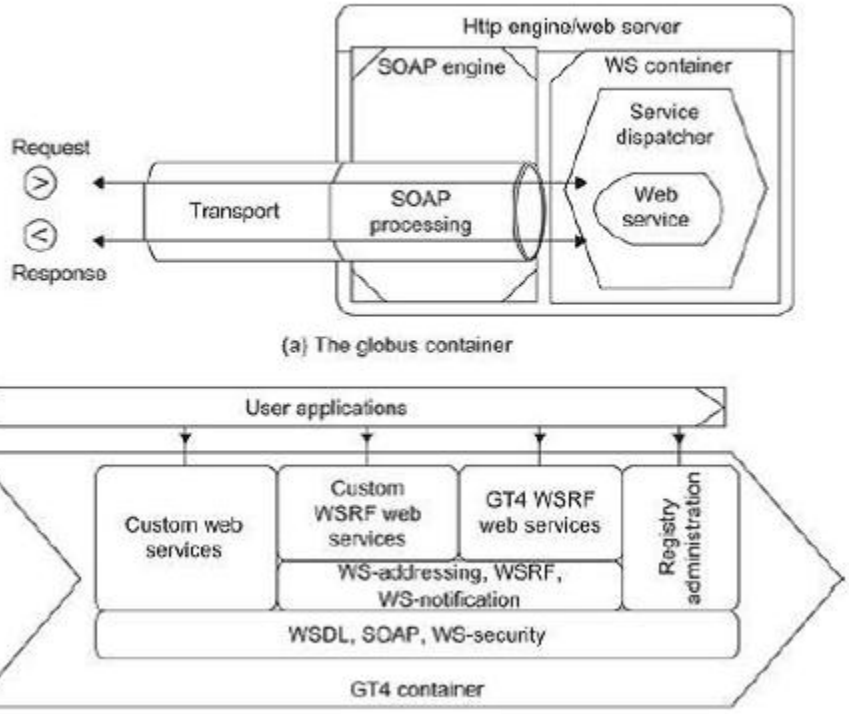


Figure: Client and GT4 server interactions; vertical boxes correspond to service programs and horizontal boxes represent the user codes.

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.

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.



2. Explain MapReduce Model in detail

The model is based on two distinct steps for an application:

• ***Map***: An initial ingestion and transformation step, in which individual input records can be processed in parallel.

• ***Reduce***: An aggregation or summarization step, in which all associated records must be processed together by a single entity.

The core concept of MapReduce in Hadoop is that input may be split into logical chunks, and each chunk may be initially processed independently, by a map task. The results of these individual processing chunks can be physically partitioned into distinct sets, which are then sorted. Each sorted chunk is passed to a reduce task.

A map task may run on any compute node in the cluster, and multiple map tasks may be running in parallel across the cluster. The map task is responsible for transforming the input records into key/value pairs. The output of all of the maps will be partitioned, and each partition will be sorted. There will be one partition for each reduce task. Each partition's sorted keys and the values associated with the keys are then processed by the reduce task. There may be multiple reduce tasks running in parallel on the cluster.

The application developer needs to provide only four items to the Hadoop framework: the class that will read the input records and transform them into one key/value pair per record, a map method, a reduce method, and a class that will transform the key/value pairs that the reduce method outputs into output records.

My first MapReduce application was a specialized web crawler. This crawler received as input large sets of media URLs that were to have their content fetched and processed. The media items were large, and fetching them had a significant cost in time and resources.

The job had several steps:
1. Ingest the URLs and their associated metadata.
2. Normalize the URLs.
3. Eliminate duplicate URLs.
4. Filter the URLs against a set of exclusion and inclusion filters.
5. Filter the URLs against a do not fetch list.
6. Filter the URLs against a recently seen set.
7. Fetch the URLs.
8. Fingerprint the content items.
9. Update the recently seen set.
10. Prepare the work list for the next application.

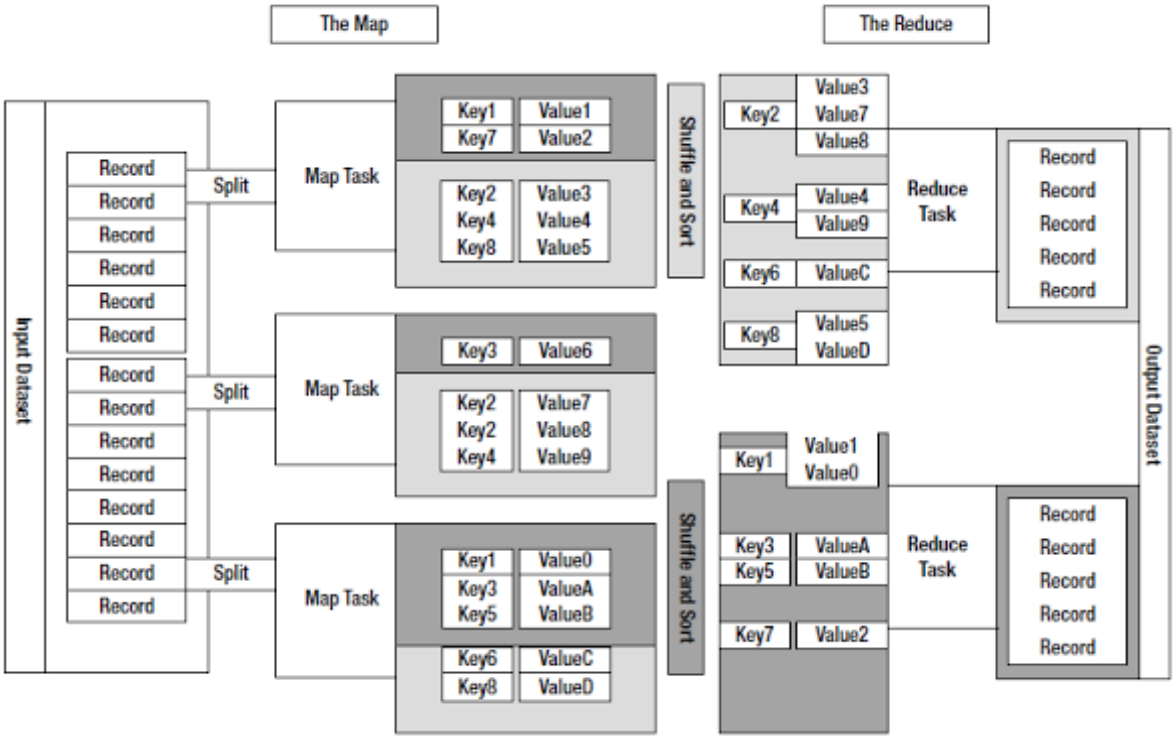


Figure: *The MapReduce model*

Introducing Hadoop

Hadoop is the Apache Software Foundation top-level project that holds the various Hadoop subprojects that graduated from the Apache Incubator. The Hadoop project provides and supports the development of open source software that supplies a framework for the development of highly scalable distributed computing applications. The Hadoop framework handles the processing details, leaving developers free to focus on application logic.

The introduction on the Hadoop project web page states:

The Apache Hadoop project develops open-source software for reliable, scalable, distributed computing, including:

***Hadoop** Core, our flagship sub-project, provides a distributed filesystem (HDFS) and support for the MapReduce distributed computing metaphor.*

HBase builds on Hadoop Core to provide a scalable, distributed database.

***Pig** is a high-level data-flow language and execution framework for parallel computation.It is built on top of Hadoop Core.*

***ZooKeeper** is a highly available and reliable coordination system. Distributed applications use ZooKeeper to store and mediate updates for critical shared state.*

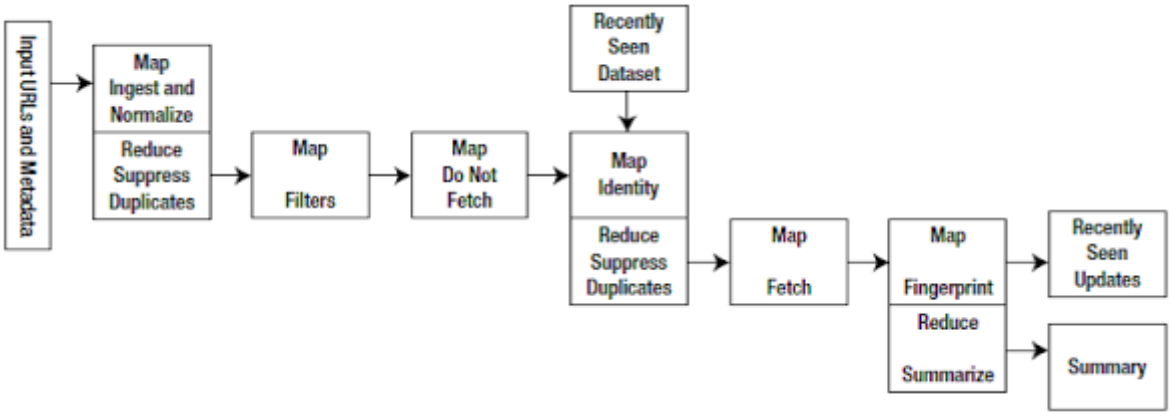
***Hive** is a data warehouse infrastructure built on Hadoop Core that provides data summarization, adhoc querying and analysis of datasets.*

The Hadoop Core project provides the basic services for building a cloud computing environment with commodity hardware, and the APIs for developing software that will run on that cloud.

The two fundamental pieces of Hadoop Core are the MapReduce framework, the cloud computing environment, and he Hadoop Distributed File System (HDFS).

The Hadoop Core MapReduce framework requires a shared file system. This shared file system does not need to be a system-level file system, as long as there is a distributed file system plug-in available to the framework.

The Hadoop Core framework comes with plug-ins for HDFS, CloudStore, and S3. Users are also free to use any distributed file system that is visible as a system-mounted file system, such as Network File System (NFS), Global File System (GFS), or Lustre.



The Hadoop Distributed File System (HDFS)MapReduce environment provides the user with a sophisticated framework to manage the execution of map and reduce tasks across a cluster of machines.

The user is required to tell the framework the following:

- The location(s) in the distributed file system of the job input
- The location(s) in the distributed file system for the job output
- The input format
- The output format
- The class containing the map function
- Optionally. the class containing the reduce function

• The JAR file(s) containing the map and reduce functions and any support classes

The final output will be moved to the output directory, and the job status will be reported to the user.MapReduce is oriented around key/value pairs. The framework will convert each record of input into a key/value pair, and each pair will be input to the map function once. The map output is a set of key/value pairs—nominally one pair that is the transformed input pair. The map output pairs are grouped and sorted by key. The reduce function is called one time for each key, in sort sequence, with the key and the set of values that share that key. The reduce method may output an arbitrary number of key/value pairs, which are written to the output files in the job output directory. If the reduce output keys are unchanged from the reduce input keys, the final output will be sorted. The framework provides two processes that handle the management of MapReduce jobs:

• TaskTracker manages the execution of individual map and reduce tasks on a compute node in the cluster.

• JobTracker accepts job submissions, provides job monitoring and control, and manages the distribution of tasks to the TaskTracker nodes.

The JobTracker is a single point of failure, and the JobTracker will work around the failure of individual TaskTracker processes.

The Hadoop Distributed File System

HDFS is a file system that is designed for use for MapReduce jobs that read input in large chunks of input, process it, and write potentially large chunks of output. HDFS does not handle random access particularly well. For reliability, file data is simply mirrored to multiple storage nodes. This is referred to as *replication* in the Hadoop community. As long as at least one replica of a data chunk is available, the consumer of that data will not know of storage server failures.

HDFS services are provided by two processes:

• NameNode handles management of the file system metadata, and provides management and control services.

• DataNode provides block storage and retrieval services.

There will be one NameNode process in an HDFS file system, and this is a single point of failure. Hadoop Core provides recovery and automatic backup of the NameNode, but no hot failover services. There will be multiple DataNode processes within the cluster, with typically one DataNode process per storage node in a cluster.

3. Explain Map & Reduce function?

A Simple Map Function: IdentityMapper

The Hadoop framework provides a very simple map function, called IdentityMapper. It is used in jobs that only need to reduce the input, and not transform the raw input. All map functions must implement the Mapper interface, which guarantees that the map function will always be called with a key. The key is an instance of a WritableComparable object, a value that is an instance of a Writable object, an output object, and a reporter.

IdentityMapper.java

```
package org.apache.hadoop.mapred.lib;
import java.io.IOException;
import org.apache.hadoop.mapred.Mapper;
```

```
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapred.MapReduceBase;
/** Implements the identity function, mapping inputs directly to outputs. */
public class IdentityMapper<K, V>
extends MapReduceBase implements Mapper<K, V, K, V> {
/** The identify function. Input key/value pair is written directly to
* output.*/
public void map(K key, V val,
OutputCollector<K, V> output, Reporter reporter)
throws IOException {
output.collect(key, val);
}
}
```

A Simple Reduce Function: IdentityReducer

The Hadoop framework calls the reduce function one time for each unique key. The framework provides the key and the set of values that share that key.

IdentityReducer.java

```
package org.apache.hadoop.mapred.lib;
import java.io.IOException;
import java.util.Iterator;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapred.MapReduceBase;
/** Performs no reduction, writing all input values directly to the output. */
public class IdentityReducer<K, V>
extends MapReduceBase implements Reducer<K, V, K, V> {
```

Chapter 2 THE BASICS OF A MAPREDUCE JOB **35**

```
/** Writes all keys and values directly to output. */
public void reduce(K key, Iterator<V> values,
OutputCollector<K, V> output, Reporter reporter)
throws IOException {
while (values.hasNext()) {
output.collect(key, values.next());
}
}
```

If you require the output of your job to be sorted, the reducer function must pass the key objects to the output.collect() method unchanged. The reduce phase is, however, free to output any number of records, including zero records, with the same key and different values.

4. Explain HDFS Concepts in detail?

Blocks

A disk has a block size, which is the minimum amount of data that it can read or write. Filesystem blocks are typically a few kilobytes in size, while disk blocks are normally 512 bytes. HDFShas the concept of a block, but it is a much larger unit—64 MB by default.File0sin HDFS are broken into block-sized chunks, which are stored as independent units. Unlike a filesystem for a single disk, a file in HDFS that is smaller than a single block does not occupy a full block's worth of underlying storage.

Simplicity is something to strive for all in all systems, but is especially important for a distributed system in which the failure modes are so varied. The storage subsystem deals with blocks, simplifying storage management and eliminating metadata concerns

Namenodes and Datanodes

An HDFS cluster has two types of node operating in a master-worker pattern: a *namenode* (the master) and a number of *datanodes* (workers). The namenode manages the filesystem namespace. It maintains the filesystem tree and the metadata for all the files and directories in the tree.

The namenode also knows the datanodes on which all the blocks for a given file are located, however, it does not store block locations persistently, since this information is reconstructed from datanodes when the system starts.

A *client* accesses the filesystem on behalf of the user by communicating with the namenode and datanodes. Datanodes are the workhorses of the filesystem. Hadoop can be configured so that the namenode writes its persistent state to multiple filesystems. These writes are synchronous and atomic. The usual configuration choice is to write to local disk as well as a remote NFS mount.

It is also possible to run a *secondary namenode*, which despite its name does not act as a namenode. Its main role is to periodically merge the namespace image with the edit log to prevent the edit log from becoming too large. The secondary namenode usually runs on a separate physical machine, since it requires plenty of CPU and as much memory as the namenode to perform the merge. It keeps a copy of the merged namespace image, which can be used in the event of the namenode failing.

HDFS Federation

The namenode keeps a reference to every file and block in the filesystem in memory, which means that on very large clusters with many files, memory becomes the limiting factor for scaling.

HDFS Federation, introduced in the 0.23 release series, allows a cluster to scale by adding namenodes, each of which manages a portion of the filesystem namespace. For example, one namenode might manage all the files rooted under */user*, say, and a second

Namenode might handle files under */share*.Under federation, each namenode manages a *namespace volume*, which is made up of the metadata for the namespace, and a *block pool* containing all the blocks for the files in the namespace. Namespace volumes are independent of each other, which means namenodes do not communicate with one another, and furthermore the failure of one namenode does not affect the availability of the namespaces managed by other namenodes.

Block pool storage is *not* partitioned, however, so datanodes register with each namenode in the cluster and store blocks from multiple block pools.

HDFS High-Availability

The combination of replicating namenode metadata on multiple filesystems, and using

the secondary namenode to create checkpoints protects against data loss, but does not provide high-availability of the filesystem. The namenode is still a *single point of failure* (SPOF), since if it did fail, all clients—including MapReduce jobs—would be unable to read, write, or list files, because the namenode is the sole repository of the metadata and the file-to-block mapping. In such an event the whole Hadoop system would effectively be out of service until a new namenode could be brought online. In the event of the failure of the active namenode, the standby takes over its duties to continue servicing client requests without a significant interruption.

A few architectural changes are needed to allow this to happen:

• The namenodes must use highly-available shared storage to share the edit log.

When a standby namenode comes up it reads up to the end of the shared edit log to synchronize its state with the active namenode, and then continues to read new entries as they are written by the active namenode.

• Datanodes must send block reports to both namenodes since the block mappings are stored in a namenode's memory, and not on disk.

• Clients must be configured to handle namenode failover, which uses a mechanism that is transparent to users.

If the active namenode fails, then the standby can take over very quickly since it has the latest state available in memory: both the latest edit log entries, and an up-to-date block mapping. The actual observed failover time will be longer in practice (around a minute or so), since the system needs to be conservative in deciding that the active namenode has failed.

Failover and fencing

The transition from the active namenode to the standby is managed by a new entity in the system called the *failover controller*. Failover controllers are pluggable, but the first implementation uses ZooKeeper to ensure that only one namenode is active. Each namenode runs a lightweight failover controller process whose job it is to monitor its namenode for failures and trigger a failover should a namenode fail.

Failover may also be initiated manually by an administrator, in the case of routine maintenance, for example.

In the case of an ungraceful failover, however, it is impossible to be sure that the failed namenode has stopped running. The HA implementation goes to great lengths to ensure that the previously active namenode is prevented from doing any damage and causing corruption—a method known as *fencing*. The system employs a range of fencing mechanisms, including killing the namenode's process, revoking its access to the shared storage directory, and disabling its network port via a remote management command. As a last resort, the previously active namenode can be fenced with a technique rather graphically known as *STONITH*, or "shoot the other node in the head", which uses a specialized power distribution unit to forcibly power down the host machine. Client failover is handled transparently by the client library. The simplest implementation uses client-side configuration to control failover. The HDFS URI uses a logical hostname which is mapped to a pair of namenode addresses, and the client library tries each namenode address until the operation succeeds.

5. Explain Anatomy of a File Read?

The client opens the file it wishes to read by calling open () on the FileSystem object, which for HDFS is an instance of DistributedFileSystem. DistributedFileSystem calls the namenode, using RPC, to determine the locations of the blocks for the first few blocks in the file. The namenode returns the addresses of the datanodes that have a copy of that block.

If the client is itself a datanode ,then it will read from the local datanode, if it hosts a copy of the block .The DistributedFileSystem returns an FSDataInputStream to the client for it to read data from. FSDataInputStream in turn wraps a DFSInputStream, which manages the datanode and namenode I/O.

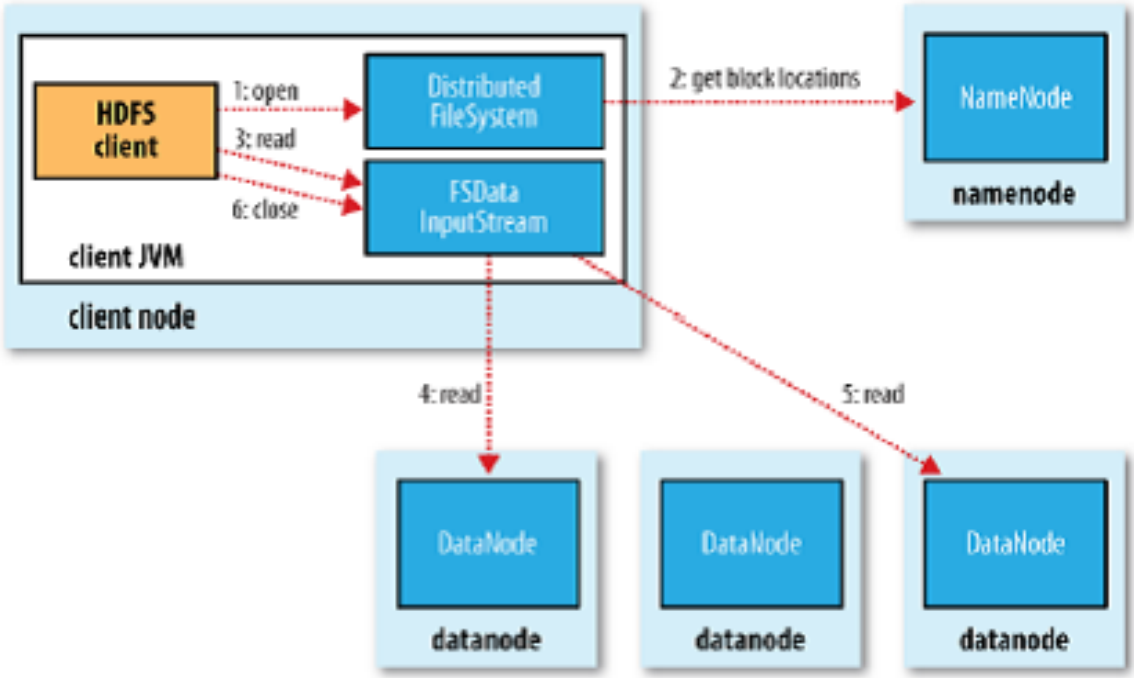


Figure: *A client reading data from HDFS*

The client then calls read () on the stream. DFSInputStream, which has stored the datanode addresses for the first few blocks in the file, then connects to the first (closest) datanode for the first block in the file. Data is streamed from the datanode back to the client, which calls read () repeatedly on the stream. When the end of the block is reached, DFSInputStream will close the connection to the datanode, then find the best datanode for the next block. This happens transparently to the client, which from its point of view is just reading a continuous stream.

Blocks are read in order with the DFSInputStream opening new connections to datanodes as the client reads through the stream. It will also call the namenode to retrieve the datanode locations for the next batch of blocks as needed. When the client has finished reading, it calls close () on the FSDataInputStream .

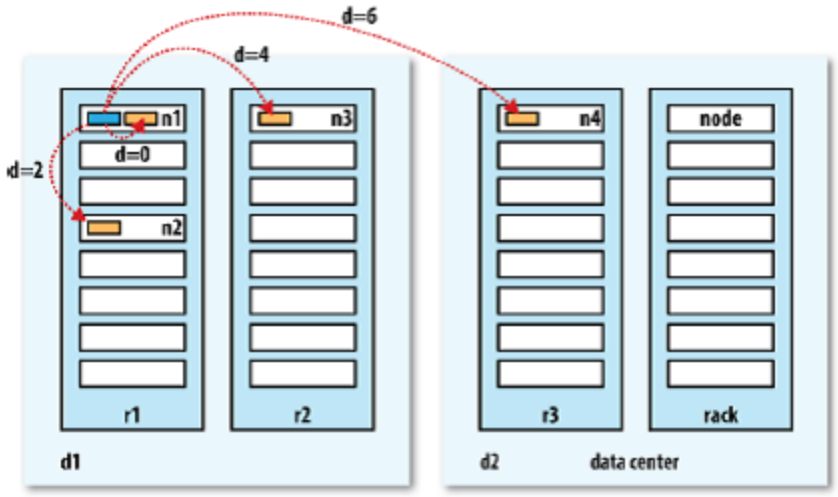


Figure: *Network distance in Hadoop*

During reading, if the DFSInputStream encounters an error while communicating with a datanode, then it will try the next closest one for that block. It will also remember datanodes that have failed so that it doesn't needlessly retry them for later blocks. The DFSInputStream also verifies checksums for the data transferred to it from the datanode.

If a corrupted block is found, it is reported to the namenode before the DFSInput Stream attempts to read a replica of the block from another datanode.One important aspect of this design is that the client contacts datanodes directly to retrieve data and is guided by the namenode to the best datanode for each block. This design allows HDFS to scale to a large number of concurrent clients, since the data traffic is spread across all the datanodes in the cluster.

6. Explain Anatomy of a File write?

The client creates the file by calling create () on DistributedFileSystem. DistributedFileSystem makes an RPC call to the namenode to create a new file in the filesystem's namespace, with no blocks associated with it (step 2). The namenode performs various checks to make sure the file doesn't already exist, and that the client has the right permissions to create the file. If these checks pass, the namenode makes a record of the new file.

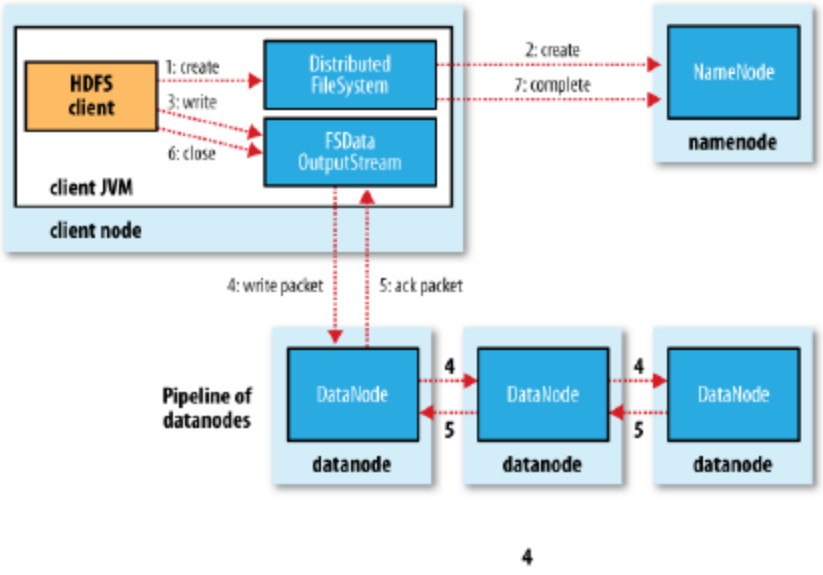


Figure: *A client writing data to HDFS*

The DistributedFileSystem returns an FSDataOutputStream for the client to start writing data to. Just as in the read case, FSDataOutputStream wraps a DFSOutput Stream, which handles communication with the datanodes and namenode.As the client writes data (step 3), DFSOutputStream splits it into packets, which it writes to an internal queue, called the *data queue*. The data queue is consumed by the Data Streamer, whose responsibility it is to ask the namenode to allocate new blocks by picking a list of suitable datanodes to store the replicas. The list of datanodes forms a pipeline—we'll assume the replication level is three, so there are three nodes in the pipeline. The DataStreamer streams the packets to the first datanode in the pipeline, which stores the packet and forwards it to the second datanode in the pipeline. Similarly, the second datanode stores the packet and forwards it to the third (and last) datanode in the pipeline (step 4).DFSOutputStream also maintains an internal queue of packets that are waiting to be acknowledged by datanodes, called the *ack queue*. A packet is removed from the ack queue only when it has been acknowledged by all the datanodes in the pipeline (step 5).If a datanode fails while data is being written to it, then the following actions are taken, which are transparent to the client writing the data.

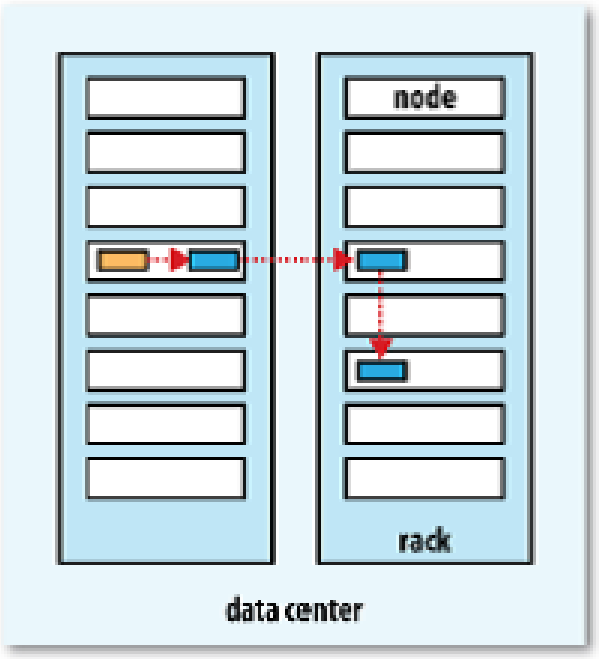


Figure: *A typical replica pipeline*

First the pipeline is closed, and any packets in the ack queue are added to the front of the data queue so that datanodes that are downstream from the failed node will not miss any packets. The current block 0on the good datanodes is given a new identity, which is communicated to the namenode, so that the partial block on the failed datanode will be deleted if the failed. Datanode recovers later on. The failed datanode is removed from the pipeline and the remainder of the block's data is written to the two good datanodes in the pipeline. The namenode notices that the block is under-replicated, and it arranges for a further replica to be created on another node. Subsequent blocks are then treated as normal. It's possible, but unlikely, that multiple datanodes fail while a block is being written. As long as dfs.replication.min replicas (default one) are written, the write will succeed, and the block will be asynchronously replicated across the cluster until its target replication factor is reached.

When the client has finished writing data, it calls close () on the stream (step 6). This action flushes all the remaining packets to the datanode pipeline and waits for acknowledgments before contacting the namenode to signal that the file is complete (step 7). The namenode already knows which blocks the file is made up.