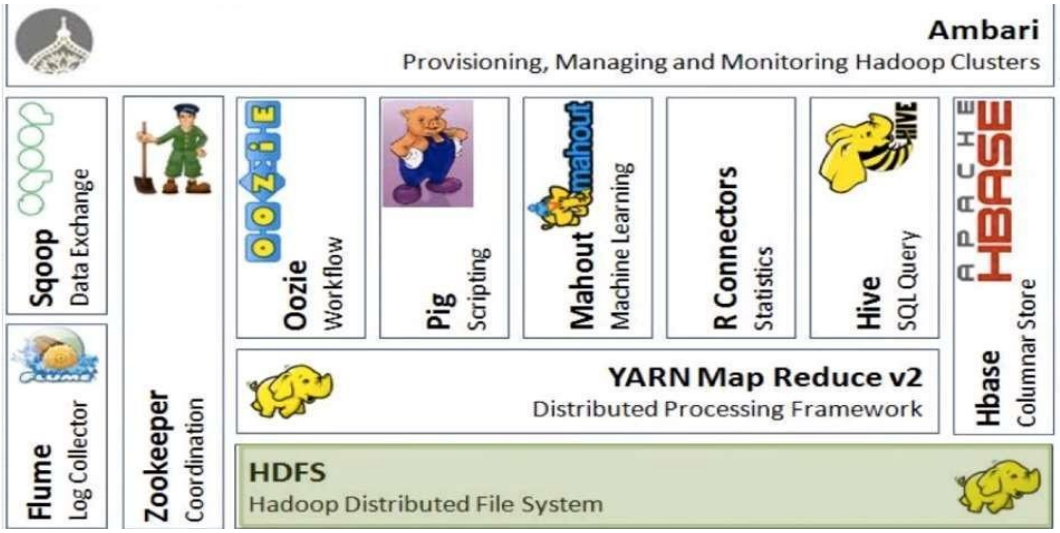
Illustrate the Hadoop ecosystem and describe its components using a labeled diagram (10 Marks)

1

Hadoop Ecosystem is neither a programming language nor a service, it is a platform or framework which solves big data problems. You can consider it as a suite which encompasses a number of services (ingesting, storing, analyzing and maintaining) inside it. The figure below describes the typical view of Hadoop Ecosystem:



The components of Hadoop are listed as follows:

Common – a set of components and interfaces for filesystems and I/O.

Avro – a serialization system for RPC and persistent data storage.

MapReduce – a distributed data processing model.

YARN - Yet Another Resource Negotiator

HDFS – a distributed filesystem running on large clusters of machines.

Pig – a data flow language and execution environment for large datasets.

Hive – a distributed data warehouse providing SQL-like query language.

HBase – a distributed, column-oriented database.

Mahout, Spark MLlib - Machine Learning

Apache Drill -SQL on Hadoop

ZooKeeper – a distributed, highly available coordination service.

Sqoop – a tool for efficiently moving data between relational DB and HDFS.

Oozie - Job Scheduling

Flume- Data Ingesting Services

Solr & Lucene - Searching & Indexing

Ambari - Provision, Monitor and Maintain cluster

.

2 **Compare and contrast MapReduce with a traditional RDBMS. (10 Marks)**

Difference between MapReduce and RDBMS

#	MapReduce	RDBMS
1	Good fit for problems that analyze the whole data set in a batch fashion	RDBMS is good for point queries or updates, where the data set has been ordered to deliver low latency retrieval
2	It suits well for applications where the data is written once and read many times	Relational database is good for data that are continuously updated
3	Works on semi-structured or unstructured data	Operates on structured data
4	Ex: spreadsheets, images, text etc..	Ex: database tables, XML docs etc..
5	It is designed to interpret the data at processing time (Schema on Read)	It is designed to interpret the data run time (Schema on Write)
6	Normalization creates a problem in Hadoop because, it reading a record is non-local operation, instead Hadoop makes it possible to perform streaming reads and writes	RDBMS data is often normalized to avoid redundancy and to retain integrity.
7	MapReduce can process the data in parallel	Parallel processing is not true for SQL RDBMS queries

3 **Highlight the differences between MapReduce, Grid Computing and Volunteer Computing (10 Marks)**

Grid Computing MPI	MapReduce
Works well for predominantly compute intensive jobs, but it becomes a problem where nodes access larger data volumes since the network bandwidth computing node becomes idle	Hadoop tries to collocate with the data and compute nodes so that the data access is fast because of data locality.
MPI programs have to explicitly manage their own checkpoint and recovery which gives more control to the programmer but makes them more difficult	In MapReduce, since the implementation detects the failed task and re-schedules replacement on the machine.
MPI is a shared architecture	MapReduce is a share nothing architecture
Gives great control to the programmer, it also requires that they explicitly handle the mechanics of data flow	Processing in Hadoop happens only at higher level the program thinks in terms of data model since the data flow is implicit

MapReduce	Volunteer Computing
A programming model and processing technique for distributed computing based on the map and reduce functions.	A computing paradigm where volunteers donate their idle computer resources to contribute to large-scale computational tasks.
Runs in a controlled, homogeneous data center or cloud infrastructure.	Runs on a heterogeneous, loosely connected network of volunteer devices (e.g., PCs, phones).
Data is stored in distributed file systems (like HDFS); tightly managed.	Data is distributed over the internet to volunteer machines with less control over security.
Built-in mechanisms handle node failures automatically.	Faults are expected and common; systems must be designed to tolerate unreliable nodes.
Centralized scheduler allocates tasks based on proximity to data and system load.	Tasks are sent to volunteer clients, often with redundancy to ensure correctness.
Secure and private, used in enterprise and cloud environments.	Less secure; data may be exposed to untrusted volunteers.
High and predictable; optimized for throughput and scalability.	Variable performance depending on volunteer availability and reliability.
Hadoop, Apache Spark (in MapReduce mode)	SETI@home, BOINC, Folding@home

4 **List any four major Hadoop releases and mention one key feature introduced in each. (10 Marks)**

There are a few active release series. The 1.x release series is a continuation of the 0.20 release series, and contains the most stable versions of Hadoop currently available. This series includes secure Kerberos authentication, which prevents unauthorized access to Hadoop data. Almost all production clusters use these releases, or derived versions (such as commercial distributions).

The 0.22 and 0.23 release series are currently marked as alpha releases (as of early 2012), but this is likely to change by the time you read this as they get more real-world testing and become more stable (consult the Apache Hadoop releases page for the latest status). 0.23 includes several major new features:

A new MapReduce runtime, called **MapReduce 2**, implemented on a new system called YARN (Yet Another Resource Negotiator), which is a general resource management system for running distributed applications. MapReduce 2 replaces the classic runtime in previous releases. It is described in more depth in "YARN".

HDFS federation, which partitions the HDFS namespace across multiple namenodes to support clusters with very large numbers of files.

HDFS high-availability, which removes the namenode as a single point of failure by supporting standby namenodes for failover.

The following figure shows the configuration of Hadoop 1.0 and Hadoop 2.0.

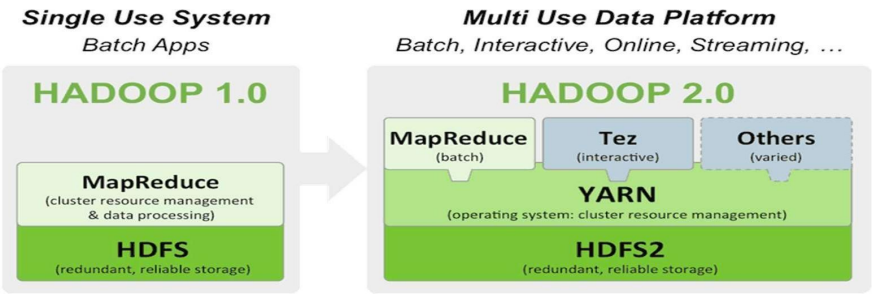


Table covers features in HDFS and MapReduce. Other projects in the Hadoop ecosystem are continually evolving too, and picking a combination of components that work well together can be a challenge.

Feature	1.x	0.22	0.23
Secure authentication	Yes	No	Yes
Old configuration names	Yes	Deprecated	Deprecated
New configuration names	No	Yes	Yes
Old MapReduce API	Yes	Deprecated	Deprecated
New MapReduce API	Partial	Yes	Yes
MapReduce 1 runtime (Classic)	Yes	Yes	No
MapReduce 2 runtime (YARN)	No	No	Yes
HDFS federation	No	No	Yes
HDFS high-availability	No	No	Planned

5 **Read and print files from the Hadoop filesystem using `URLStreamHandler` (10 Marks)**

Displaying files from a Hadoop filesystem on standard output using aURLStreamHandler

```
public class URLCat {
static {
URL.setURLStreamHandlerFactory(new FsUrlStreamHandlerFactory());
}
public static void main(String[] args) throws Exception {
InputStream in = null;
try {
in = new URL(args[0]).openStream();
IOUtils.copyBytes(in, System.out, 4096, false);
} finally {
IOUtils.closeStream(in);
}
}
}
```

6 **Read and print a file from the Hadoop filesystem twice using the `seek` method(). (10 Marks)**

Displaying files from a Hadoop filesystem on standard output twice, by using seek

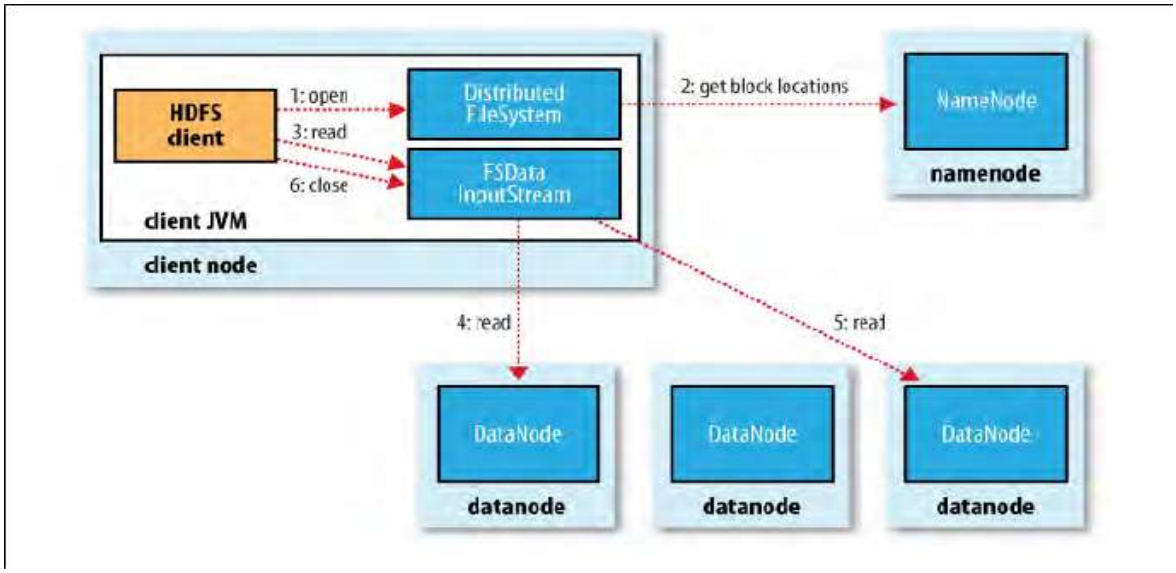
```
public class FileSystemDoubleCat {
public static void main(String[] args) throws Exception {
```

```
String uri = args[0];
Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(URI.create(uri), conf);
FSDataInputStream in = null;
try {
in = fs.open(new Path(uri));
IOUtils.copyBytes(in, System.out, 4096, false);
in.seek(0); // go back to the start of the file
IOUtils.copyBytes(in, System.out, 4096, false);
} finally {
IOUtils.closeStream(in);
}
}
}
```

7 **Describe in detail the process involved in reading a file from the Hadoop Distributed File System. (10 Marks)**

Anatomy of a File Read

To get an idea of how data flows between the client interacting with HDFS, the name-node and the datanodes, consider the below Figure, which shows the main sequence of events when reading a file.



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 (step 2). For each block, the namenode returns the addresses of the datanodes that have a copy of that block. Furthermore, the datanodes are sorted according to their proximity to the client If the client is itself a datanode (in the case of a MapReduce task, for instance), then it will read from the local datanode, if it hosts a copy of the block.

The DistributedFileSystem returns an FSDataInputStream (an input stream that sup-ports file seeks) to the client for it to read data from. FSDataInputStream in turn wraps a DFSInputStream, which manages the datanode and namenode I/O.

The client then calls read() on the stream (step 3). 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 (step 4).

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 (step 5). 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 (step 6).

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. The namenode meanwhile merely has to service block location requests (which it stores in memory, making them very efficient) and does not, for example, serve data, which would quickly become a bottleneck as the number of clients grew.

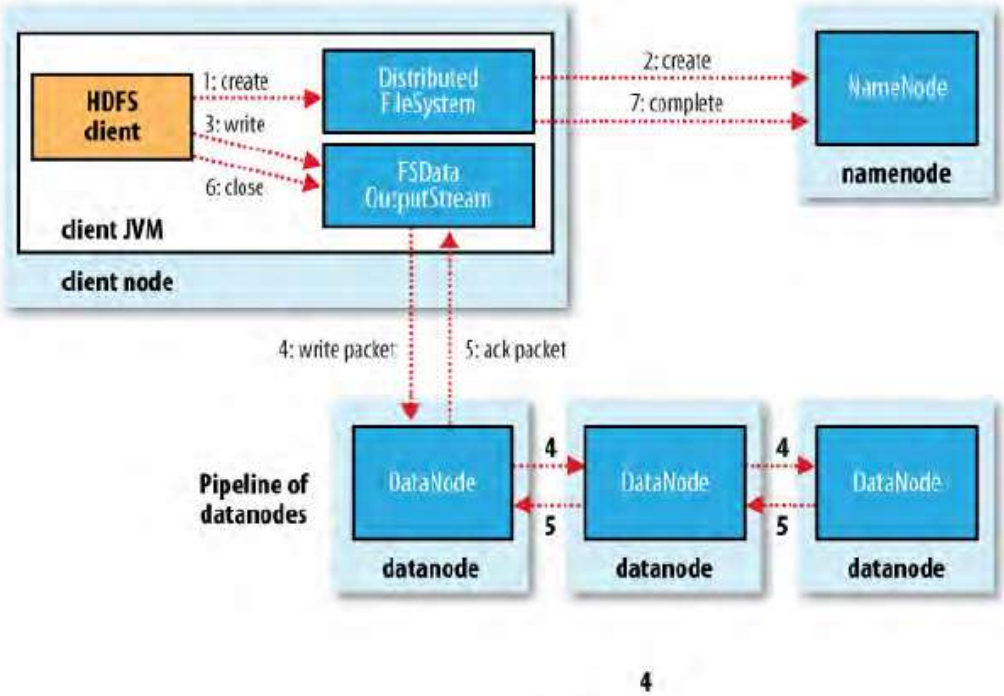
8 **Describe in detail the process involved in writing a file to the Hadoop Distributed File System.** **(10 Marks)**

Anatomy of a File Write

Next we'll look at how files are written to HDFS. Although quite detailed, it is instructive to understand the data flow since it clarifies HDFS's coherency model.

The case we're going to consider is the case of creating a new file, writing data to it, then closing the file.
The client creates the file by calling create() on DistributedFileSystem (step 1). 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 name-node 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; otherwise, file creation fails and the client is thrown an IOException. 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. Simi-larly, the second datanode stores the packet and forwards it to the third (and last) datanode in the pipeline (step 4).



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 ackqueue 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. 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 on the good datanodes is given a new identity, which is communicated to the name- node, 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 rep-lication factor is reached (dfs.replication, which defaults to three). 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 ac-knowledgments before contacting the namenode to signal that the file is complete (step7). The namenode already knows which blocks the file is made up of (via DataStreamer asking for block allocations), so it only has to wait for blocks to be minimally replicated before returning successfully.

9 **Elaborate on the following terms:** **(10 Marks)**

a) Failover mechanism

The architectural changes are needed to replace active name node with standby name node:

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 (in a few tens of seconds) 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.

In the unlikely event of the standby being down when the active fails, the administrator can still start the standby from cold. This is no worse than the non- HA case, and from an operational point of view it's an improvement, since the process is a standard operational procedure built into Hadoop.

b) Fencing strategy

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 (using a simple heartbeating mechanism) and trigger a failover should a namenode fail.

Failover may also be initiated manually by an adminstrator, in the case of routine maintenance, for example. This is known as a *graceful failover*, since the failover controller arranges an orderly transition for both namenodes to switch roles.

In the case of an ungraceful failover, however, it is impossible to be sure that the failed namenode has stopped running. For example, a slow network or a network partition can trigger a failover transition, even though the previously active namenode is still running, and thinks it is still the active namenode.

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 (typically by using a vendor-specific NFS command), 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 (in the configuration file), and the client library tries each namenode address until the operation succeeds.

We're going to have a look at HDFS by interacting with it from the command line. There are many other interfaces to HDFS, but the command line is one of the simplest and, to many developers, the most familiar.

There are two properties that we set in the pseudo-distributed configuration that deserve further explanation. The first is fs.default.name, set to *hdfs://localhost/*, which is used to set a default filesystem for Hadoop. Filesystems are specified by a URI, and here we have used an hdfs URI to configure Hadoop to use HDFS by default. The HDFS daemons will use this property to determine the host and port for the HDFS namenode.

We will be running it on localhost, on the default HDFS port, 8020. And HDFS clients will use this property to work out where the namenode is running so they can connect to it. We set the second property, dfs.replication, to 1 so that HDFS doesn't replicate filesystem blocks by the default factor of three. When running with a single datanode, HDFS can't replicate blocks to three datanodes, so it would perpetually warn about blocks being under-replicated. This setting solves that problem.

10 **Write short notes on the following:** **(10 Marks)**

a) Network topology in Hadoop

Network Topology and Hadoop

What does it mean for two nodes in a local network to be "close" to each other? In the context of high-volume data processing, the limiting factor is the rate at which we can transfer data between nodes—bandwidth is a scarce commodity. The idea is to use the bandwidth between two nodes as a measure of distance.

Rather than measuring bandwidth between nodes, which can be difficult to do in prac-tice (it requires a quiet cluster, and the number of pairs of nodes in a cluster grows as the square of the number of nodes), Hadoop takes a simple approach in which the network is represented as a tree and the distance between two nodes is the sum of their distances to their closest common ancestor. Levels in the tree are not predefined, but it is common to have levels that correspond to the data center, the rack, and the node that a process is running on. The idea is that the bandwidth available for each of the following scenarios becomes progressively less:

• Processes on the same node
• Different nodes on the same rack
• Nodes on different racks in the same data center
• Nodes in different data centers7

For example, imagine a node n1 on rack r1 in data center d1. This can be represented as /d1/r1/n1. Using this notation, here are the distances for the four scenarios:

• distance(/d1/r1/n1, /d1/r1/n1) = 0 (processes on the same node)
• distance(/d1/r1/n1, /d1/r1/n2) = 2 (different nodes on the same rack)
• distance(/d1/r1/n1, /d1/r2/n3) = 4 (nodes on different racks in the same data center)
• distance(/d1/r1/n1, /d2/r3/n4) = 6 (nodes in different data centers)

b) Replica placement and replacement strategy

Replica Placement

How does the namenode choose which datanodes to store replicas on? There's a trade-off between reliability and write bandwidth and read bandwidth here. For example, placing all replicas on a single node incurs the lowest write bandwidth penalty since the replication pipeline runs on a single node, but this offers no real redundancy (if the node fails, the data for that block is lost). Also, the read bandwidth is high for off-rack reads. At the other extreme, placing replicas in different data centers may maximize redundancy, but at the cost of bandwidth. Even in the same data center (which is what all Hadoop clusters to date have run in), there are a variety of placement strategies. Indeed, Hadoop changed its placement strategy in release 0.17.0 to one that helps keep a fairly even distribution of blocks across the cluster. (See "balancer" on page 348 for details on keeping a cluster balanced.) And from 0.21.0, block placement policies are pluggable. Hadoop's default strategy is to place the first replica on the same node as the client (for clients running outside the cluster, a node is chosen at random, although the system tries not to pick nodes that are too full or too busy). The second replica is placed on a different rack from the first (off-rack), chosen at random. The third replica is placed on the same rack as the second, but on a different node chosen at random. Further replicas are placed on random nodes on the cluster, although the system tries to avoid placing too many replicas on the same rack. Once the replica locations have been chosen, a pipeline is built, taking network topology into account.