Distributed File Systems

Chien-Min Wang

Institute of Information Science

Academia Sinica

Contents

Lecture 4
Distributed File Systems for Clouds

Outline

- Google File System (GFS)
- Hadoop Distributed File System (HDFS)
- HDFS Laboratory

GFS Introduction

- Design constraints
  - Component failures are the norm
    - 1000s of components
    - Bugs, human errors, failures of memory, disk, connectors, networking, and power supplies
    - Monitoring, error detection, fault tolerance, automatic recovery
  - Files are huge by traditional standards
    - Multi-GB files are common
    - Billions of objects

GFS Introduction

- Design constraints
  - Most modifications are appends
    - Random writes are practically nonexistent
    - Many files are written once, and read sequentially
  - Two types of reads
    - Large streaming reads
    - Small random reads (in the forward direction)
  - Sustained bandwidth more important than latency
  - File system APIs are open to changes

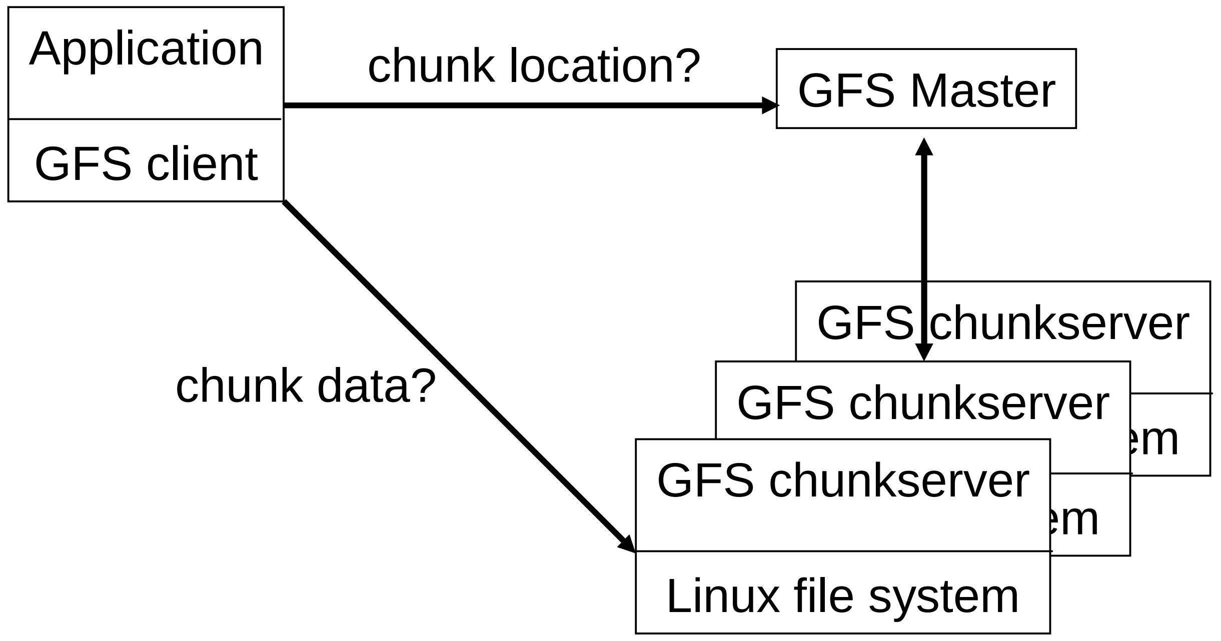
Interface Design

- Not POSIX compliant
- Additional operations
  - Snapshot
  - Record append

Architectural Design

- A GFS cluster
  - A single master
  - Multiple chunkservers per master
    - Accessed by multiple clients
  - Running on commodity Linux machines
- A file
  - Represented as fixed-sized chunks
    - Labeled with 64-bit unique global IDs
    - Stored at chunkservers
    - 3-way Mirrored across chunkservers

Architectural Design



Architectural Design

- Master server
  - Maintains all metadata
    - Name space, access control, file-to-chunk mappings, garbage collection, chunk migration
- GPS clients
  - Consult master for metadata
  - Access data from chunkservers
  - Does not go through VFS
  - No caching at clients and chunkservers due to the frequent case of streaming

Single-Master Design

- Simple
- Master answers only chunk locations
- A client typically asks for multiple chunk locations in a single request
- The master also predicatively provide chunk locations immediately following those requested

Chunk Size

- 64 MB
- Fewer chunk location requests to the master
- Reduced overhead to access a chunk
- Fewer metadata entries
  - Kept in memory

- Some potential problems with fragmentation

Metadata

- Three major types
  - File and chunk namespaces
  - File-to-chunk mappings
  - Locations of a chunk’s replicas

Metadata

- All kept in memory
  - Fast!
  - Quick global scans
    - Garbage collections
    - Reorganizations
  - 64 bytes per 64 MB of data
  - Prefix compression

Chunk Locations

- No persistent states
  - Polls chunkservers at startup
  - Use heartbeat messages to monitor servers
  - Simplicity
  - On-demand approach vs. coordination
    - On-demand wins when changes (failures) are often

Operation Logs

- Metadata updates are logged
  - e.g., <old value, new value> pairs
  - Log replicated on remote machines
- Take global snapshots (checkpoints) to truncate logs
  - Memory mapped (no serialization/deserialization)
  - Checkpoints can be created while updates arrive
- Recovery
  - Latest checkpoint + subsequent log file

Consistency Model

- Relaxed consistency
  - Concurrent changes are consistent but undefined
  - An append is atomically committed at least once
    - Occasional duplications
- All changes to a chunk are applied in the same order to all replicas
- Use version number to detect missed updates

System Interactions

- The master grants a chunk lease to a replica
- The replica holding the lease determines the order of updates to all replicas
- Lease
  - 60 second timeouts
  - Can be extended indefinitely
  - Extension request are piggybacked on heartbeat messages
  - After a timeout expires, the master can grant new leases

Data Flow

- Separation of control and data flows
  - Avoid network bottleneck
- Updates are pushed linearly among replicas
- Pipelined transfers
- 13 MB/second with 100 Mbps network

Snapshot

- Copy-on-write approach
  - Revoke outstanding leases
  - New updates are logged while taking the snapshot
  - Commit the log to disk
  - Apply to the log to a copy of metadata
  - A chunk is not copied until the next update

Master Operation

- No directories
- No hard links and symbolic links
- Full path name to metadata mapping
  - With prefix compression

Locking Operations

- A lock per path
  - To access /d1/d2/leaf
  - Need to lock /d1, /d1/d2, and /d1/d2/leaf
  - Can modify a directory concurrently
    - Each thread acquires
      - A read lock on a directory
      - A write lock on a file
  - Totally ordered locking to prevent deadlocks

Replica Placement

- Goals:
  - Maximize data reliability and availability
  - Maximize network bandwidth
- Need to spread chunk replicas across machines and racks
- Higher priority to replica chunks with lower replication factors
- Limited resources spent on replication

Garbage Collection

- Simpler than eager deletion due to
  - Unfinished replicated creation
  - Lost deletion messages
- Deleted files are hidden for three days
- Then they are garbage collected
- Combined with other background operations (taking snapshots)
- Safety net against accidents

Fault Tolerance and Diagnosis

- Fast recovery
  - Master and chunkserver are designed to restore their states and start in seconds regardless of termination conditions
- Chunk replication
- Master replication
  - Shadow masters provide read-only access when the primary master is down

Fault Tolerance and Diagnosis

- Data integrity
  - A chunk is divided into 64-KB blocks
  - Each with its checksum
  - Verified at read and write times
  - Also background scans for rarely used data

Measurements

- Chunkserver workload
  - Bimodal distribution of small and large files
  - Ratio of write to append operations: 3:1 to 8:1
  - Virtually no overwrites
- Master workload
  - Most request for chunk locations and open files
- Reads achieve 75% of the network limit
- Writes achieve 50% of the network limit

Major Innovations

- File system API tailored to stylized workload
- Single-master design to simplify coordination
- Metadata fit in memory
- Flat namespace

Outline

- Google File System (GFS)
- Hadoop Distributed File System (HDFS)
- HDFS Laboratory

Basic Features: HDFS

- Highly fault-tolerant
- High throughput
- Suitable for applications with large data sets
- Streaming access to file system data
- Can be built out of commodity hardware

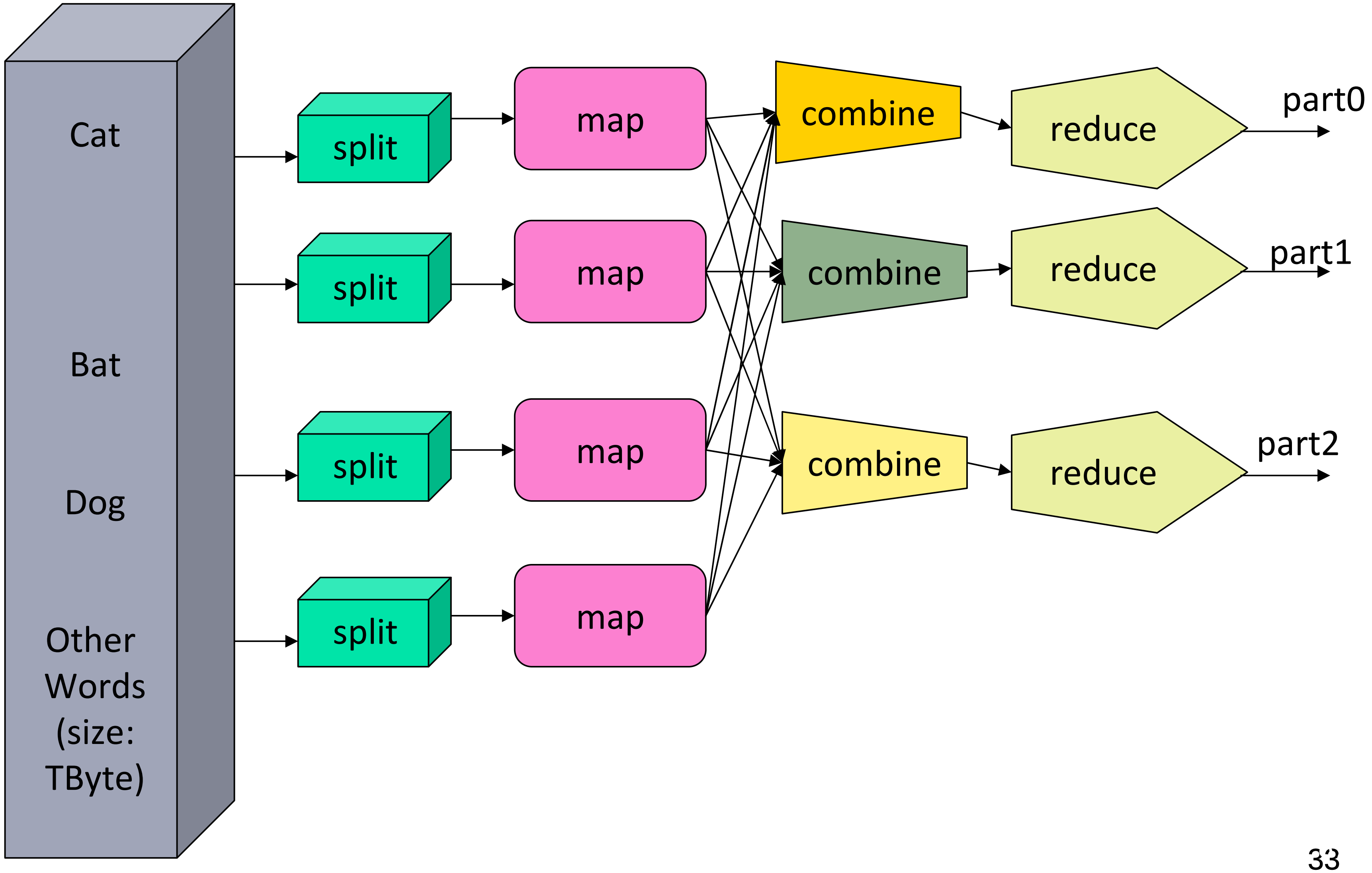
Fault Tolerance

- Failure is the norm rather than exception
- A HDFS instance may consist of thousands of server machines, each storing part of the file system’s data.
- Since we have huge number of components and that each component has non-trivial probability of failure means that there is always some component that is non-functional.
- Detection of faults and quick, automatic recovery from them is a core architectural goal of HDFS.

Data Characteristics

- Streaming data access
- Batch processing rather than interactive user access.
- Large data sets and files: gigabytes to terabytes size
- High aggregate data bandwidth
- Scale to hundreds of nodes in a cluster
- Tens of millions of files in a single instance
- Write-once-read-many: a file once created, written and closed need not be changed – this assumption simplifies coherency
- A map-reduce application or web-crawler application fits perfectly with this model.

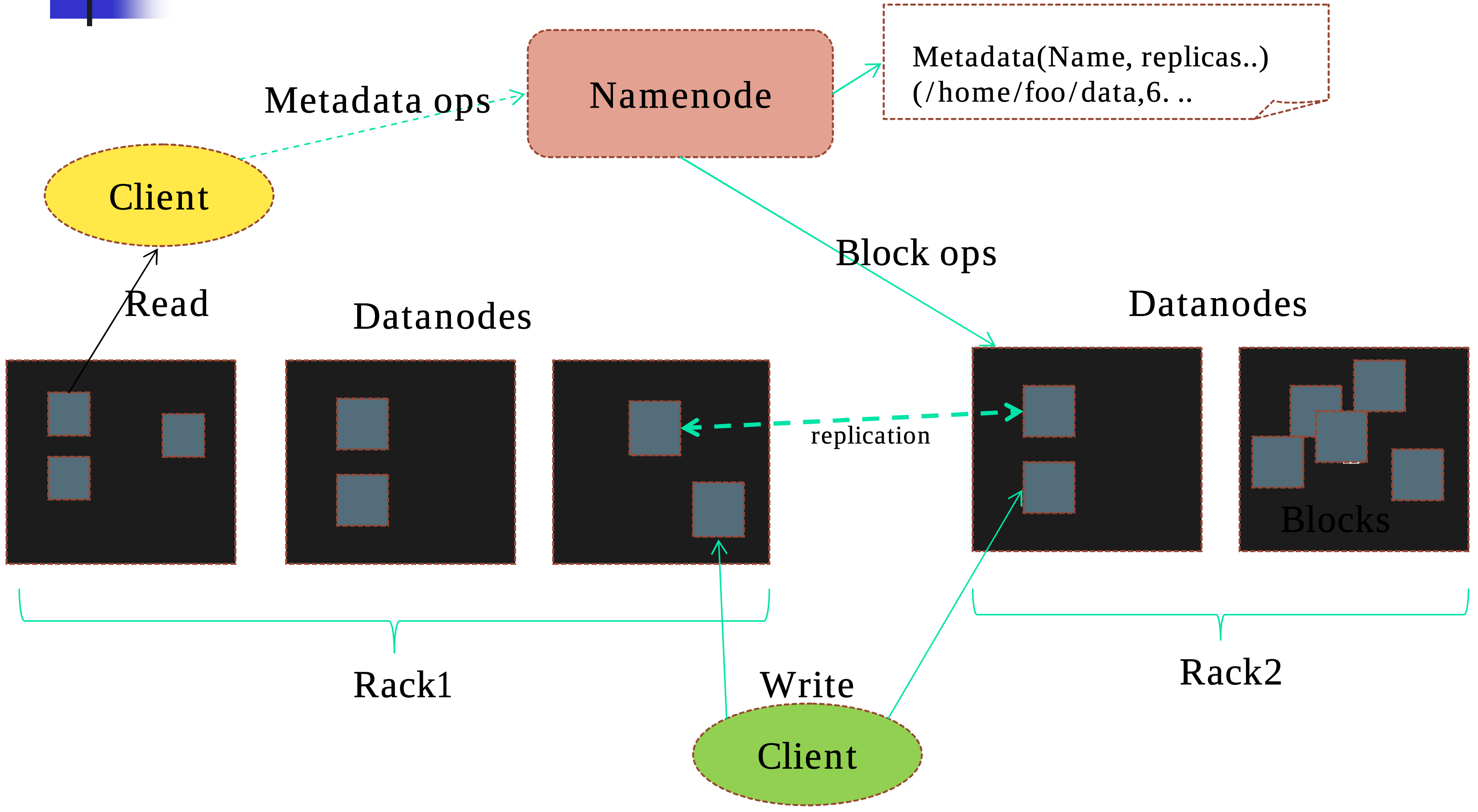
MapReduce


Cat
Bat
Dog
Other Words (size: TByte)
split
split
split
split
map
map
map
map
combine
combine
combine
reduce
reduce
reduce
part0
part1
part2

Namenode and Datanodes

- Master/slave architecture
- HDFS cluster consists of a single **Namenode**, a master server that manages the file system namespace and regulates access to files by clients.
- There are a number of **DataNodes** usually one per node in a cluster.
- The DataNodes manage storage attached to the nodes that they run on.
- HDFS exposes a file system namespace and allows user data to be stored in files.
- A file is split into one or more blocks and set of blocks are stored in DataNodes.
- DataNodes: serves read, write requests, performs block creation, deletion, and replication upon instruction from Namenode.

HDFS Architecture



File System Namespace

- Hierarchical file system with directories and files
- Create, remove, move, rename etc.
- Namenode maintains the file system
- Any meta information changes to the file system recorded by the Namenode.
- An application can specify the number of replicas of the file needed: replication factor of the file. This information is stored in the Namenode.

Data Replication

- HDFS is designed to store very large files across machines in a large cluster.
- Each file is a sequence of blocks.
- All blocks in the file except the last are of the same size.
- Blocks are replicated for fault tolerance.
- Block size and replicas are configurable per file.
- The Namenode receives a Heartbeat and a BlockReport from each DataNode in the cluster.
- BlockReport contains all the blocks on a Datanode.

Replica Placement

- The placement of the replicas is critical to HDFS reliability and performance.
- Optimizing replica placement distinguishes HDFS from other distributed file systems.
- Rack-aware replica placement:
  - Goal: improve reliability, availability and network bandwidth utilization
  - Research topic
- Many racks, communication between racks are through switches.
- Network bandwidth between machines on the same rack is greater than those in different racks.
- Namenode determines the rack id for each DataNode.
- Replicas are typically placed on unique racks
  - Simple but non-optimal
  - Writes are expensive
  - Replication factor is 3
  - Another research topic?
- Replicas are placed: one on a node in a local rack, one on a different node in the local rack and one on a node in a different rack.
- 1/3 of the replica on a node, 2/3 on a rack and 1/3 distributed evenly across remaining racks.

Replica Selection

- Replica selection for READ operation: HDFS tries to minimize the bandwidth consumption and latency.
- If there is a replica on the Reader node then that is preferred.
- HDFS cluster may span multiple data centers: replica in the local data center is preferred over the remote one.

Safemode Startup

- On startup Namenode enters Safemode.
- Replication of data blocks do not occur in Safemode.
- Each DataNode checks in with Heartbeat and BlockReport.
- Namenode verifies that each block has acceptable number of replicas
- After a configurable percentage of safely replicated blocks check in with the Namenode, Namenode exits Safemode.
- It then makes the list of blocks that need to be replicated.
- Namenode then proceeds to replicate these blocks to other Datanodes.

Filesystem Metadata

- The HDFS namespace is stored by Namenode.
- Namenode uses a transaction log called the EditLog to record every change that occurs to the filesystem meta data.
  - For example, creating a new file.
  - Change replication factor of a file
  - EditLog is stored in the Namenode's local filesystem
- Entire filesystem namespace including mapping of blocks to files and file system properties is stored in a file FsImage. Stored in Namenode's local filesystem.

Namenode

- Keeps image of entire file system namespace and file Blockmap in memory.
- 4GB of local RAM is sufficient to support the above data structures that represent the huge number of files and directories.
- When the Namenode starts up it gets the FsImage and Editlog from its local file system, update FsImage with EditLog information and then stores a copy of the FsImage on the filesytstem as a checkpoint.
- Periodic checkpointing is done. So that the system can recover back to the last checkpointed state in case of a crash.

Datanode

- A Datanode stores data in files in its local file system.
- Datanode has no knowledge about HDFS filesystem
- It stores each block of HDFS data in a separate file.
- Datanode does not create all files in the same directory.
- It uses heuristics to determine optimal number of files per directory and creates directories appropriately:
  - Research issue?
- When the filesystem starts up it generates a list of all HDFS blocks and send this report to Namenode: Blockreport.

The Communication Protocol

- All HDFS communication protocols are layered on top of the TCP/IP protocol
- A client establishes a connection to a configurable TCP port on the Namenode machine. It talks ClientProtocol with the Namenode.
- The Datanodes talk to the Namenode using Datanode protocol.
- RPC abstraction wraps both ClientProtocol and Datanode protocol.
- Namenode is simply a server and never initiates a request; it only responds to RPC requests issued by DataNodes or clients.

Objectives

- Primary objective of HDFS is to store data reliably in the presence of failures.
- Three common failures are: Namenode failure, Datanode failure and network partition.

DataNode Failure and Heartbeat

- A network partition can cause a subset of Datanodes to lose connectivity with the Namenode.
- Namenode detects this condition by the absence of a Heartbeat message.
- Namenode marks Datanodes without Hearbeat and does not send any IO requests to them.
- Any data registered to the failed Datanode is not available to the HDFS.
- Also the death of a Datanode may cause replication factor of some of the blocks to fall below their specified value.

Re-replication

- The necessity for re-replication may arise due to:
  - A Datanode may become unavailable,
  - A replica may become corrupted,
  - A hard disk on a Datanode may fail, or
  - The replication factor on the block may be increased.

Cluster Rebalancing

- HDFS architecture is compatible with data rebalancing schemes.
- A scheme might move data from one Datanode to another if the free space on a Datanode falls below a certain threshold.
- In the event of a sudden high demand for a particular file, a scheme might dynamically create additional replicas and rebalance other data in the cluster.
- These types of data rebalancing are not yet implemented: research issue.

Data Integrity

- Consider a situation: a block of data fetched from Datanode arrives corrupted.
- This corruption may occur because of faults in a storage device, network faults, or buggy software.
- A HDFS client creates the checksum of every block of its file and stores it in hidden files in the HDFS namespace.
- When a clients retrieves the contents of file, it verifies that the corresponding checksums match.
- If does not match, the client can retrieve the block from a replica.

Metadata Disk Failure

- FsImage and EditLog are central data structures of HDFS.
- A corruption of these files can cause a HDFS instance to be non-functional.
- For this reason, a Namenode can be configured to maintain multiple copies of the FsImage and EditLog.
- Multiple copies of the FsImage and EditLog files are updated synchronously.
- Meta-data is not data-intensive.
- The Namenode could be single point failure: automatic failover is NOT supported! Another research topic.

Data Blocks

- HDFS support write-once-read-many with reads at streaming speeds.
- A typical block size is 64MB (or even 128 MB).
- A file is chopped into 64MB chunks and stored.

Staging

- A client request to create a file does not reach Namenode immediately.
- HDFS client caches the data into a temporary file. When the data reached a HDFS block size the client contacts the Namenode.
- Namenode inserts the filename into its hierarchy and allocates a data block for it.
- The Namenode responds to the client with the identity of the Datanode and the destination of the replicas (Datanodes) for the block.
- Then the client flushes it from its local memory.

Staging (contd.)

- The client sends a message that the file is closed.
- Namenode proceeds to commit the file for creation operation into the persistent store.
- If the Namenode dies before file is closed, the file is lost.
- This client side caching is required to avoid network congestion; also it has precedence is AFS (Andrew file system).

Replication Pipelining

- When the client receives response from Namenode, it flushes its block in small pieces (4K) to the first replica, that in turn copies it to the next replica and so on.
- Thus data is pipelined from Datanode to the next.

Space Reclamation

- When a file is deleted by a client, HDFS renames file to a file in be the /trash directory for a configurable amount of time.
- A client can request for an undelete in this allowed time.
- After the specified time the file is deleted and the space is reclaimed.
- When the replication factor is reduced, the Namenode selects excess replicas that can be deleted.
- Next heartbeat(?) transfers this information to the Datanode that clears the blocks for use.

Application Programming Interface

- HDFS provides Java API for application to use.
- Python access is also used in many applications.
- A C language wrapper for Java API is also available.
- A HTTP browser can be used to browse the files of a HDFS instance.

Shell, Admin and Browser Interface

- HDFS organizes its data in files and directories.
- It provides a command line interface called the FS shell that lets the user interact with data in the HDFS.
- The syntax of the commands is similar to bash and csh.
- Example: to create a directory /foodir
- /bin/hadoop dfs –mkdir /foodir
- There is also DFSAdmin interface available
- Browser interface is also available to view the namespace.

Using HDFS

- hadoop fs

```
[-ls <path>]
[-du <path>]
[-cp <src> <dst>]
[-rm <path>]
[-put <localsrc> <dst>]
[-copyFromLocal <localsrc> <dst>]
[-moveFromLocal <localsrc> <dst>]
[-get [-crc] <src> <localdst>]
[-cat <src>]
[-copyToLocal [-crc] <src> <localdst>]
[-moveToLocal [-crc] <src> <localdst>]
[-mkdir <path>]
[-touchz <path>]
[-test -[ezd] <path>]
[-stat [format] <path>]
[-help [cmd]]
```

Using HDFS

- Want to list files in your HDFS home directory?
- Easy
  - hadoop fs -ls
- Basically we see most commands look similar
  - Format: hadoop *command -action options*
  - If you just type “hadoop”, you get all possible Hadoop commands
  - If you just type “hadoop fs”, you get all possible HDFS actions

Reference

- HDFS Architecture
- HDFS File System Shell Guide
- FileSystem (Hadoop 0.20.0 API)

Outline

- Google File System (GFS)
- Hadoop Distributed File System (HDFS)
- HDFS Laboratory

Step 1: Register an Account

- Please register an account at http://hadoop.nchc.org.tw
- You will receive an e-mail to activate your account.
- After activation, you will receive another pair of account/password for entering the Hadoop environment.

Step 2: Enter Hadoop Environment

- You can enter the Hadoop environment via http://hadoop.nchc.org.tw.
  - Use the account/password for that web site.
- You can also enter the Hadoop environment by using SSH client programs such as PuTTY or PieTTY.
  - Connect to hadoop.nchc.org.tw via port 22.
  - Use the second pair of account/password mailed to you.

Step 3: Access your file systems

- Local File System
  - Current working directory: `$ pwd`
  - List content of a directory: `$ ls`
- Hadoop Distributed File System
  - List content of a directory

```
$ hadoop fs -ls
```

Step 4: Set CLASSPATH

- You can edit .profile at your home directory to set CLASSPATH
  - Append the following two lines to .profile

```
CLASSPATH=/usr/lib/hadoop/hadoop-0.18.3-6cloudera0.3.0-core.jar
export CLASSPATH
```

  - Logout and re-connect to make the change effective.

Step 5: Copy Example Programs

- Copy example Java programs at /home/h830/hdfs-examples/*.java

```
$ cp /home/h830/hdfs-examples/*.java .
```

Step 6: Compile Java Programs

- Compile Java source programs (*.java) into Java byte codes (*.class)
  - Format: javac *javafile(s)*

```
$ javac hdfs01.java
```

- Make Java Jar files (*.jar) from Java byte codes (*.class)
  - Format: jar –cvf *jarfile classfile(s)*

```
$ jar -cvf hdfs01.jar hdfs01.class
```

Step 7: Execute Hadoop Programs

- Execute Java program in the Hadoop environment
  - Format: hadoop jar *jarfile mainclass argument(s)*

Ex 1: Copy from a local file

```
Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(conf);
DistributedFileSystem hdfs = (DistributedFileSystem)fs;

// Copy a file from the local file system to HDFS
Path srcPath = new Path(args[0]);
Path dstPath = new Path(args[1]);
hdfs.copyFromLocalFile(srcPath, dstPath);
```

Hw 1: Copy a file to local

- Write a Java program hdfshw01.java that copy a file in HDFS to a local file.
  - First argument: pathname of a HDFS file
  - Second argument: pathname of a local file
- Compile and test your program in the Hadoop environment
- Mail your Java program to me before 8/7
  - Email: cmwang@iis.sinica.edu.tw

Ex 2: Create a file

```
Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(conf);
DistributedFileSystem hdfs = (DistributedFileSystem)fs;

// Create a file in HDFS
Path filePath = new Path(args[0]);
FSDataOutputStream outputStream =
   hdfs.create(filePath);
byte[] buff = args[1].getBytes();
outputStream.write(buff, 0, buff.length);
```

Ex 3: Rename a file

```
Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(conf);
DistributedFileSystem hdfs = (DistributedFileSystem)fs;

// Rename a file in HDFS
Path fromPath = new Path(args[0]);
Path toPath = new Path(args[1]);
boolean isRenamed = hdfs.rename(fromPath, toPath);
```

Ex 4: Delete a file

```
Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(conf);
DistributedFileSystem hdfs = (DistributedFileSystem)fs;

// Delete HDFS file(s)
Path targetPath = new Path(args[0]);
boolean isDeleted = hdfs.delete(targetPath, false);
```

Ex 5: Get status of a file

```
Path targetPath = new Path(args[0]);

// Check if a file or directory exists in HDFS
if (!hdfs.exists(targetPath)) {
    System.out.print("File/Directory not found!\n");
    return;
}

// Check the status of a file or directory in HDFS
FileStatus fileStatus = hdfs.getFileStatus(targetPath);
if (fileStatus.isDir())
    System.out.print("Type: Directory\n");
else
    System.out.print("Type: File\n");
System.out.print("Owner: "+fileStatus.getOwner()+"\n");
System.out.print("Group: "+fileStatus.getGroup()+"\n");
```

Ex 6: Get locations of blocks

```
Path targetPath = new Path(args[0]);
FileStatus fileStatus = hdfs.getFileStatus(targetPath);

// Get the locations of blocks of a file in HDFS
BlockLocation[] blkLocations =
    hdfs.getFileBlockLocations(fileStatus, 0,
  fileStatus.getLen());
int blkCount = blkLocations.length;
for (int i=0; i < blkCount; i++) {
    String[] hosts = blkLocations[i].getHosts();
    System.out.print("Block "+i+": ");
    for (int j=0; j < hosts.length; j++) {
        System.out.print(hosts[j]+" ");
    }
    System.out.print("\n");
}
```

Hw 2: Character count of a text file

- Write a Java program hdfshw02.java that count the appearance of characters ‘a’ to ‘z’ in a file in HDFS.
  - Case insensitive
  - First argument: pathname of a HDFS file
- Sample output

```
count[a] = 1
count[b] = 0
count[c] = 2
. . .
```

- Mail your Java program to me before 8/7
  - Email: cmwang@iis.sinica.edu.tw