

CS435 Distributed systems

DIST. FILE SYSTEMS

Dr. Basit Qureshi

PhD FHEA SMIEEE MACM

www.drbasit.org

TOPICS

- What is a file system?
- Big data and storage
- Dist File Systems
- Google File System
- Hadoop Dist File System

© 2024 - Dr. Basit Qureshi

CS435 Distributed systems

A FILE SYSTEM

Dr. Basit Qureshi

PhD FHEA SMIEEE MACM

www.drbasit.org



A file system is a method or structure that a computer operating system uses to organize, store, retrieve, and manage files and data on storage devices.

Off system/online storage/ secondary memory	File system abstraction/ Databases	Offline/ tertiary memory/ DFS
RAID: Redundant Array of Inexpensive Disks	NAS: Network Accessible Storage	SAN: Storage area networks

File System Modules

Directory module:	relates file names to file IDs
Fil e module:	relates file IDs to particular files
Access control module:	checks permission for operation requested
Fileacœss module:	reads or writes file data or attributes
Block module:	acœsses and al locates disk blocks
Device module:	disk I/O and buffering

File Attribute Record
File length
Creation timestamp
Read timestamp
Write timestamp
Attribute timestamp
Reference count
Owner
File type
Access control list

UNIX file system operations

filedes = open(name, mode) filedes = creat(name, mode)	Opens an existing file with the given <i>name</i> . Creates a new file with the given <i>name</i> . Both operations deliver a file descriptor referencing the open file. The <i>mode</i> is <i>read</i> , <i>write</i> or both.
status = close(filedes)	Closes the open file <i>filedes</i> .
<pre>count = read(filedes, buffer, n) count = write(filedes, buffer, n)</pre>	Transfers <i>n</i> bytes from the file referenced by <i>filedes</i> to <i>buffer</i> . Transfers <i>n</i> bytes to the file referenced by <i>filedes</i> from buffer. Both operations deliver the number of bytes actually transferred and advance the read-write pointer.
pos = lseek(filedes, offset, whence)	Moves the read-write pointer to offset (relative or absolute, depending on <i>whence</i>).
<pre>status = unlink(name)</pre>	Removes the file <i>name</i> from the directory structure. If the file has no other names, it is deleted.
<pre>status = link(name1, name2)</pre>	Adds a new name (name2) for a file (name1).
<pre>status = stat(name, buffer)</pre>	Gets the file attributes for file name into buffer.

© 2024 - Dr. Basit Qureshi

CS435 Distributed systems

BIG DATA AND STORAGE

Dr. Basit Qureshi

PhD FHEA SMIEEE MACM

www.drbasit.org

Growth of Internet



BIG DATA AND STORAGE



Each rack contains 16-64 nodes

© 2024 - Dr. Basit Qureshi



BIG DATA AND STORAGE

- Large-scale computing for data mining problems on commodity hardware
- Challenges:
 - How do you distribute computation?
 - How can we make it easy to write distributed programs?
 - Machines fail:
 - One server may stay up 3 years (1,000 days)
 - If you have 1,000 servers, expect to loose 1/day
 - People estimated Google had ~1M machines in 2011
 - 1,000 machines fail every day!

BIG DATA AND STORAGE

• Problem:

- If nodes fail, how to store data persistently?
- Answer:

• Distributed File System:

- Provides global file namespace
- Google GFS; Hadoop HDFS;

• Typical usage pattern

- Huge files (100s of GB to TB)
- Data is rarely updated in place
- Reads and appends are common



© 2024 - Dr. Basit Qureshi

CS435 Distributed systems

DISTRIBUTED FILE SYSTEMS

Dr. Basit Qureshi

PhD FHEA SMIEEE MACM

www.drbasit.org

• Present a single view of all files across multiple computers

- Shared directory structure
- Shared files

Chunk servers

- File is split into contiguous chunks
- Typically each chunk is 16-128 MB
- Each chunk replicated (usually 2x or 3x)
- Try to keep replicas in different racks

Master node

- a.k.a. Name Node in Hadoop's HDFS
- Stores metadata about where files are stored
- Might be replicated

• Client library for file access

- Talks to master to find chunk servers
- Connects directly to chunk servers to access data



Reliable distributed file system

- Data kept in "chunks" spread across machines
- Each chunk **replicated** on different machines
 - Seamless recovery from disk or machine failure



Bring computation directly to the data!

Chunk servers also serve as compute servers

Desirable Properties from a DFS perspective

- Files are stored on a server machine
 - Client machine(s) do RPCs to server to perform operations on file
- Transparency: client accesses DFS files as if it were accessing local (say, Unix) files
 - Same API as local files, i.e., client code doesn't change
 - Need to make location, replication, etc. invisible to client
- Support concurrent clients
 - Multiple client processes reading/writing the file concurrently
- Replication: for fault-tolerance
- One-copy update semantics: when file is replicated, its contents, as visible to clients, are no different from when the file has exactly 1 replica

• Naming?

- The process of translating a name into its corresponding resource or identifier
 - File system organizes files in directories
 - /etc/file.txt
- Directory structures differentiated by:
 - Global vs Local naming:
 - Single global structure or different for each user?
 - /home/**user**/documents/file.txt
 - Location transparency:
 - Does the path name reveal anything about machine or server?
 - /rack3/server1/documents/file.txt
 - Location independence
 - When a file moves between machines, does its path name change?

GLOBAL DIRECTORY STRUCTURE

• Combine local directory structures under a new common root



GLOBAL DIRECTORY STRUCTURE

- Problem with "Combine under new common root:"
 - Using / for new root invalidates existing local names
- Solution (Unix United):
 - Use / for local root
 - Use . . to move to new root
 - Example: reach **u1** from **u2**: can use either

../../S1/usr/u1

or

```
/../S1/usr/u1
```

• Names are *not* location transparent



LOCAL DIRECTORY STRUCTURES

- Mounting
 - Subtree on one machine is *mounted* over/in-the-place-of a directory on another machine (called the *mount point*)
 - Original contents of mount point are invisible during mount (so usually an empty directory is chosen)
 - Structure changes dynamically
 - Each user has own view of File System



On S1: /mp On S2: /usr

On S1: /mp/u2/x On S2: /usr/u2/x

SHARED DIRECTORY SUBSTRUCTURE

- Each machine has local file system
- One subtree is shared by all machines



SEMANTICS OF FILE SHARING

- Unix semantics
 - + All updates are immediately visible
 - Generates a lot of network traffic
- Session semantics
 - + Updates visible when file closes
 - - Simultaneous updates are unpredictable (lost)
- Transaction semantics
 - Updates visible at end of transaction
- Immutable-files semantics
 - Updates create a **new version** of file
 - Now the problem is one of version management



- Basic Architecture
 - Client/Server Virtual file system (cf., Sun's NFS):
 - If file is local, access local file system
 - If file is remote, **communicate** with remote server



• Caching reduces

- Network delay
- Disk access delay
- Server caching is simple
 - No disk access on subsequent access
 - No cache coherence problems
 - But network delay still exists

• Client caching is more complicated

- When to update file on server?
- When/how to inform other processes when files is updated on server?
 - Consistency problems!

Challenges

- When to update file on server?
 - Write-through
 - Allows Unix semantics but overhead is significant
 - Delayed writing
 - Requires weaker semantics
 - Session semantics: only propagate update when file is closed
 - Transaction semantics: only propagate updates at end of transactions
- How to propagate changes to other caches/replicas?
 - Server initiates/informs other processes
 - Violates client/server relationship
 - Clients check periodically
 - Checking before each access defeats purpose of caching
 - Checking less frequently requires weaker semantics
 - Session semantics: only check when opening the file

- Stateless vs. Stateful Server
- **Stateful** = Maintain state of open files
- Client passes commands & data between user process & server

Problem when server crashes:

- State of open files is lost
- Client must restore state when server recovers



- Stateless Server (e.g., NFS) =
 Client maintains state of open files
- (Most) commands are *idempotent* (can be repeated). (File deletion and renaming aren't). When server crashes:
 - Client waits until server recovers
 - Client reissues read/write commands



- File replication improves
 - Availability: Multiple copies available
 - Reliability: Multiple copies help in recovery
 - Performance: Multiple copies remove bottlenecks and reduce network latency
 - Scalability: Multiple copies reduce bottlenecks
- Problem: File copies must be consistent
 - **Replication protocols**
 - Read-Any/Write-All
 - Problem: What if a server is temporarily unavailable?
 - Quorum-Based Read/Write
 - N copies; r = read quorum; w = write quorum
 - r+w > N and w > N/2
 - Any read sees at least one current copy
 - No disjoint writes



© 2024 - Dr. Basit Qureshi

CS435 Distributed systems

GOOGLE FILE SYSTEM

The following slides are taken from Virginia Tech

Dr. Basit Qureshi

PhD FHEA SMIEEE MACM

www.drbasit.org

Google Disk Farm





Design

Design factors

Failures are common (built from inexpensive commodity components)

Files

- large (multi-GB)
- mutation principally via appending new data
- low-overhead atomicity essential
- Co-design applications and file system API
- Sustained bandwidth more critical than low latency

File structure

Virginia

- Divided into 64 MB chunks
- **Chunk identified by 64-bit handle**
- Chunks replicated (default 3 replicas)
- Chunks divided into 64KB blocks
- Each block has a 32-bit checksum



Architecture



□ Manages namespace/metadata

□ Manages chunk creation, replication, placement

- Performs snapshot operation to create duplicate of file or directory tree
- Performs checkpointing and logging of changes to metadata
- Chunkservers

Virginia

ITech

- Stores chunk data and checksum for each block
- □ On startup/failure recovery, reports chunks to master
- Periodically reports sub-set of chunks to master (to detect no longer needed chunks)

Mutation operations

- Primary replica
 - □ Holds lease assigned by master (60 sec. default)
 - Assigns serial order for all mutation operations performed on replicas
- Write operation

Virginia

- 1-2: client obtains replica locations and identity of primary replica
- 3: client pushes data to replicas (stored in LRU buffer by chunk servers holding replicas)
- 4: client issues update request to primary
- □ 5: primary forwards/performs write request
- □ 6: primary receives replies from replica
- □ 7: primary replies to client
- Record append operation
 - Performed atomically (one byte sequence)
 - □ At-least-once semantics
 - □ Append location chosen by GFS and returned to client
 - **Extension to step 5:**
 - If record fits in current chunk: write record and tell replicas the offset
 - If record exceeds chunk: pad the chunk, reply to client to use next chunk



Consistency Guarantees



Write

Virginia

- Concurrent writes may be consistent but undefined
- Write operations that are large or cross chunk boundaries are subdivided by client into individual writes
- Concurrent writes may become interleaved
- Record append
 - Atomically, at-least-once semantics
 - Client retries failed operation
 - After successful retry, replicas are defined in region of append but may have intervening undefined regions
- Application safeguards
 - Use record append rather than write
 - □ Insert checksums in record headers to detect fragments
 - □ Insert sequence numbers to detect duplicates

	Write	Record Append
Serial	defined	defined
success		interspersed with
Concurrent	consistent	in consistent
successes	but undefined	
Failure	inconsistent	

Metadata management

	pathname	lock	chunk list
Logical structure	/home	read	Chunk4400488,
	/save		Chunk8ffe07783,
	/home/user/foo	write	Chunk6254ee0,
	/home/user	read	Chunk88f703,

- Namespace
 - Logically a mapping from pathname to chunk list
 - Allows concurrent file creation in same directory
 - **Read/write locks prevent conflicting operations**
 - **File deletion by renaming to a hidden name; removed during regular scan**
- Operation log
 - Historical record of metadata changes
 - Kept on multiple remote machines
 - **Checkpoint created when log exceeds threshold**
 - □ When checkpointing, switch to new log and create checkpoint in separate thread
 - Recovery made from most recent checkpoint and subsequent log
- Snapshot

Virginia

- □ Revokes leases on chunks in file/directory
- Log operation
- Duplicate metadata (not the chunks!) for the source
- On first client write to chunk:
 - Required for client to gain access to chunk
 - Reference count > 1 indicates a duplicated chunk
 - Create a new chunk and update chunk list for duplicate

Chunk/replica management

- Placement
 - □ On chunkservers with below-average disk space utilization
 - Limit number of "recent" creations on a chunkserver (since access traffic will follow)
 - □ Spread replicas across racks (for reliability)
- Reclamation

Virginia

- **Chunk become garbage when file of which they are a part is deleted**
- Lazy strategy (garbage college) is used since no attempt is made to reclaim chunks at time of deletion
- In periodic "HeartBeat" message chunkserver reports to the master a subset of its current chunks
- Master identifies which reported chunks are no longer accessible (i.e., are garbage)
- **Chunkserver reclaims garbage chunks**
- Stale replica detection
 - □ Master assigns a version number to each chunk/replica
 - □ Version number incremented each time a lease is granted
 - **Replicas on failed chunkservers will not have the current version number**
 - □ Stale replicas removed as part of garbage collection

Performance

Cluster	А	В
Chunkservers	342	227
Available disk space	72 TB	180 TB
Used disk space	55 TB	$155 \ \mathrm{TB}$
Number of Files	735 k	737 k
Number of Dead files	22 k	232 k
Number of Chunks	992 k	1550 k
Metadata at chunkservers	13 GB	21 GB
Metadata at master	$48 \mathrm{MB}$	60 MB



Cluster	А	В
Read rate (last minute)	583 MB/s	380 MB/s
Read rate (last hour)	562 MB/s	384 MB/s
Read rate (since restart)	589 MB/s	49 MB/s
Write rate (last minute)	1 MB/s	101 MB/s
Write rate (last hour)	2 MB/s	117 MB/s
Write rate (since restart)	25 MB/s	13 MB/s
Master ops (last minute)	325 Ops/s	533 Ops/s
Master ops (last hour)	381 Ops/s	518 Ops/s
Master ops (since restart)	202 Ops/s	347 Ops/s



© 2024 - Dr. Basit Qureshi

38

© 2024 - Dr. Basit Qureshi

CS435 Distributed systems

HADOOP DIST FILE SYSTEM

Dr. Basit Qureshi

PhD FHEA SMIEEE MACM

www.drbasit.org

The following slides are taken from **Prasanth Kothuri**, CERN

Introduction to HDFS

Prasanth Kothuri, CERN



What's HDFS

- HDFS is a distributed file system that is fault tolerant, scalable and extremely easy to expand.
- HDFS is the primary distributed storage for Hadoop applications.
- HDFS provides interfaces for applications to move themselves closer to data.
- HDFS is designed to 'just work', however a working knowledge helps in diagnostics and improvements.



Components of HDFS

There are two (*and a half*) types of machines in a HDFS cluster

- <u>NameNode</u> :- is the heart of an HDFS filesystem, it maintains and manages the file system metadata. E.g; what blocks make up a file, and on which datanodes those blocks are stored.
- <u>DataNode</u> :- where HDFS stores the actual data, there are usually quite a few of these.



HDFS Architecture





Introduction to HDFS

Unique features of HDFS

HDFS also has a bunch of unique features that make it ideal for distributed systems:

- <u>Failure tolerant</u> data is duplicated across multiple DataNodes to protect against machine failures. The default is a replication factor of 3 (every block is stored on three machines).
- Scalability data transfers happen directly with the DataNodes so your read/write capacity scales fairly well with the number of DataNodes
- <u>Space</u> need more disk space? Just add more DataNodes and rebalance
- <u>Industry standard</u> Other distributed applications are built on top of HDFS (HBase, Map-Reduce)

HDFS is designed to process large data sets with write-once-read-many semantics, it is not for low latency access



HDFS – Data Organization

- Each file written into HDFS is split into data blocks
- Each block is stored on one or more nodes
- Each copy of the block is called replica
- Block placement policy
 - First replica is placed on the local node
 - Second replica is placed in a different rack
 - Third replica is placed in the same rack as the second replica



Read Operation in HDFS





Write Operation in HDFS





HDFS Security

- Authentication to Hadoop
 - Simple insecure way of using OS username to determine hadoop identity
 - Kerberos authentication using kerberos ticket
 - Set by hadoop.security.authentication=simple|kerberos
- File and Directory permissions are same like in POSIX
 - read (r), write (w), and execute (x) permissions
 - also has an owner, group and mode
 - enabled by default (dfs.permissions.enabled=true)
- ACLs are used for implemention permissions that differ from natural hierarchy of users and groups
 - enabled by dfs.namenode.acls.enabled=true



HDFS Configuration

HDFS Defaults

- Block Size 64 MB
- Replication Factor 3
- Web UI Port 50070

HDFS conf file - /etc/hadoop/conf/hdfs-site.xml

```
<property>
  <name>dfs.namenode.name.dir</name>
  <value>file:///data1/cloudera/dfs/nn,file:///data2/cloudera/dfs/nn</value>
  </property>
```

<property> <name>dfs.blocksize</name> <value>268435456</value> </property>

```
<property>
        <name>dfs.replication</name>
        <value>3</value>
        </property>
```

```
<property>
        <name>dfs.namenode.http-address</name>
        <value>itracXXX.cern.ch:50070</value>
        </property>
```



Interfaces to HDFS

- Java API (DistributedFileSystem)
- C wrapper (libhdfs)
- HTTP protocol
- WebDAV protocol
- Shell Commands

However the command line is one of the simplest and most familiar



HDFS – Shell Commands

There are two types of shell commands User Commands

- hdfs dfs runs filesystem commands on the HDFS
- hdfs fsck runs a HDFS filesystem checking command

Administration Commands

hdfs dfsadmin - runs HDFS administration commands



HDFS – User Commands (dfs)

List directory contents

hdfs dfs -ls hdfs dfs -ls / hdfs dfs -ls -R /var

Display the disk space used by files

hdfs	dfs	-du	-h	/
hdfs	dfs	-du	/hk	base/data/hbase/namespace/
hdfs	dfs	-du	-h	/hbase/data/hbase/namespace/
hdfs	dfs	-du	-s	/hbase/data/hbase/namespace/



HDFS – User Commands (dfs)

Copy data to HDFS

hdfs dfs -mkdir tdata

hdfs dfs -ls

hdfs dfs -copyFromLocal tutorials/data/geneva.csv tdata

hdfs dfs -ls -R

Copy the file back to local filesystem

cd tutorials/data/

hdfs dfs -copyToLocal tdata/geneva.csv geneva.csv.hdfs md5sum geneva.csv geneva.csv.hdfs



HDFS – User Commands (acls)

List acl for a file

hdfs dfs -getfacl tdata/geneva.csv

List the file statistics – (%r – replication factor)

hdfs dfs -stat "%r" tdata/geneva.csv

Write to hdfs reading from stdin

echo "blah blah blah" | hdfs dfs -put - tdataset/tfile.txt hdfs dfs -ls -R hdfs dfs -cat tdataset/tfile.txt



HDFS – User Commands (fsck)

Removing a file

hdfs dfs -rm tdataset/tfile.txt hdfs dfs -ls -R

List the blocks of a file and their locations

hdfs fsck /user/cloudera/tdata/geneva.csv - files -blocks -locations

Print missing blocks and the files they belong to

hdfs fsck / -list-corruptfileblocks



HDFS – Adminstration Commands

Comprehensive status report of HDFS cluster

hdfs dfsadmin -report

Prints a tree of racks and their nodes

hdfs dfsadmin -printTopology

Get the information for a given datanode (like ping)

hdfs dfsadmin -getDatanodeInfo localhost:50020



HDFS – Advanced Commands

Get a list of namenodes in the Hadoop cluster

hdfs getconf -namenodes

Dump the NameNode fsimage to XML file

cd /var/lib/hadoop-hdfs/cache/hdfs/dfs/name/current hdfs oiv -i fsimage_00000000000003388 -o /tmp/fsimage.xml -p XML

The general command line syntax is

hdfs command [genericOptions] [commandOptions]



Other Interfaces to HDFS

HTTP Interface

http://quickstart.cloudera:50070

MountableHDFS – FUSE

mkdir /home/cloudera/hdfs
sudo hadoop-fuse-dfs dfs://quickstart.cloudera:8020
/home/cloudera/hdfs

Once mounted all operations on HDFS can be performed using standard Unix utilities such as 'ls', 'cd', 'cp', 'mkdir', 'find', 'grep',

