Distributed Systems

Dr. Rajiv Misra
Department of Computer Science and Engineering
Indian Institute of Technology, Patna

Lecture - 20
Google File System (GFS)

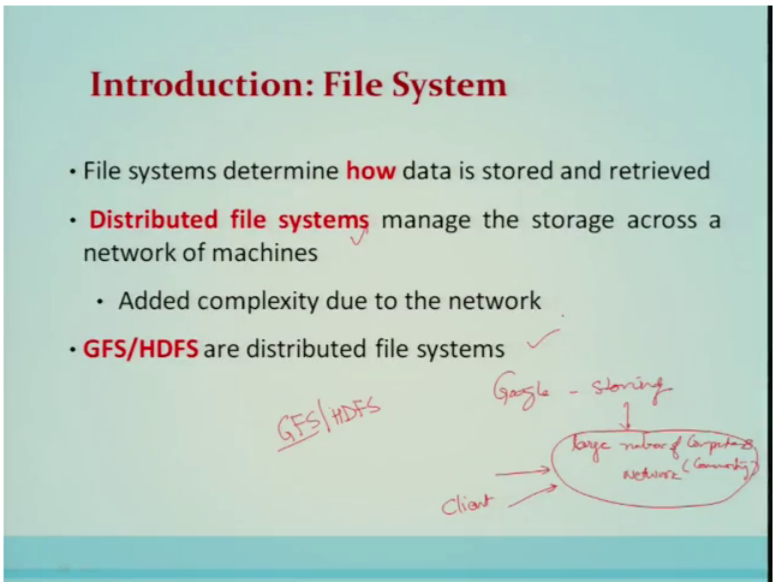
(Refer Slide Time: 00:15)

The Google File System (GFS)

Rajiv Misra
Dept. of Computer Science & Engineering
Indian Institute of Technology Patna
rajivm@iitp.ac.in

Google File System; GFS.

(Refer Slide Time: 00:17)



Introduction: File System

- File systems determine **how** data is stored and retrieved
- **Distributed file systems** manage the storage across a network of machines
    - Added complexity due to the network
- **GFS/HDFS** are distributed file systems ✓

GFS/HDFS
Google - Storing
large number of Computers
Network (Connectivity)
Client

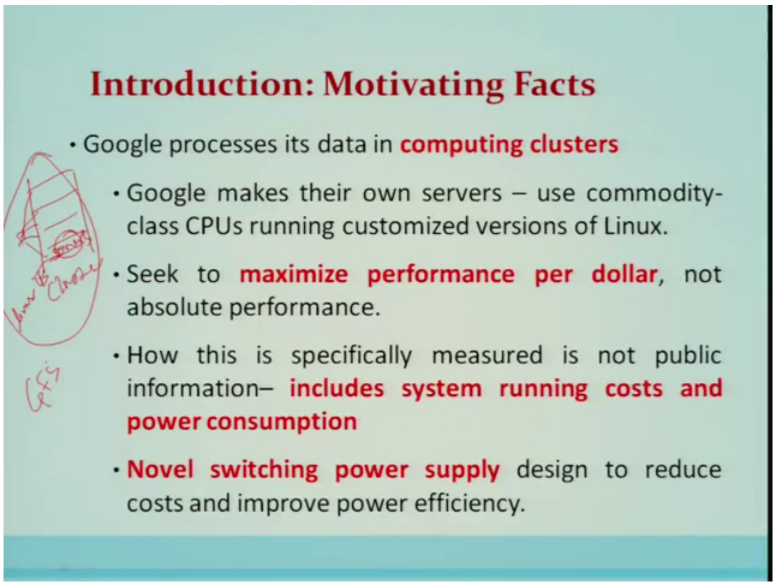
Introduction to a file system; a file system determines how the data is stored and retrieved. Distributed file systems manage the storage across the network of machines; added complexity is due to the network.

GFS, HDFS are distributed file systems. Before we start the topic of this particular lecture, let me introduce you to the motivation and the background for Google file system. So, Google is storing it is data over a large number of computers, which are having the storage within it. So, it is exploiting the storage of large number of computers which is also called as commodity hardware. This particular large number of computers which are connected through the network, are now used to store the information which the Google requires for its application.

Now, once having stored this information on a large number of computers; maybe thousands or more than a large number of clients required to access these particular data which is stored in the distributed environment simultaneously without any problem for this Google has devised a file system which is called as a Google file system and after that HDFS file system, hadoop has also made the same kind of file system with different terminologies within it.

So, let us understand this Google file system. So, in the Google file system description that is why we are going to make why, because the distributed file system existed earlier also, but as far as modern day applications are concerned, which involves the large data sets or large amount of information which is to be stored which cannot be stored at one place, which is stored in a distributed systems. If it is stored; then how many clients can simultaneously access it, so for that we are going to see a thing which is called a Google file system or a file systems which will facilitate not only to store the file over a large with the help of a large number of computers and also it will support how the large number of clients can access to that stored information in a distributed system.

(Refer Slide Time: 03:47)



Introduction: Motivating Facts

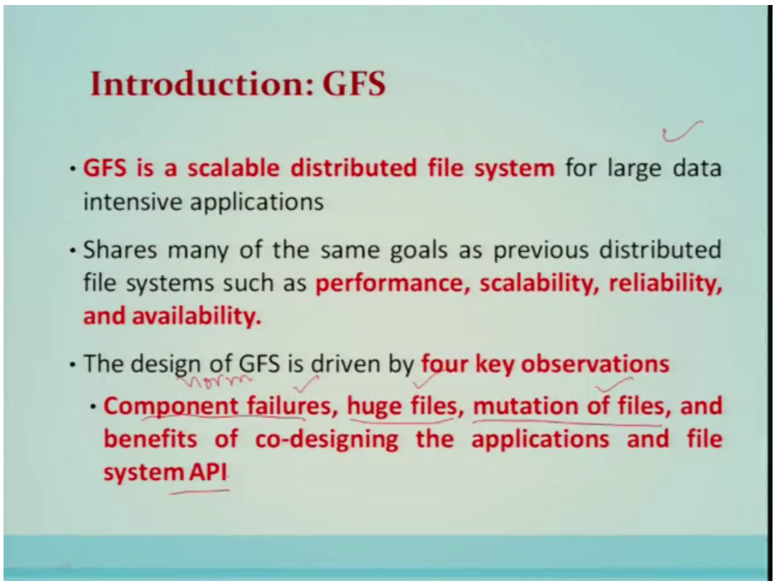
- Google processes its data in **computing clusters**
  - Google makes their own servers – use commodity-class CPUs running customized versions of Linux.
  - Seek to **maximize performance per dollar**, not absolute performance.
  - How this is specifically measured is not public information– **includes system running costs and power consumption**
  - **Novel switching power supply** design to reduce costs and improve power efficiency.

So, the introduction says like this Google processes it is data in a computing cluster. Google makes their own servers use commodity class CPUs running customized versions of Linux machines. We need to say that the cluster comprises of several servers is called a cluster maybe they are networked with each other. Now each of these servers they are running the Linux operating system within it.

So, Google will make use of this particular scenario, to implement the Google file system. Now this will seek to maximize performance and how this is specifically measured is not public information, then novel switching, power supply all these are generic problems of infrastructure that is going to solve if the commodity hardware's are used instead of being machines like supercomputers, that is; the philosophy of a Google. So, GFS utilizing these commodity hardware's and it does not require a very big infrastructure to support it.

So, let us go for the introduction of a GFS.

(Refer Slide Time: 05:11)



Introduction: GFS

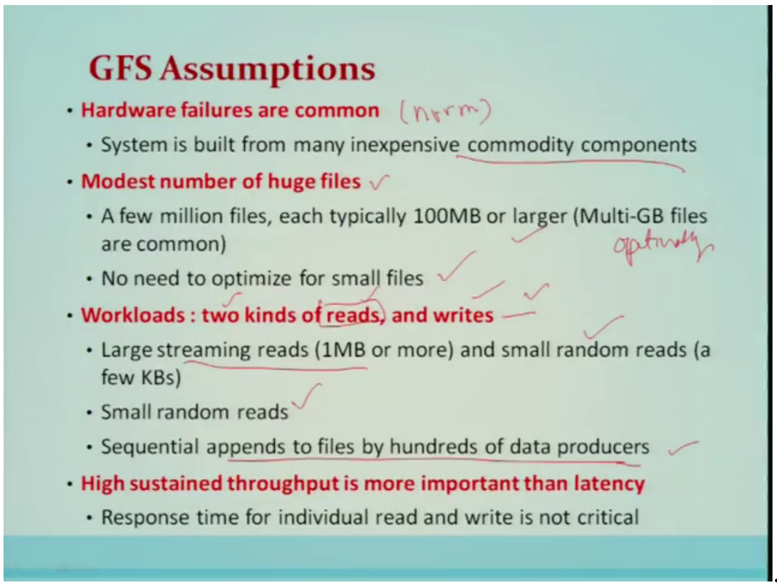
- GFS is a scalable distributed file system for large data intensive applications
- Shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability.
- The design of GFS is driven by four key observations
  - Component failures, huge files, mutation of files, and benefits of co-designing the applications and file system API

norm

GFS is scalable distributed file system for large data intensive applications shares many of the same goals as previous file systems such as performance, scalability, reliability and availability. They were there in earlier file system also here also they are also will be there, but in addition to that it is going to basically support the large data intensive applications.

So, the design of GFS is driven by four key observations, that is; if the commodity hardware's are there, then there is a possibility that the failures will be a norm, it is not an exception. Now another important thing is the file size is a big huge. So, it is huge files which are basically are going to be supported in your modern applications. Third one is the mutation of the files and the benefits of co-designing the applications and the API is to access all that things are basically these are the main point to be supported in a Google file system.

(Refer Slide Time: 06:26)



GFS Assumptions

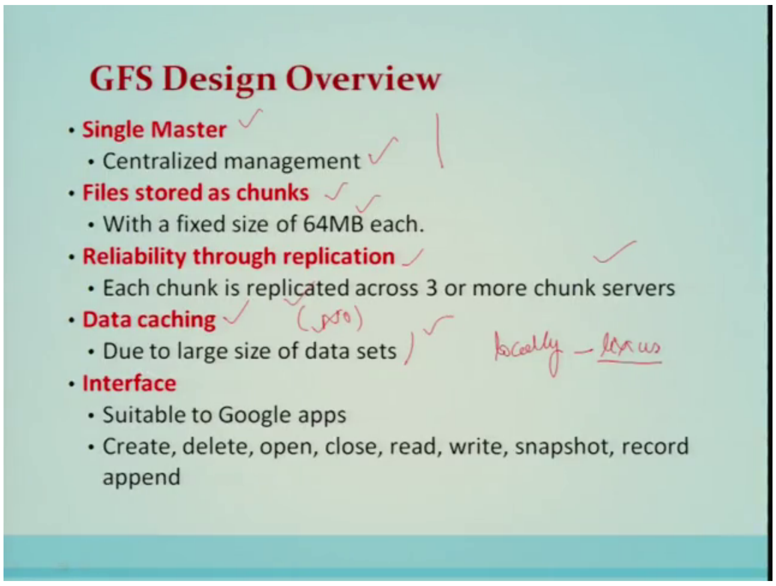
- Hardware failures are common (norm)
  - System is built from many inexpensive commodity components
- Modest number of huge files ✓
  - A few million files, each typically 100MB or larger (Multi-GB files are common) optimally
  - No need to optimize for small files ✓
- Workloads : two kinds of reads, and writes
  - Large streaming reads (1MB or more) and small random reads (a few KBs)
  - Small random reads ✓
  - Sequential appends to files by hundreds of data producers ✓
- High sustained throughput is more important than latency
  - Response time for individual read and write is not critical

Google file system assumptions the hardware failures are a common that it has become a non why? Because, if the commodity machine they are inexpensive commodity machines they are prone to failures and they may fail also. Similarly, as far as the files are concerned, now here modest numbers of huge files are there and see these files of a bigger size are now very common in this kind of environment. So, hence they are to be supported optimally whereas, small files are not that very common. So, basically it will not focus on optimizing about topped small files.

The workloads are two kinds mostly they are of reads and writes. So, writes means the files are written that is a big data set is generated once in the form of writes and after that most of the time it will be read only operations. So, two kinds of operations are supported. So, large streaming reads of 1 Megabyte or more and small random reads, so random reads small random reads will not be more focused. So, only the; it is large streaming reads or a sequential read operations will be supported here.

So, sequential append to a file by 100 of data producers will become an important issue as far as handling workloads are concerned. High sustained throughput is more important than the latency. So, the response time for individual read and writes operation is not very critical here in this kind of scenario.

(Refer Slide Time: 08:16)



GFS Design Overview

- Single Master
  - Centralized management
- Files stored as chunks
  - With a fixed size of 64MB each.
- Reliability through replication
  - Each chunk is replicated across 3 or more chunk servers
- Data caching (No)
  - Due to large size of data sets
- Interface
  - Suitable to Google apps
  - Create, delete, open, close, read, write, snapshot, record append

locally – linux

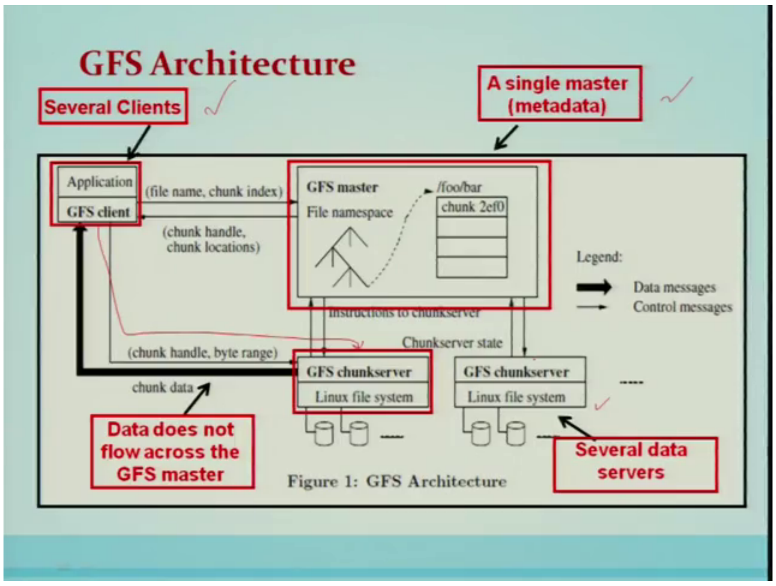
Google file system design overview. So, in Google file system these are the following components which will be participating and realizing this file system operation. The first one is called a master, there will be a single master and which will handle the centralized management. The second entity here is called file which are stored as the chunks. So, chunks are the another important entity in Google file system.

Chunk is nothing but a fixed size of 64 MB is called a chunk and the file is nothing, but file is stored in the form of a chunks. So, if a file is bigger than 64 MB, then file may contain more than one chunks and the entire data of the file will be stored. Reliability through replication these chunks are stored on the commodity hardware which are prone to the failures. So, the only way to overcome from these failure fault tolerant is basically go for the replication. So, by default the replication number is 3; that means, a particular chunk is replicated in three different chunk servers.

Now, the next entity is called data caching due to the large size of data sets data is not cached or is not required to be cached at the client as well as at the chunk servers; however, the chunk servers are running the Linux file system locally. So, Linux local caching is good enough to be supported here no external or no extra management of caching is supported here in Google file system. As far as the interface is concerned; it is suitable to the Google applications or Google apps it will create delete, open, close, read, write is snapshot record appends. These are the different interfaces which will support these operations that we will see in this part of the discussion.

Now, Google file system architecture. Let us go through this architecture there will be a single master which is shown over here.

(Refer Slide Time: 10:52)



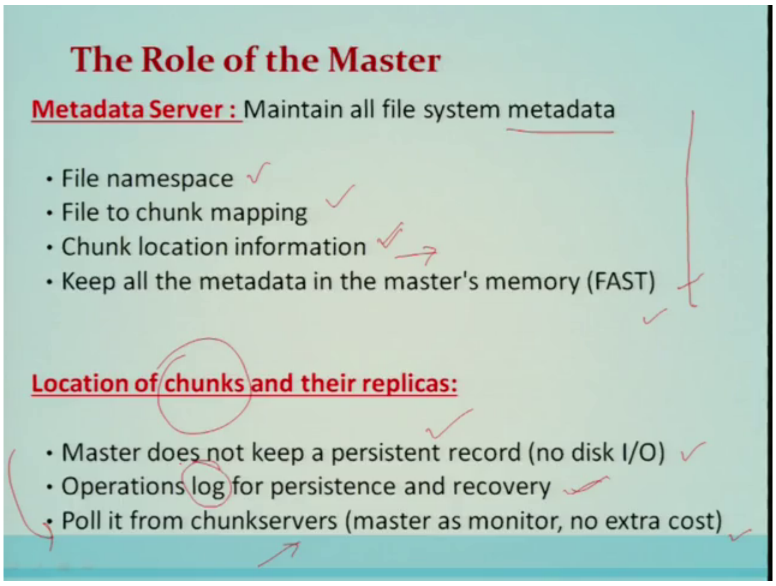
There will be single master and there will be a several clients as far as and the data will be stored on several servers which are called as chunk servers. Now the data is not flowing across client and master that you can see, rather the data is flowing from client to directly to the chunk servers which store these particular data. So, data flow does not flow across the GFS master.

(Refer Slide Time: 11:32)

GFS: Master

Now, let us go and see what the master does in Google file system.

(Refer Slide Time: 11:55)



The Role of the Master

Metadata Server : Maintain all file system metadata

- File namespace ✓
- File to chunk mapping ✓
- Chunk location information ✓
- Keep all the metadata in the master's memory (FAST)

Location of chunks and their replicas:

- Master does not keep a persistent record (no disk I/O) ✓
- Operations log for persistence and recovery
- Poll it from chunkservers (master as monitor, no extra cost) ✓

So, the role of the master; master basically is nothing, but I Metadata server; that means, it maintains all file system Metadata; that means, information about them the data where it is stored or information about the file system is called metadata and that is being maintained at the master.

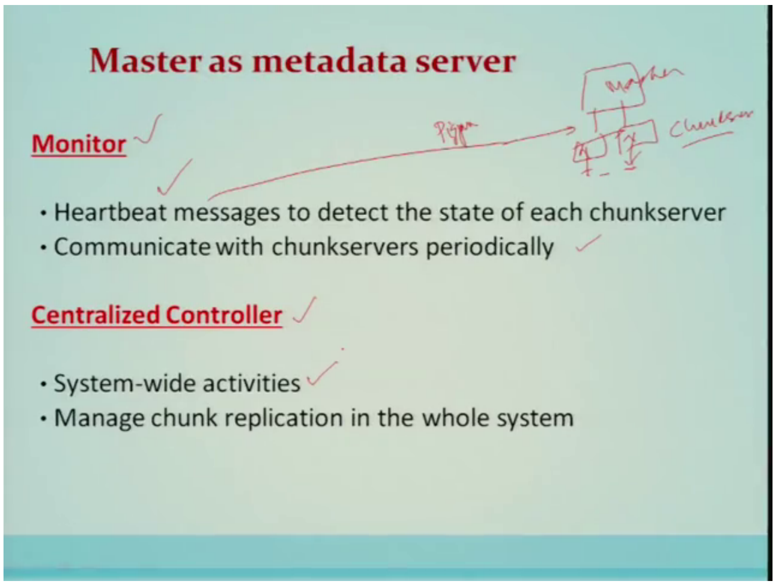
So, basically master will deal with handling four of the file name in space, handling file to chunk, mapping and it will also handle chunked location information, keep all the metadata in the master; masters first memory. Let me explain you through this particular example. So, the filename is space; that means, once the client gives a file name and file name and the directory will form the file name space filename in space is maintained by the Google file system master so and second information which is maintained by the master is about the chunk servers.

So, 2 important information which is maintained by the master, the first one is basically called namespace, and the second one is about the chunk servers. This information's the metadata is stored at the master. So, file to chunk mapping will be handled using these two information by the master chunk location information will be given. So, the file contains the file is stored in the chunks. So, those information will be provided through the chunk servers by the master that information also chunk placement information is also with the master. Now it will keep all the metadata in the master's pasty space.

Now, location of the chunk center replicas master does not keep the persistent record and operation logs for persistent and memory is maintained, pull it from the chunk servers master it as the monitor no cost meaning to say that the master does not maintain the information about the chunks or the location of the chunks. Location of the chunks are maintained by the chunk servers; however, the master continuously pulls from the chunk servers and fetch this information whenever is required. So, it is not going to store at the master level.

So, master does not keep this persistent information of the chunks and the replicas. So, operations for the persistent and the recovery for that log is maintained. So, all the informations or the metadata which is stored in the master a log will, basically be maintained for it is persistence so; that means, whenever there is a failure at any end at the end. So, master will use this log information or log will be used to restore or recover it.

(Refer Slide Time: 15:13)



Master as metadata server

Monitor

- Heartbeat messages to detect the state of each chunkserver
- Communicate with chunkservers periodically

Centralized Controller

- System-wide activities
- Manage chunk replication in the whole system

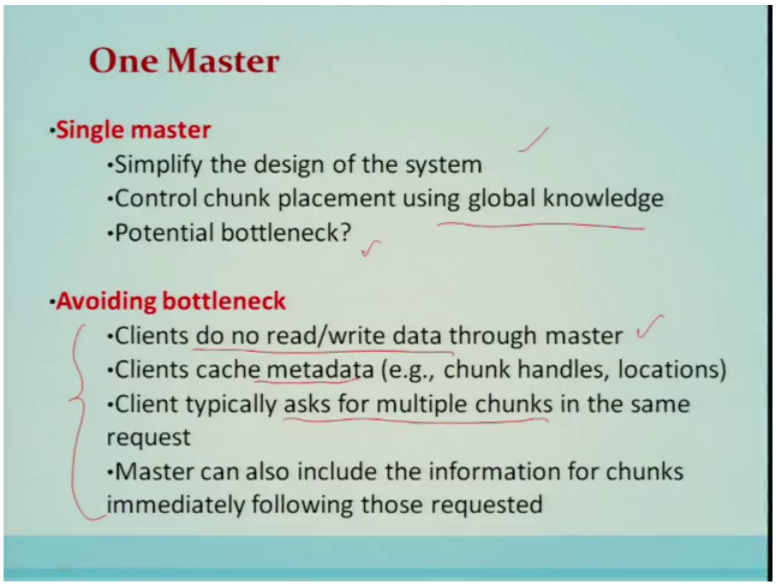
master
Piggy
Chunkserver

So, master has the metadata, server it will monitor. So, it will monitor as I told you that master does not keep the information about the chunks. So, the master and the chunks servers, so master used use this particular heartbeat messages to detect the state of these chunk servers and communicate with these chunk servers periodically. Now, whenever master requires the information of the chunks which are managed by the or which are is closed by the chunk servers, using these heartbeats and piggyback this information to fetch the details of the chunks.

And so, that is why this will communicate; master will communicate with the chunk servers periodically and also not only to get the health of the chunk servers, but also fetch the information of chunks norm it. Now, master is basically having a centralized controller it will also monitor the system-wide activities, it will manage the chunks replicas in the whole system.

So, there is a single master which will simplify the design of the system. It will control the chunk placement using the global knowledge, because the master is having the complete knowledge of the chunk servers and the placement of the chunk can be decided globally.

(Refer Slide Time: 16:43)



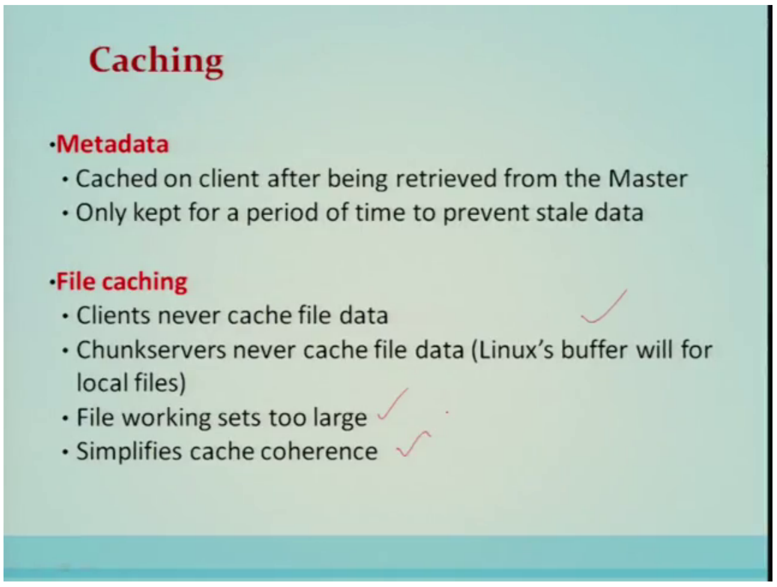
One Master

- •Single master
  - •Simplify the design of the system
  - •Control chunk placement using global knowledge
  - •Potential bottleneck?
- •Avoiding bottleneck
  - •Clients do no read/write data through master
  - •Clients cache metadata (e.g., chunk handles, locations)
  - •Client typically asks for multiple chunks in the same request
  - •Master can also include the information for chunks immediately following those requested

So, the bottlenecks are to be resolved, in the sense the client do no read write data through the master; that means, client need not have to do the read and write for the data operations through the master. So, master is free. So, the clients can directly access the data from the chunk servers.

Now, the clients cache the metadata only. So, clients typically ask for the multiple chunks in the same request and the master can also include the information for the chunks immediately following those requests. So, these particular provisions will free the master to be referred for read and write through operations from by the client. So, client is not required to access the master for each and every single operation for data related operations can directly be handled between client and the chunk service master is not basically involved in it.

(Refer Slide Time: 18:22)



Caching

- **Metadata**
  - Cached on client after being retrieved from the Master
  - Only kept for a period of time to prevent stale data
- **File caching**
  - Clients never cache file data
  - Chunkservers never cache file data (Linux's buffer will for local files)
  - File working sets too large
  - Simplifies cache coherence

Now, caching metadata that is; cast on the client after being written from the master; only kept for this; for a specific period of time to prevent by stale data. So, the caching for metadata is only done not the exact data. So, file caching clients never catch the file data as I explained to you earlier. So, chunk servers never catch the file data, because Linux buffer does that caching for the local file. So, that will be sufficient in Google file system.

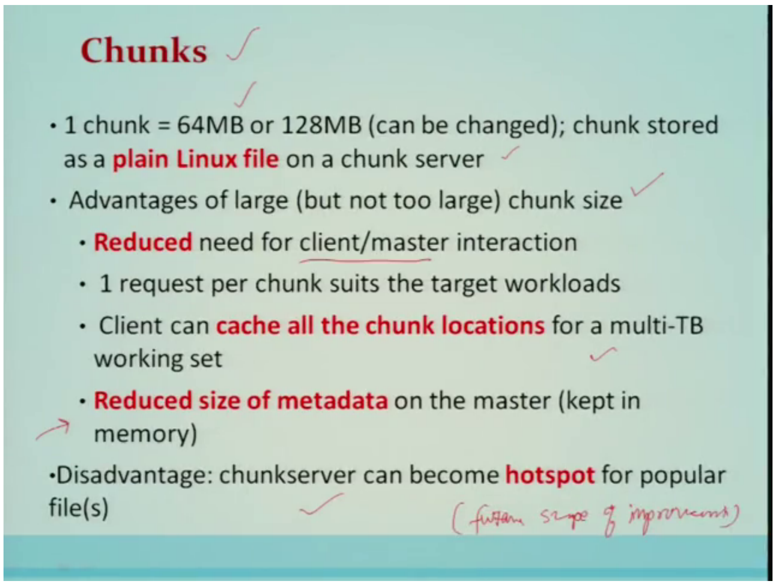
So, file working sets are too large and that is; why it is not required also to be cache why, because a big size files which are streamed as far as the operations are concerned are not required to be cached so; obviously, when caching is not there, then it will allow; it will simplify all the problems or issues of having a cache with the problems like cache coherence.

(Refer Slide Time: 19:06)

GFS: Chunks

Now, GFS chunk;

(Refer Slide Time: 19:20)



Chunks

- 1 chunk = 64MB or 128MB (can be changed); chunk stored as a **plain Linux file** on a chunk server
- Advantages of large (but not too large) chunk size
  - **Reduced** need for client/master interaction
  - 1 request per chunk suits the target workloads
  - Client can **cache all the chunk locations** for a multi-TB working set
  - **Reduced size of metadata** on the master (kept in memory)
- Disadvantage: chunkserver can become **hotspot** for popular file(s)

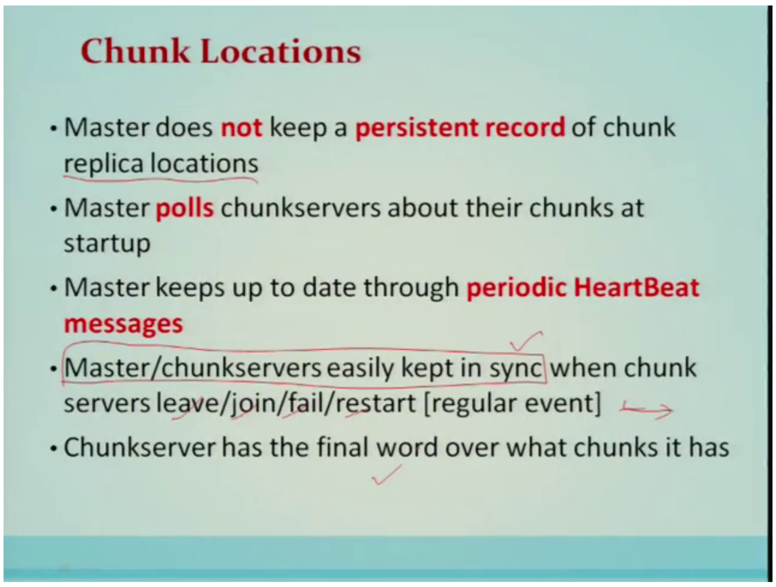
(future scope of improvement)

So chunks is the most important part of as far as the design is concerned. So, the chunk size is 64 MB. So, chunk is stored as the plain Linux file on a chunks server. So, advantage of keeping large chunk size is that, it will reduce the client and master interaction, as I told you earlier the previous slide. So, one request per chunk suits the target workload. Client can cache all the chunked location for multi-terabyte working set.

So, only the metadata is cached for the chunks so; obviously, if this particular chunk size will reduce the size of the metadata also which has to be kept in on the master. So, disadvantage of keeping a large sized chunk is that chunk servers can become the hottest part for some popular files. So, this particular problem can be can be handled in future also with some point of research that the clients who are already accessed those particular data can become the servers for the other clients and hence the hotspots can be removed, but this is for the future scope of improvement.

So, this particular hotspot problem is very practical and it requires some innovative solutions to be overcome.

(Refer Slide Time: 21:08)



Chunk Locations

- Master does **not** keep a **persistent record** of chunk replica locations
- Master **polls** chunkservers about their chunks at startup
- Master keeps up to date through **periodic HeartBeat messages**
- Master/chunkservers easily kept in sync when chunk servers leave/join/fail/restart [regular event]
- Chunkserver has the final word over what chunks it has

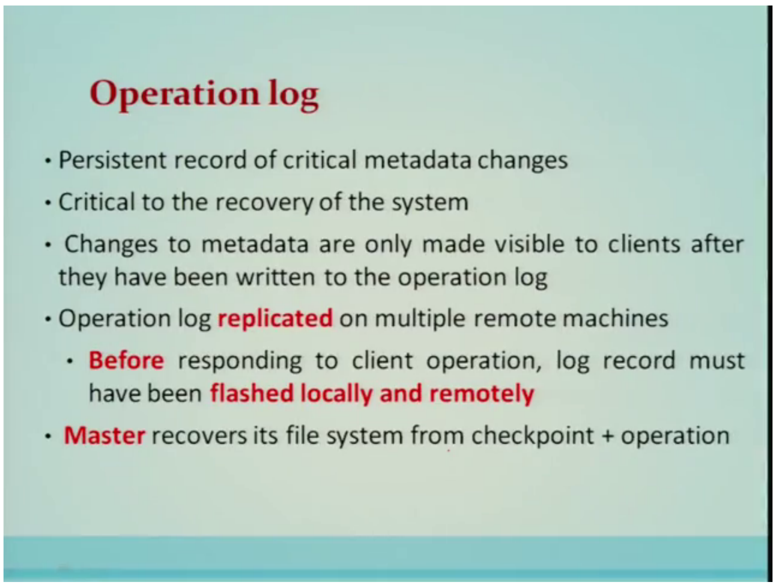
So, chunk locations master does not keep persistent record of the chunk replica locations. As I mentioned in the previous slide master polls chunkservers about their chunks at the startup. Master keeps up to date through periodic heartbeat messages. Master, chunkservers easily kept in sync when the chunk servers leave, join, failed, restart.

So, once if the master is not keeping the information about the chunk replicas locations, then it will allow the master and the chunk server to be kept in sync at all the point of time easily. By not keeping the chunk replicas locations at the master, now the master and the chunk server can easily be in the sync with the help of the heartbeat messages.

So, now when the chunks are out of the master, then basically these chunk servers can leave join fail restart at every regular interval and can be handled separately out of the master. So, chunk servers have the final words over what chunk it has.

Operation log persistent record of critical metadata changes is stored in the log. Critical to the recovery of the data, changes to the metadata are only made visible to the clients after they have written to the operation logs operation log, replicates on a multiple remote machines. So, before responding to the client operation, log must be must have been flashed locally and remotely.

(Refer Slide Time: 22:42)

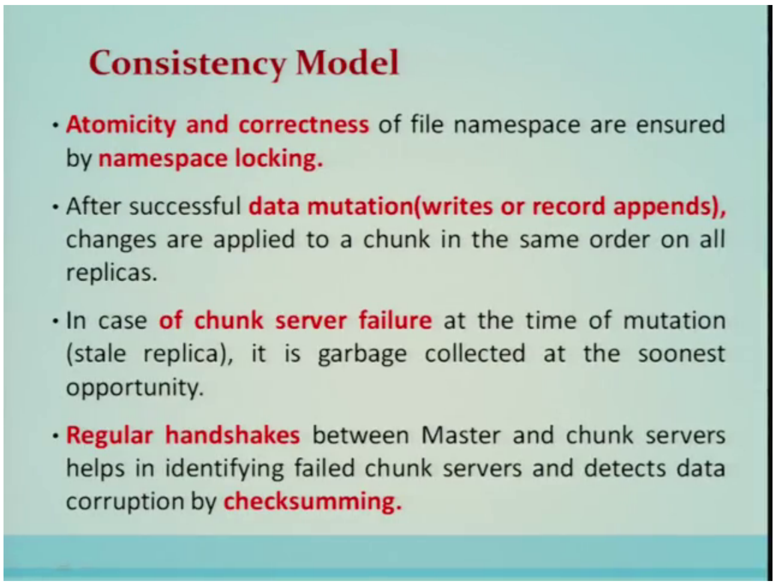


Operation log

- Persistent record of critical metadata changes
- Critical to the recovery of the system
- Changes to metadata are only made visible to clients after they have been written to the operation log
- Operation log **replicated** on multiple remote machines
    - **Before** responding to client operation, log record must have been **flashed locally and remotely**
- **Master** recovers its file system from checkpoint + operation

So, master recovers it file system from the checkpoint and the operation. So, all these aspects how the check pointing and the recovery we have already covered so that concept is used. So, that the master becomes the fault tolerant. So, the master is becoming inactive or a faulty then it will be recovered with the help of a checkpointing which is maintained in the stable storage or with the help of a mirroring of the master node.

(Refer Slide Time: 23:30)

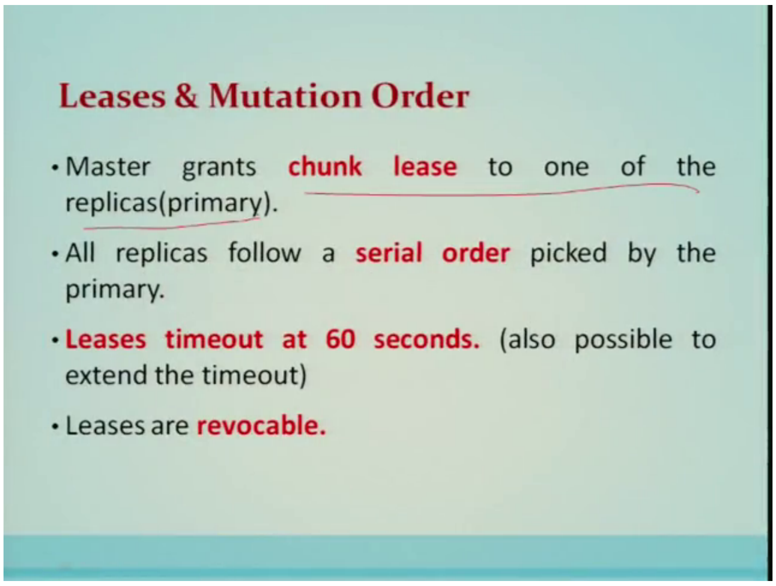


Consistency Model

- Atomicity and correctness of file namespace are ensured by namespace locking.
- After successful data mutation(writes or record appends), changes are applied to a chunk in the same order on all replicas.
- In case of chunk server failure at the time of mutation (stale replica), it is garbage collected at the soonest opportunity.
- Regular handshakes between Master and chunk servers helps in identifying failed chunk servers and detects data corruption by checksumming.

Now, consistency model atomicity and correctness of refining space are ensured by the by the namespace locking. Now, after successful data mutation writes or read appends, changes are applied to the chunk in the same order on all the replicas. In case of a chunk server failure at the time of mutation, it is the garbage collected at the soonest opportunity. So, regular handshakes between master and chunk servers helps in identifying the fields chunk servers and detect the data corruption by checksumming. So, this simplifies more complicated consistency model this will good enough to solve the consistency issues system interactions leases and mutation order.

(Refer Slide Time: 24:05)

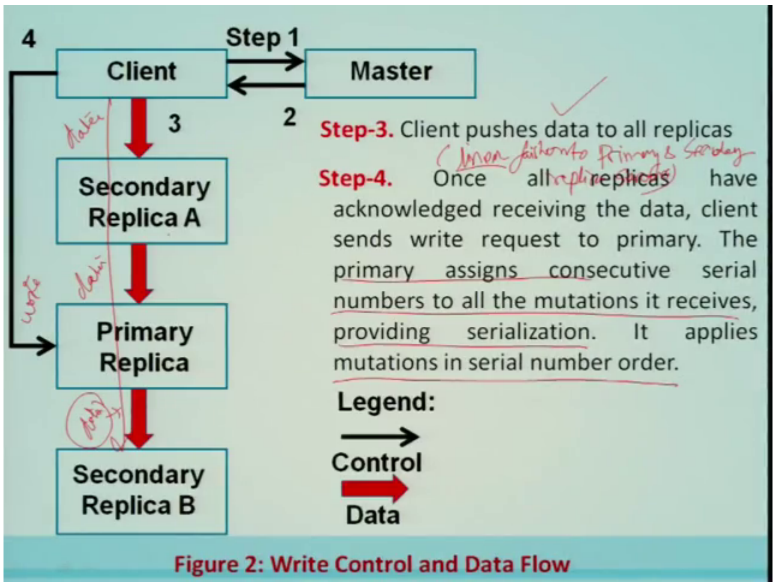


Leases & Mutation Order

- Master grants chunk lease to one of the replicas(primary).
- All replicas follow a serial order picked by the primary.
- Leases timeout at 60 seconds. (also possible to extend the timeout)
- Leases are revocable.

So, master grants the chunk lease to one of the replicas and this particular replica is called a primary replica. So, all the replicas follow the serial order picked by the by the primary. So, leases times out at 60 seconds and leases are revocable.

(Refer Slide Time: 24:42)



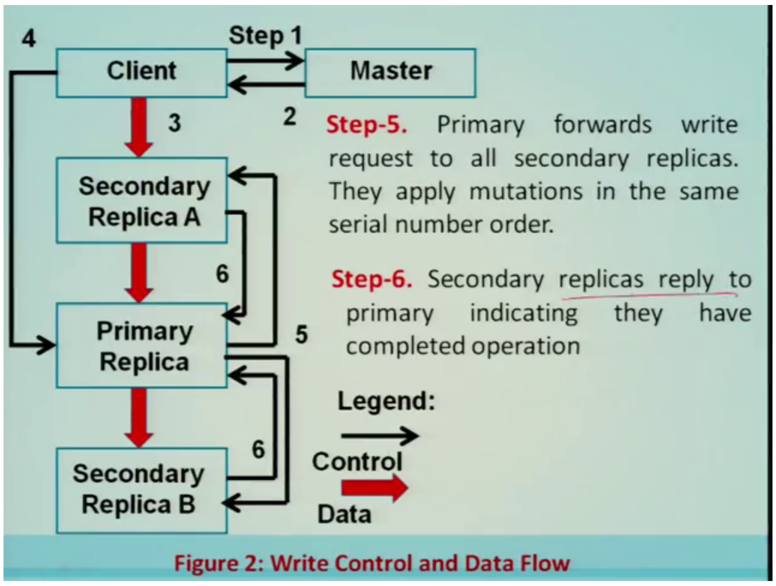
Step-3. Client pushes data to all replicas

Step-4. Once all [illegible] replicas have acknowledged receiving the data, client sends write request to primary. The primary assigns consecutive serial numbers to all the mutations it receives, providing serialization. It applies mutations in serial number order.

Figure 2: Write Control and Data Flow

Let us see the interactions. So, the first step is the client asks the master which chunk server holds the current lease of the chunk and the locations of the other replicas this is the first level interaction. This interaction happens when a client wants to access a particular file with the file name and a data at a particular location or a index, knowing the size of the chunk, it can this particular index is nothing, but an offset in a file which particular inside a file that data which is required by the application having calculated the index and the file name this particular message will be sent to the master.

(Refer Slide Time: 25:20)



Step-5. Primary forwards write request to all secondary replicas. They apply mutations in the same serial number order.

Step-6. Secondary replicas reply to primary indicating they have completed operation

Figure 2: Write Control and Data Flow

So, master asked the, so the client asked the master about which chunk servers holds that current lease of the chunk and the location of other replicas. Master will respond by giving the identity of the primary and the location of the secondary replicas. So, this information will be used by the client and the data will be pushed this is the data red line shows the data will be pushed directly in a linear fashion to the primary and secondary replica servers.

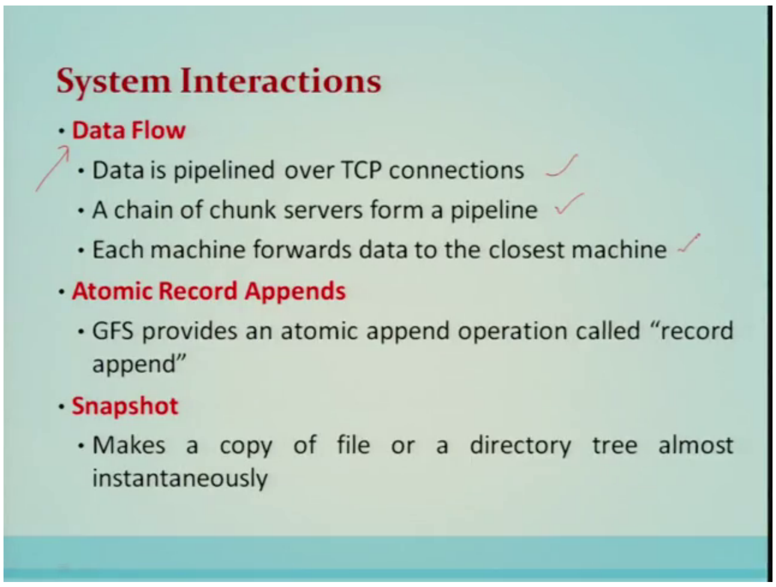
So, this particular flow of the data will use the simple network bandwidth it will not follow any topology. So, it basically it will follow a linear fashion the data will be pushed here to all the replicas, because now the client knows the primary as well as secondary replicas and it will form a chain and the data will be post.

Fourth step says that once all the replicas have acknowledged the receipt of the data, the client sends the right request to the primary; now here it says to right. So, data which is pushed to all the replicas servers that is primary and secondary replicas now the client will issue right command to the primary replica.

So, the primary is now having the leads of all the for maintaining all the replicas. So, the primary assigns a consecutive serial number to all the mutations it receives and it will provide the serialization and it applies the mutations in that serial order. So, the data in the form of the chunks will get assigned the serial order or serial number from the primary from the primary replica and these serial numbers are given or assigned and these secondary replicas will use that serial number to order all the notations. Now the primary forwards the right requires to all the secondary replica they apply the mutations in the same serial order which I have told you.

The step number six says that that all the secondary replicas, they will reply to the primary replica that they have completed the operation. Now having completed all the operation the primary replica the primary replies to the client with the success or if it is not success then some error message it will provide to the client.

(Refer Slide Time: 29:24)



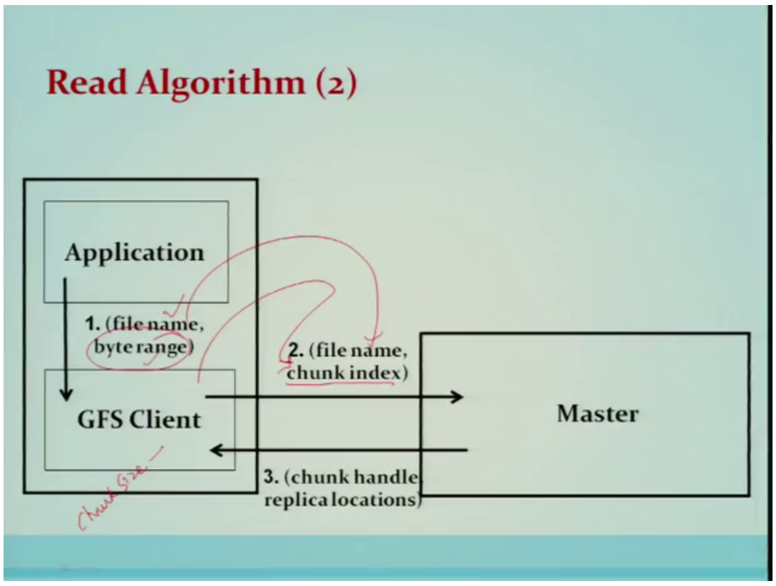
System Interactions

- **Data Flow**
  - Data is pipelined over TCP connections ✓
  - A chain of chunk servers form a pipeline ✓
  - Each machine forwards data to the closest machine ✓
- **Atomic Record Appends**
  - GFS provides an atomic append operation called "record append"
- **Snapshot**
  - Makes a copy of file or a directory tree almost instantaneously

So, that was the system interactions. So, system interactions has two parts one is called data flow; that means, data will be pushed through the network in a linear fashion among all the three servers that is primary and to secondaries. So, the data is pipelined that is called pipelining and data is pipeline over TCP connections a chain of chunk servers form the pipeline. So, each machine forwards the data to the closest machine that is called data flow.

Now, second level interaction is the atomic record appends GFS provides an atomic append operation called record append and in the snapshot means that it can make a copy of a file or a directory tree almost instantaneously. Let us see the read algorithm.

(Refer Slide Time: 30:02)



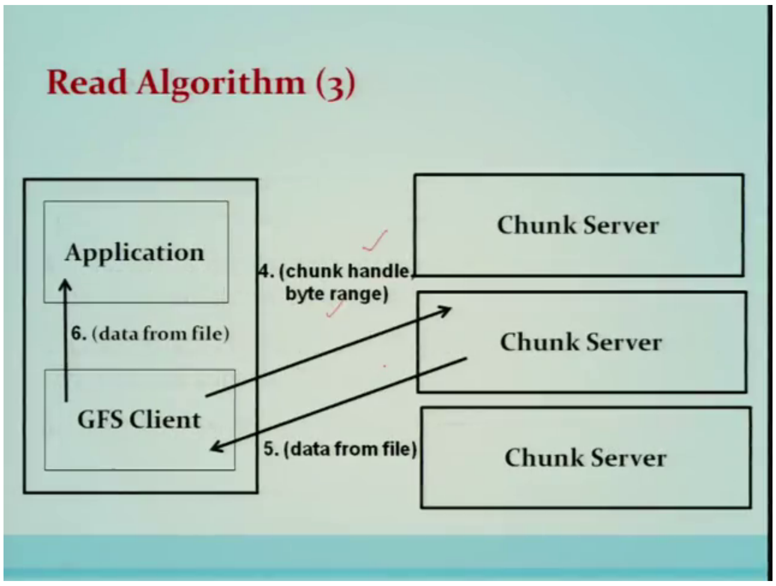
Read Algorithm (2)

Application
1. (file name, byte range)
GFS Client
2. (file name, chunk index)
Master
3. (chunk handle replica locations)
chunk size

So, the application originates the read request, here to the GFS client indicating the file name and the byte range. So, out of this particular byte range and knowing the chunk size this client will compute the chunk index and copying the file name as it is. So, inside file the client will compute the chunk index and this particular pair the file name and exchange index will be given to the master in the master intern will provide a metadata for the chunk that is called chunk handle.

And also the replicas, where that chunk is stored. This information will be passed on from master to the client.

(Refer Slide Time: 31:02)

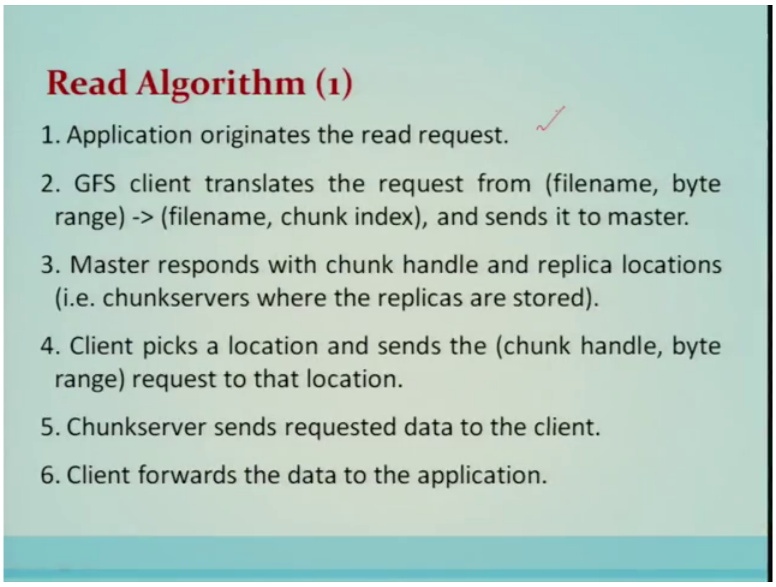


Read Algorithm (3)

Now the client, then directly send the messages that is the chunks handle and the byte to range direct to the chunk servers and these chunk servers will fetch the data from the file and will be given to the client and then the data will be passed on to that application that becomes the read algorithm.

Let me read out all the steps.

(Refer Slide Time: 31:36)

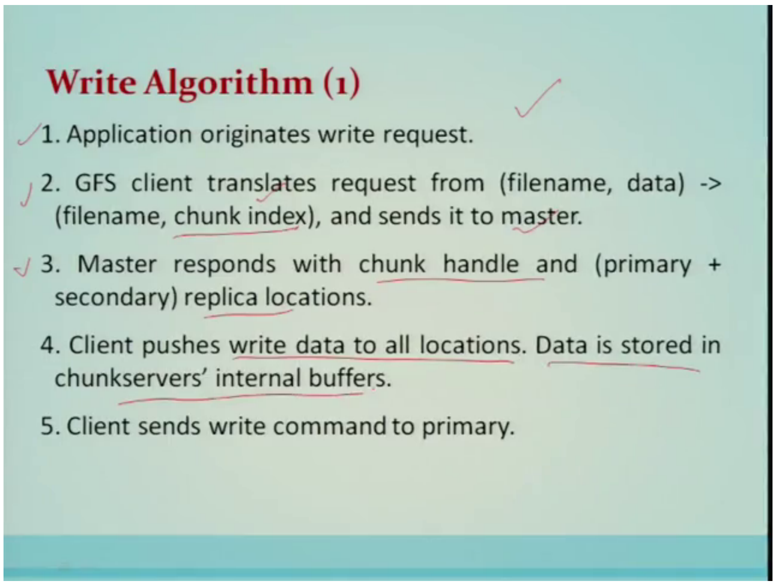


Read Algorithm (1)

1. Application originates the read request.
2. GFS client translates the request from (filename, byte range) -> (filename, chunk index), and sends it to master.
3. Master responds with chunk handle and replica locations (i.e. chunkservers where the replicas are stored).
4. Client picks a location and sends the (chunk handle, byte range) request to that location.
5. Chunkserver sends requested data to the client.
6. Client forwards the data to the application.

So, the application originates the read requests. GFS client translates the request from file name, byte range to the file name and chunk index, and send it to the master that I have explained. The master responds with the chunks handle and the replica locations. Now the client picks the location and sends the chunk handle and a byte range with the request to read to that particular location. Chunkservers send the requested data to the client and the client forwards the data back to the application.

(Refer Slide Time: 32:13)



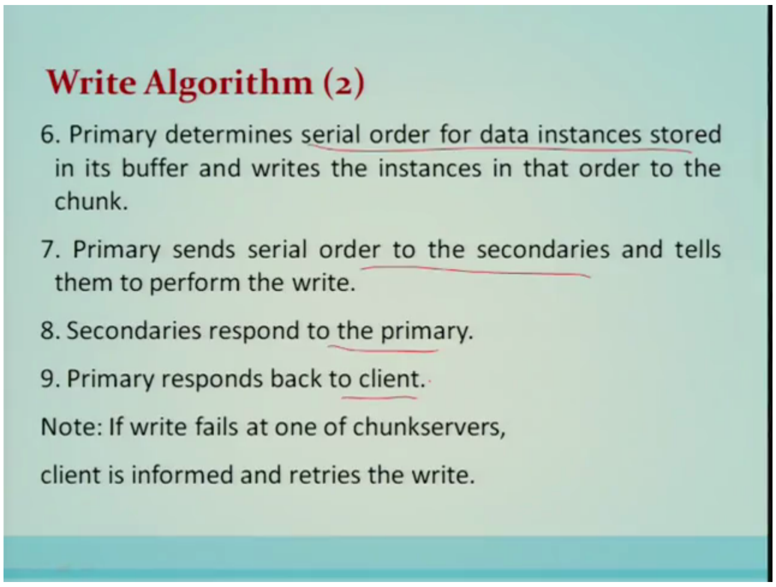
Write Algorithm (1)

1. Application originates write request.
2. GFS client translates request from (filename, data) -> (filename, chunk index), and sends it to master.
3. Master responds with chunk handle and (primary + secondary) replica locations.
4. Client pushes write data to all locations. Data is stored in chunkservers' internal buffers.
5. Client sends write command to primary.

So, that becomes the read algorithm about write algorithm. I have earlier explained during the system interactions, but let us go ahead again. So, the application originates the write request. GFS client translate the request from file, data to the file name, and data to the chunk index, and send it to the master that we have already seen in the read algorithm.

Now, master responds with a chunk handle and the replica location that also we have seen in the read algorithm. Now, the client pushes the write data to all the locations that I have already explained in the previous slide of the system interactions. The data is stored in the chunkservers in its internal buffers. When the client sends the write command to the primary memory, then the data will be looted then the data will be written.

(Refer Slide Time: 33:09)



Write Algorithm (2)

6. Primary determines serial order for data instances stored in its buffer and writes the instances in that order to the chunk.
7. Primary sends serial order to the secondaries and tells them to perform the write.
8. Secondaries respond to the primary.
9. Primary responds back to client.

Note: If write fails at one of chunkservers,

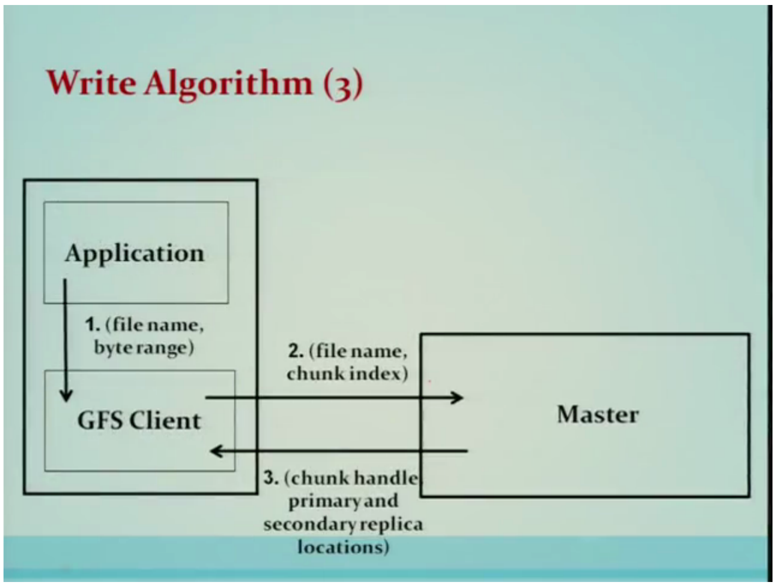
client is informed and retries the write.

So, the primary determines the serial order for the data instance is stored in its buffer and writes the instances in that order to the chunk. Primary sends the serial order to the secondaries and tell them to perform the write operation.

Secondaries perform the right operations to the primary request. So, primary response back to the client. Now, if these write fails at one of these chunks servers, then the client is informed and this write operation will be retried again to complete the write operations.

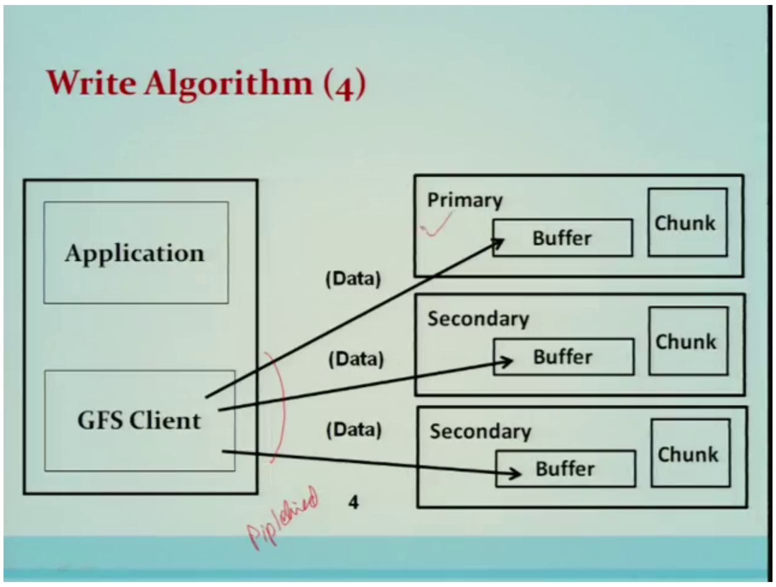
(Refer Slide Time: 33:44)

Write Algorithm (3)


Application
1. (file name, byte range)
GFS Client
2. (file name, chunk index)
Master
3. (chunk handle primary and secondary replica locations)

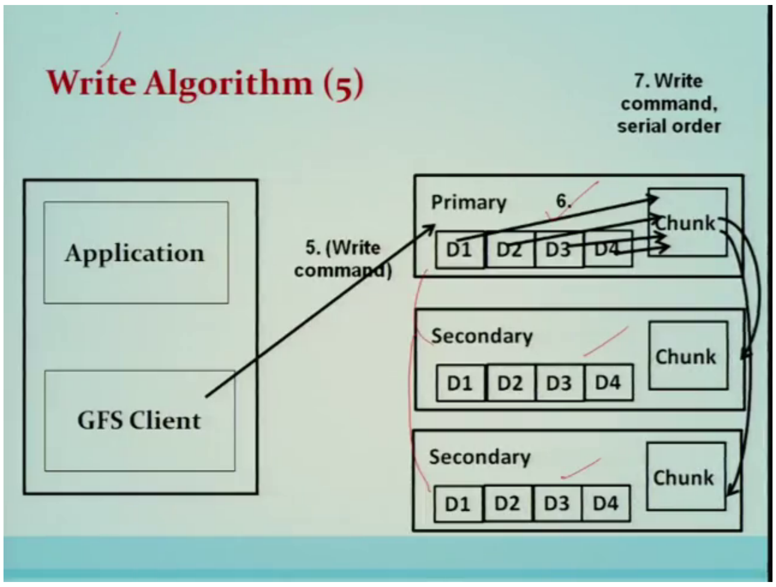
So, let us see this particular interaction.

(Refer Slide Time: 33:46)


Write Algorithm (4)

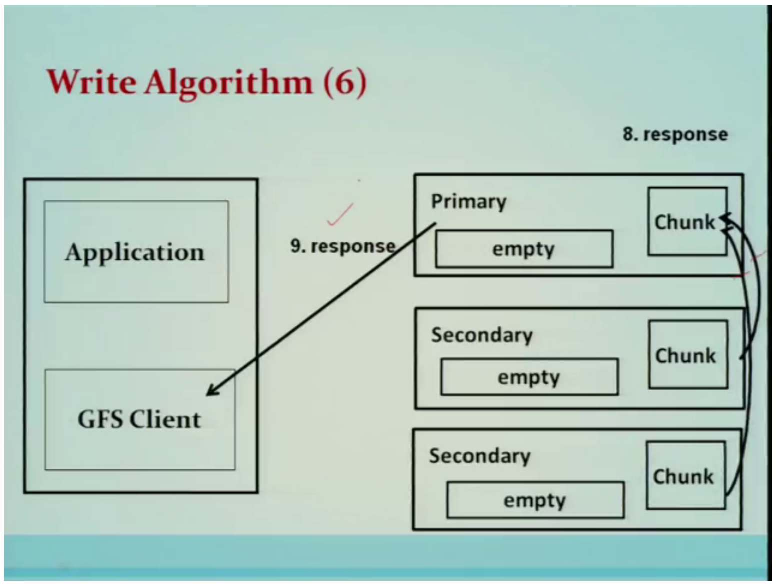
So, the data will be pushed, data will be pipelined to the primary and both the secondary servers.

(Refer Slide Time: 34:12)


Write Algorithm (5)

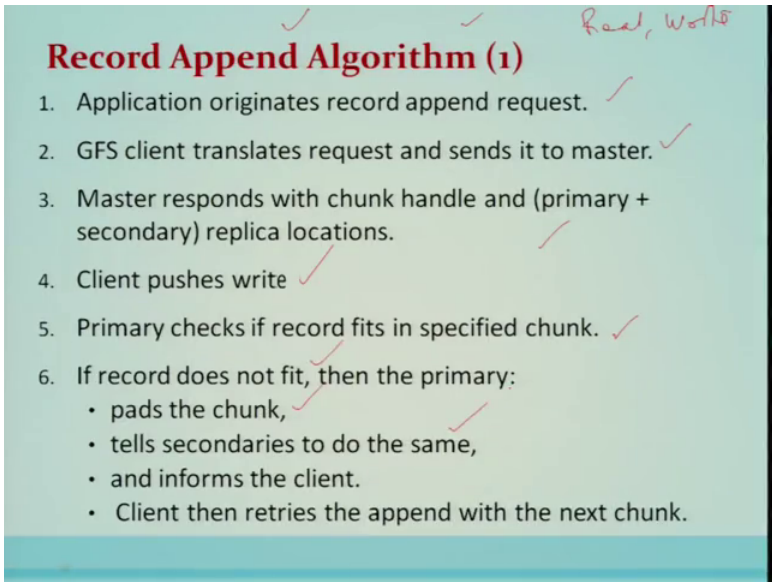
The primary then issue the sequence numbers like this and these sequence numbers will be now maintained as far as the sequence number order is maintained for storing that particular data for the write operations. This particular serial order is maintained in the chunk and they are being communicated to the; from primary to the secondary storage.

(Refer Slide Time: 34:54)



So, having written that sequence order messages in into the chunks. The primary the secondary is respond to the primary about the completion of the operation and there upon the primary responds to the client about the success.

(Refer Slide Time: 35:10)



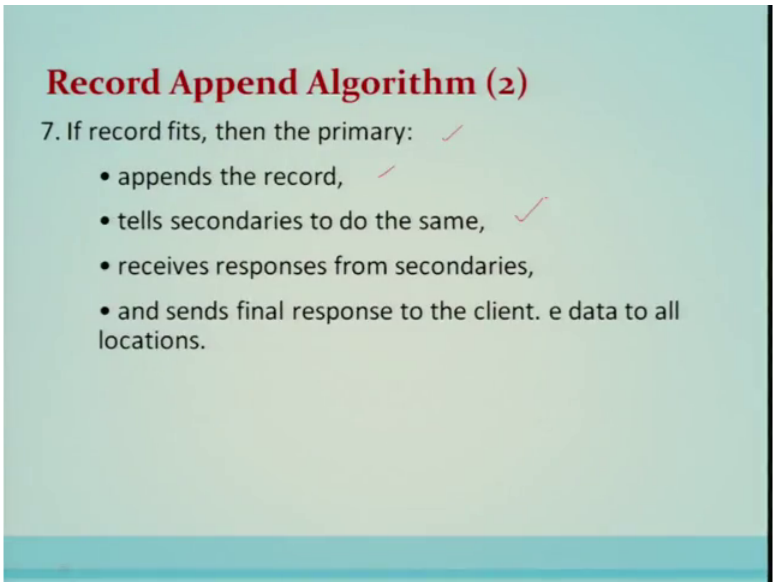
Record Append Algorithm (1)

Read, Write

1. Application originates record append request.
2. GFS client translates request and sends it to master.
3. Master responds with chunk handle and (primary + secondary) replica locations.
4. Client pushes write
5. Primary checks if record fits in specified chunk.
6. If record does not fit, then the primary:
   - pads the chunk,
   - tells secondaries to do the same,
   - and informs the client.
   - Client then retries the append with the next chunk.

Now, record append operation. As I told you that the file is created once and then it is not updated randomly, but the append operation is supported. So, the application; so let us see about the append operation which is supported here. So, we have seen the read we have seen the right now we have seen the append operation algorithm. Application originates the record a parent request. GFS client translates the question send it to the master. Master response with the end client pushes the right. So, primary checks if the required fits in the specified chunk. If the record does not fits, in the specified chunk, then primary will pad the chunk tells the secondary to do the same, and inform the client. Client then he tries the append with the next chunk and if the require fits.

(Refer Slide Time: 36:02)

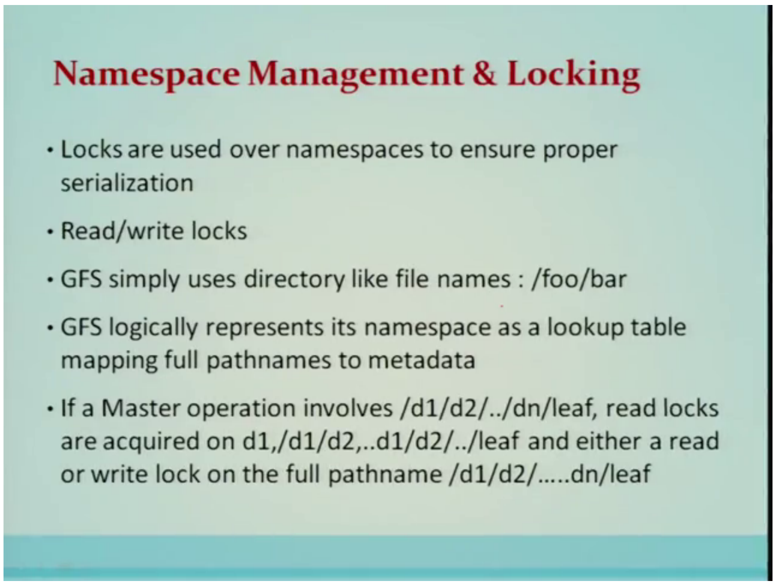


Record Append Algorithm (2)

7. If record fits, then the primary:
    - appends the record,
    - tells secondaries to do the same,
    - receives responses from secondaries,
    - and sends final response to the client. e data to all locations.

Then the primary appends the record, tells the secondaries to do, receives the response from the secondaries, and send the final response to the client that is data to all the locations. So, it is same as almost same as the write operation, but it will be written at the end that is called the append operation.

(Refer Slide Time: 36:20)



Namespace Management & Locking

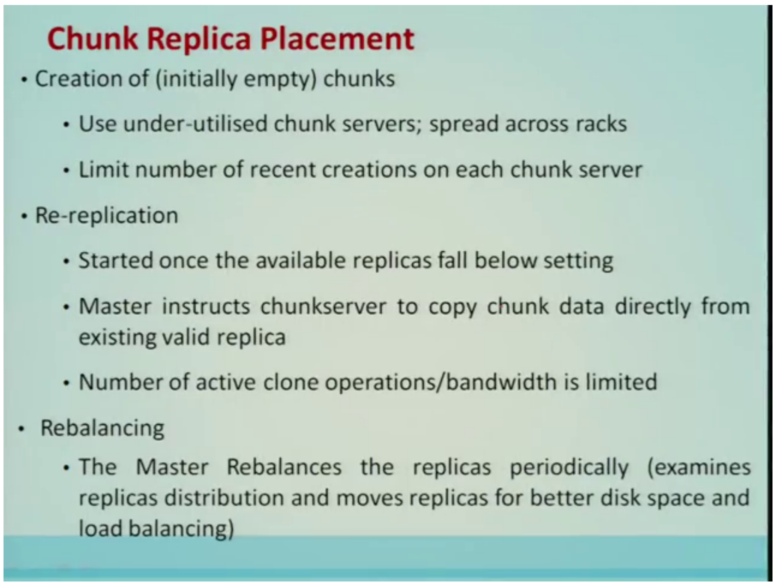
- Locks are used over namespaces to ensure proper serialization
- Read/write locks
- GFS simply uses directory like file names : /foo/bar
- GFS logically represents its namespace as a lookup table mapping full pathnames to metadata
- If a Master operation involves /d1/d2/../dn/leaf, read locks are acquired on d1,/d1/d2,..d1/d2/../leaf and either a read or write lock on the full pathname /d1/d2/.....dn/leaf

Master operation namespace management and locking locks are used over the namespace to ensure proper serialization. Read and write locks are used. GFS simply uses the directory like file names. GFS logically represents this namespaces a lookup table mapping full path name to the metadata.

So, if a master operates operation involves a file name, a path name, read locks are acquired on all these path names and either the read or a write lock on a full path name is being applied.

So, let me mention that the file name and its index is stored not in a form of a directory, but in a form of the lookup table which is being hashed. So, using this particular change the namespace lookup is very very efficient here in Google file system.

(Refer Slide Time: 37:20)

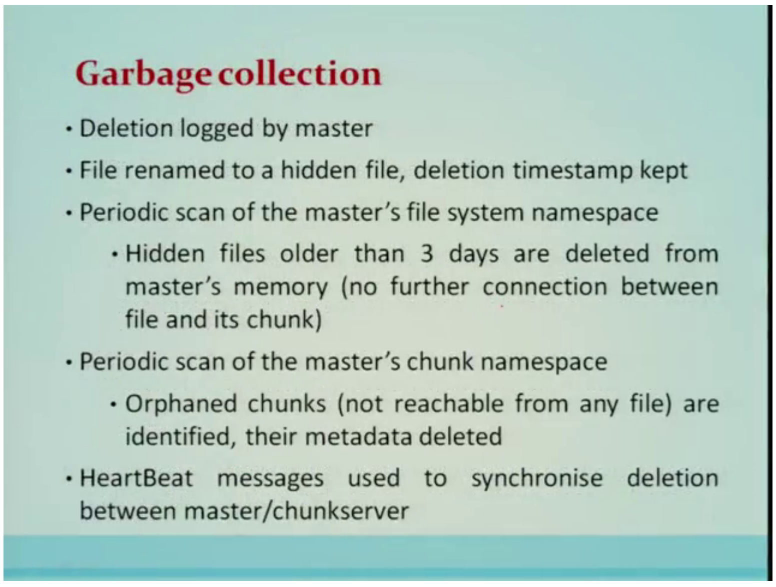


Chunk Replica Placement

- Creation of (initially empty) chunks
  - Use under-utilised chunk servers; spread across racks
  - Limit number of recent creations on each chunk server
- Re-replication
  - Started once the available replicas fall below setting
  - Master instructs chunkserver to copy chunk data directly from existing valid replica
  - Number of active clone operations/bandwidth is limited
- Rebalancing
  - The Master Rebalances the replicas periodically (examines replicas distribution and moves replicas for better disk space and load balancing)

So, chunk replica placement creation in each of initially empty chunks. Use the under-utilized chunk servers; which are spread across the racks and re application that is started once available the replicas fall below the setting. So, re application is performed. So, that. So, that all the empty servers are properly utilized. Rebalancing the master rebalance is the replica periodically examines the distribution and moves the replica for a better disk space and load balancing approach.

(Refer Slide Time: 37:54)

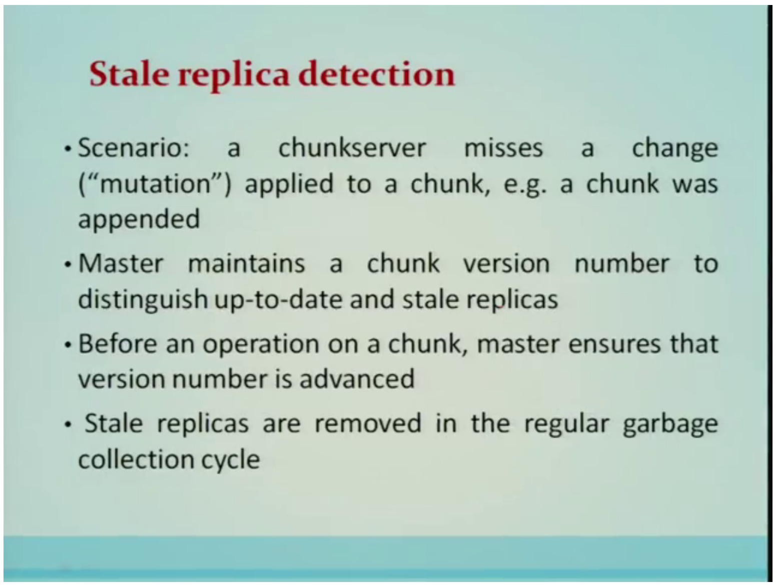


Garbage collection

- Deletion logged by master
- File renamed to a hidden file, deletion timestamp kept
- Periodic scan of the master's file system namespace
  - Hidden files older than 3 days are deleted from master's memory (no further connection between file and its chunk)
- Periodic scan of the master's chunk namespace
  - Orphaned chunks (not reachable from any file) are identified, their metadata deleted
- HeartBeat messages used to synchronise deletion between master/chunkserver

Garbage collection deletion logged by the master. So, the file rename to the hidden file, deletion timestamp kept. So, all these particular provisions are there for garbage collection why, because if a particular chunk is failed, then basically another chunk is already in place. So, those failed chunk will become garbage. So, these garbage will be periodically scanned and checked periodic scan of masters chunk namespace is also done.

(Refer Slide Time: 38:32)

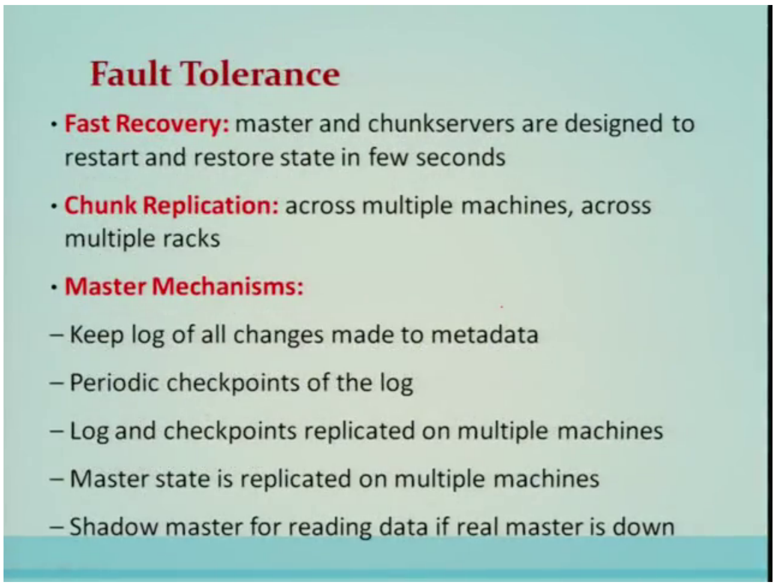


Stale replica detection

- Scenario: a chunkserver misses a change ("mutation") applied to a chunk, e.g. a chunk was appended
- Master maintains a chunk version number to distinguish up-to-date and stale replicas
- Before an operation on a chunk, master ensures that version number is advanced
- Stale replicas are removed in the regular garbage collection cycle

So, it Stale replica detection using this scanning is all already done.

(Refer Slide Time: 38:40)

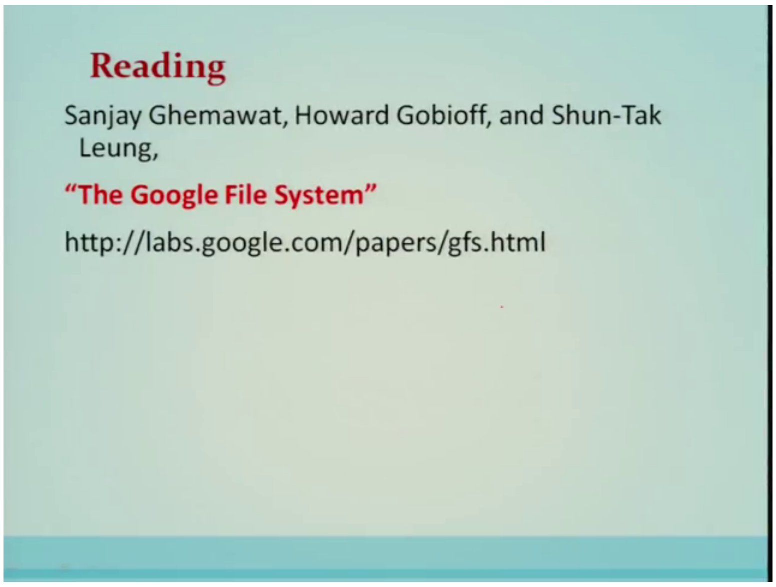


Fault Tolerance

- **Fast Recovery:** master and chunkservers are designed to restart and restore state in few seconds
- **Chunk Replication:** across multiple machines, across multiple racks
- **Master Mechanisms:**
  - Keep log of all changes made to metadata
  - Periodic checkpoints of the log
  - Log and checkpoints replicated on multiple machines
  - Master state is replicated on multiple machines
  - Shadow master for reading data if real master is down

Fault tolerance so this is also very important aspect in Google file system. So, fault tolerance is achieved with a fast recovery, chunk replication and master mechanisms. Fast recovery master and chunk servers are designed to restart restore in a few seconds. Chunk replication across multiple machines, across multiple racks is being made. So, master mechanisms keeps log of all changes made to the metadata, periodic checkpoints of the locks are maintained log and checkpoints are replicated on the multiple machines. So, whenever a master state is replicated on a multiple machines shadow master for reading data if the master is particular down.

(Refer Slide Time: 39:22)



Reading

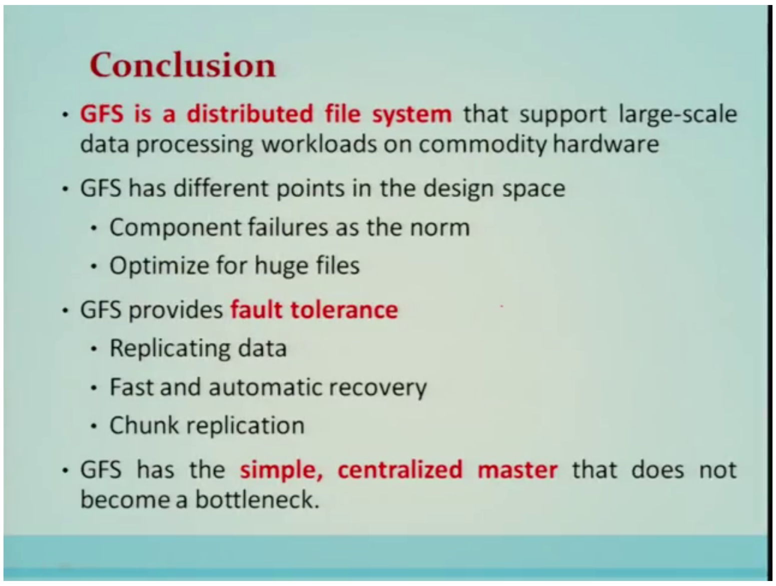
Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung,

"The Google File System"

http://labs.google.com/papers/gfs.html

Now, for further reading on this particular topic you can refer to them to the paper by Sanjay Ghemawat; the Google file system which is available at the Google link.

(Refer Slide Time: 39:38)



Conclusion

- **GFS is a distributed file system** that support large-scale data processing workloads on commodity hardware
- GFS has different points in the design space
  - Component failures as the norm
  - Optimize for huge files
- GFS provides **fault tolerance**
  - Replicating data
  - Fast and automatic recovery
  - Chunk replication
- GFS has the **simple, centralized master** that does not become a bottleneck.

Conclusion: GFS is a distributed file system that supports large-scale data processing workloads on commodity hardware. GFS has different points in the design space: component failures are a norm and optimize for a huge files or a large data sets operation or a computation. GFS provides fault tolerance by replicating the data, fast and automatic recovery, and doing the chunk replication. So, GFS has made a very simple, centralized master that does not become a bottleneck in this particular problem.

Thank you.