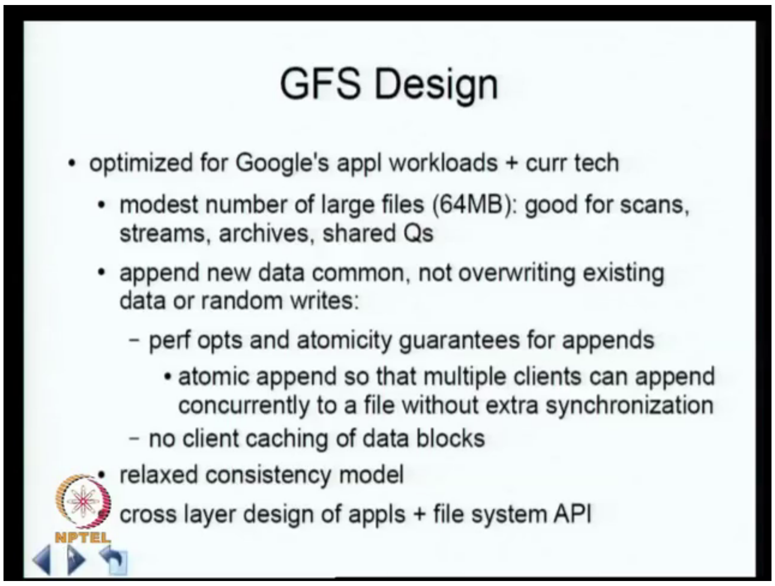
Storage Systems

Dr. K. Gopinath
Department of Computer Science and Engineering
Indian Institute of Science, Bangalore

Lecture – 41
GFS functions and operations

Welcome again to a NPTEL course on storage systems. So, the last 2 classes we are looking at the Google file system.

(Refer Slide Time: 00:38)

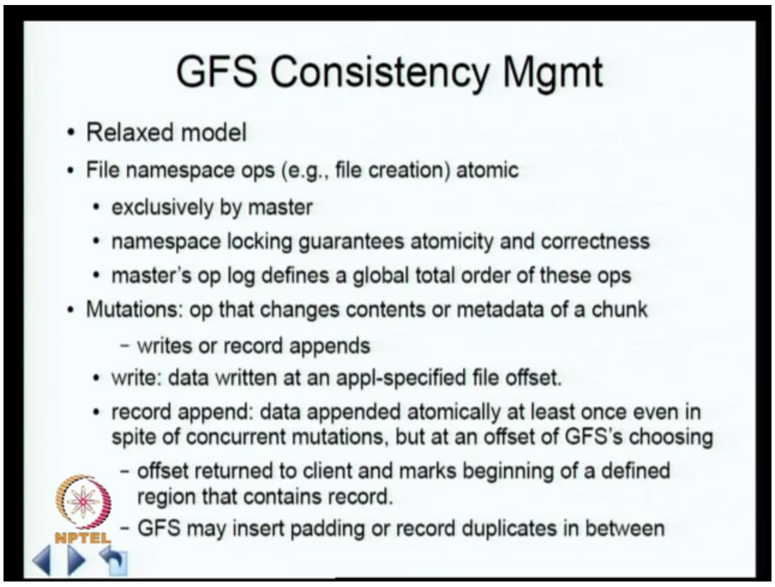


GFS Design

- optimized for Google's appl workloads + curr tech
  - modest number of large files (64MB): good for scans, streams, archives, shared Qs
  - append new data common, not overwriting existing data or random writes:
    - perf opts and atomicity guarantees for appends
      - atomic append so that multiple clients can append concurrently to a file without extra synchronization
    - no client caching of data blocks
  - relaxed consistency model
  - cross layer design of appls + file system API

And basically, in the first part of this we looked at the GFS high level design.

(Refer Slide Time: 00:41)

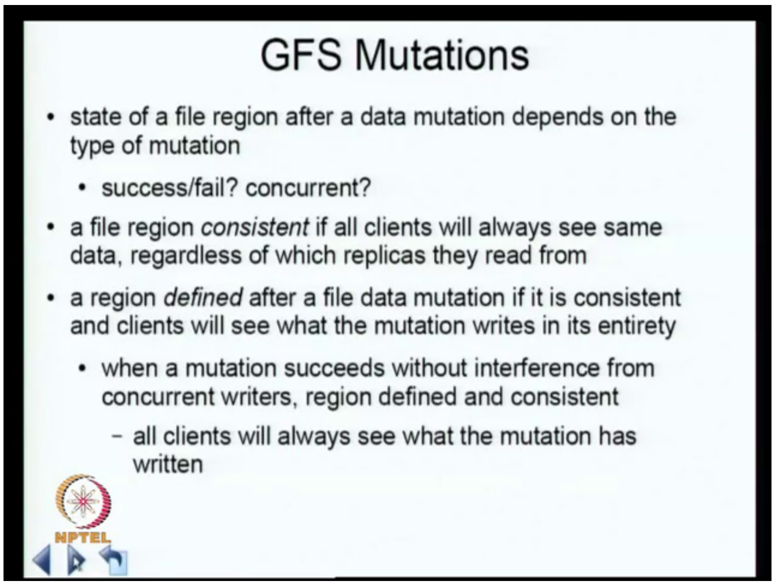


GFS Consistency Mgmt

- Relaxed model
- File namespace ops (e.g., file creation) atomic
  - exclusively by master
  - namespace locking guarantees atomicity and correctness
  - master's op log defines a global total order of these ops
- Mutations: op that changes contents or metadata of a chunk
  - writes or record appends
  - write: data written at an appl-specified file offset.
  - record append: data appended atomically at least once even in spite of concurrent mutations, but at an offset of GFS's choosing
    - offset returned to client and marks beginning of a defined region that contains record.
    - GFS may insert padding or record duplicates in between

And in the second part of it we looked at the consistency management.

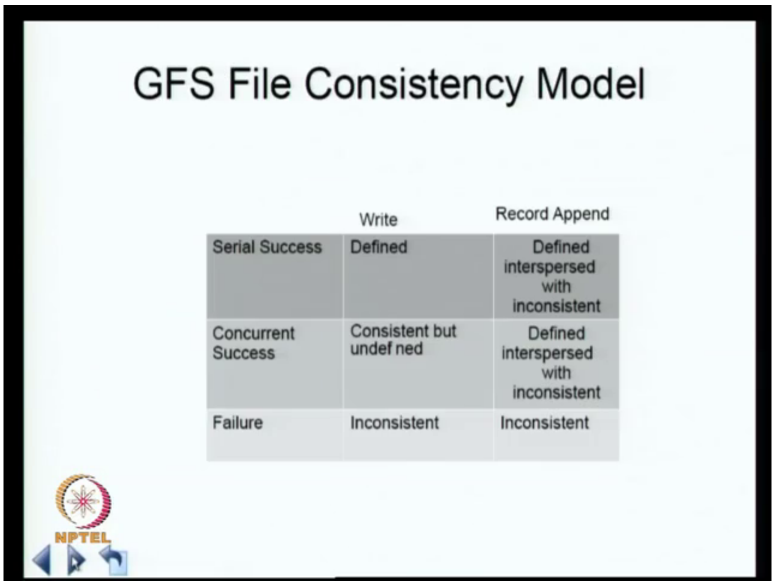
(Refer Slide Time: 00:52)



GFS Mutations

- state of a file region after a data mutation depends on the type of mutation
  - success/fail? concurrent?
- a file region *consistent* if all clients will always see same data, regardless of which replicas they read from
- a region *defined* after a file data mutation if it is consistent and clients will see what the mutation writes in its entirety
  - when a mutation succeeds without interference from concurrent writers, region defined and consistent
    - all clients will always see what the mutation has written

(Refer Slide Time: 00:54)



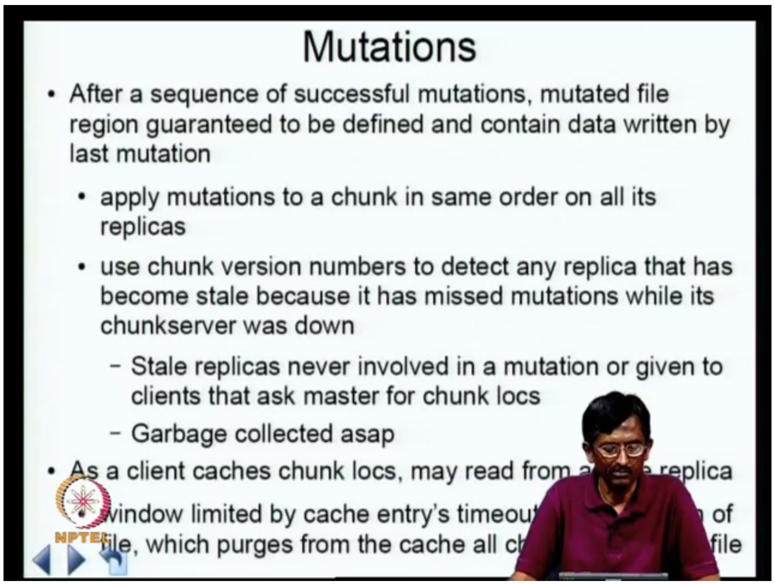
GFS File Consistency Model

	Write	Record Append
Serial Success	Defined	Defined interspersed with inconsistent
Concurrent Success	Consistent but undef ned	Defined interspersed with inconsistent
Failure	Inconsistent	Inconsistent

And basically, we went over some aspects of this model. Basically, that you might have what is called either a defined situation respect to a file or it could be consistent, but not defined undefined.

So, basically because of concurrent rate operations. Different chunks could be written in different even though they are getting returned to all the replicas in the same way. The actual chunks may be written in different values. So, that is why reason why it can be consistent, but not defined of course, it is a failure we can have inconsistency. And similarly, with record appends also, we have a similar situation. Once a succeed it is defined, but it might be that there are some earlier parts at lower offsets, which are some stuff that was put in, but could not be carried forward. Because some chunks some chunks are was field they are they could not complete that part of right. So, that is basically what this is.

(Refer Slide Time: 02:12)

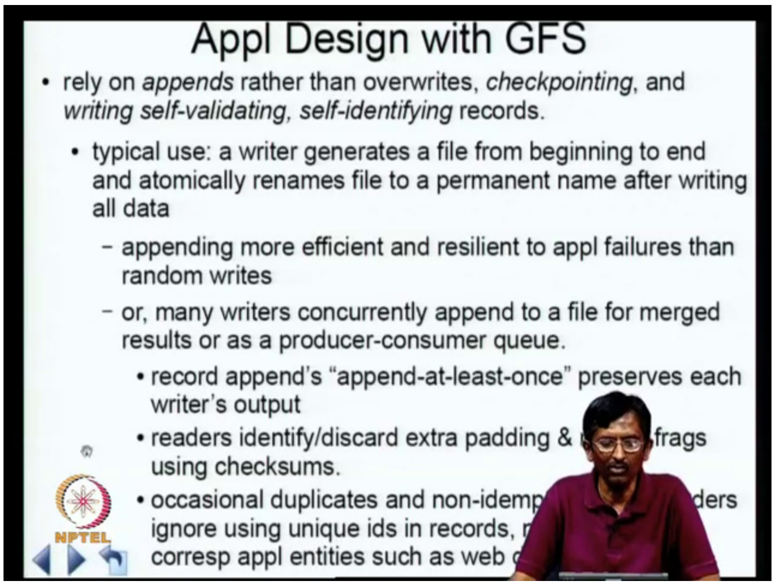


Mutations

- After a sequence of successful mutations, mutated file region guaranteed to be defined and contain data written by last mutation
  - apply mutations to a chunk in same order on all its replicas
  - use chunk version numbers to detect any replica that has become stale because it has missed mutations while its chunkserver was down
    - Stale replicas never involved in a mutation or given to clients that ask master for chunk locs
    - Garbage collected asap
- As a client caches chunk locs, may read from a replica
  window limited by cache entry's timeout of file, which purges from the cache all c file

So, went through some of this last time.

(Refer Slide Time: 03:16)

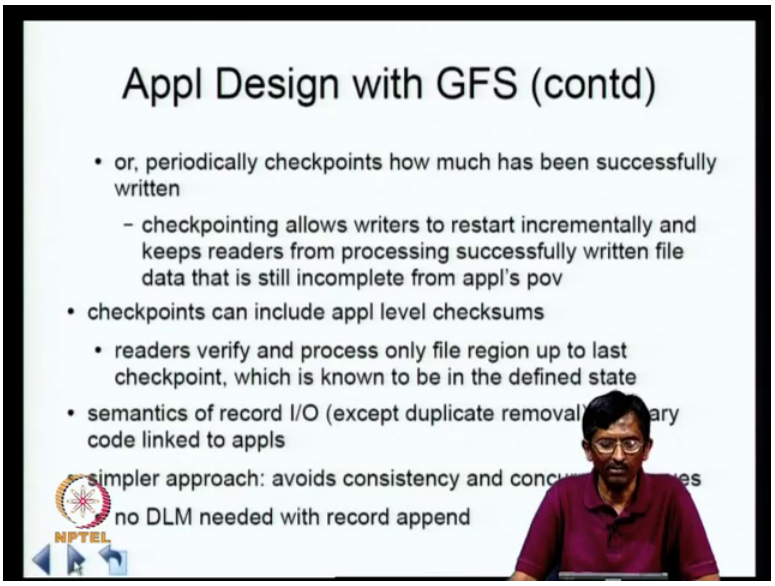


Appl Design with GFS

- rely on *appends* rather than overwrites, *checkpointing*, and *writing self-validating, self-identifying* records.
  - typical use: a writer generates a file from beginning to end and atomically renames file to a permanent name after writing all data
    - appending more efficient and resilient to appl failures than random writes
    - or, many writers concurrently append to a file for merged results or as a producer-consumer queue.
      - record append's "append-at-least-once" preserves each writer's output
      - readers identify/discard extra padding & frags using checksums.
      - occasional duplicates and non-idemp ders ignore using unique ids in records, corresp appl entities such as web

And we also looked at a bit about how the application design has to change a bit.

(Refer Slide Time: 02:23)

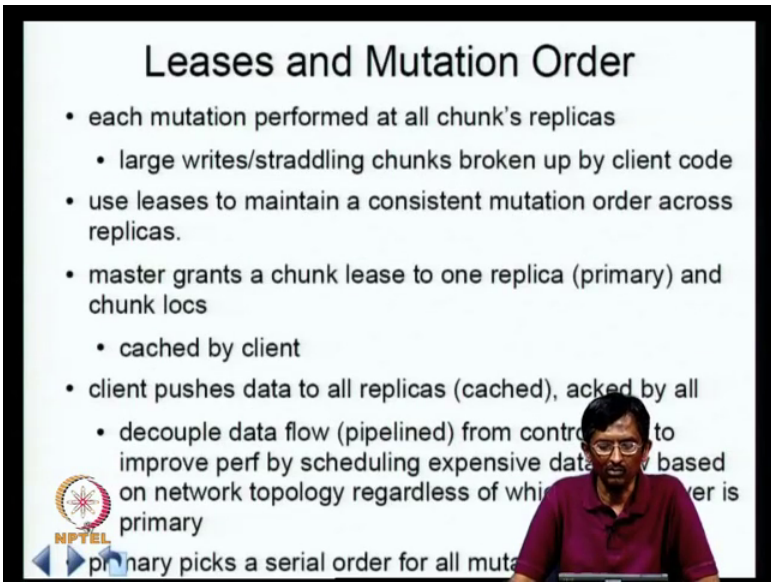


Appl Design with GFS (contd)

- or, periodically checkpoints how much has been successfully written
  - checkpointing allows writers to restart incrementally and keeps readers from processing successfully written file data that is still incomplete from appl's pov
- checkpoints can include appl level checksums
  - readers verify and process only file region up to last checkpoint, which is known to be in the defined state
- semantics of record I/O (except duplicate removal) ary code linked to appls
- simpler approach: avoids consistency and conc es
  - no DLM needed with record append

And thus, good thing about this particular design is that if you use appends you do not really need a heavy duty distributed lock manager. So, basically you try to avoid the costly consistency and concurrence issues by is slightly different simpler method.

(Refer Slide Time: 02:48)

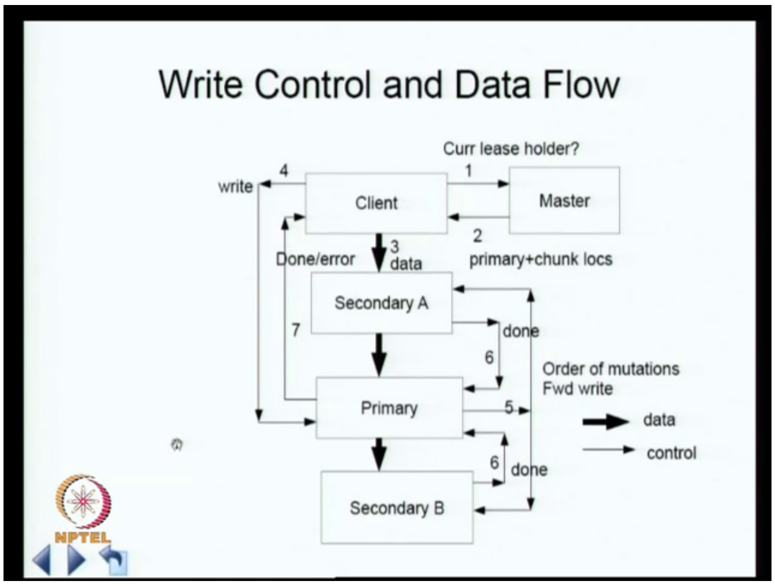


Leases and Mutation Order

- each mutation performed at all chunk's replicas
  - large writes/straddling chunks broken up by client code
- use leases to maintain a consistent mutation order across replicas.
- master grants a chunk lease to one replica (primary) and chunk locs
  - cached by client
- client pushes data to all replicas (cached), acked by all
  - decouple data flow (pipelined) from contr to improve perf by scheduling expensive data based on network topology regardless of whi er is primary
- primary picks a serial order for all muta

So, again we looked at this write control.

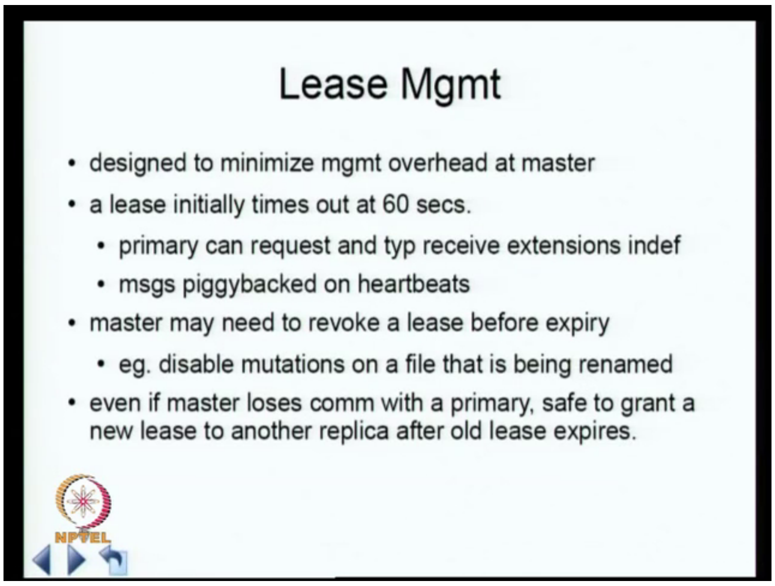
(Refer Slide Time: 02:52)


Write Control and Data Flow

And the data flow and basically what you mentioned was to ensure that you have you can exploit the bandwidth in the system. As soon as the client figures out, where the chunk locations are where the where the primaries on their chunk locations are. It can start pushing it immediately. And then once all the locations primaries and secondaries report that they got the data, then a client actually does the actual act of committing the writes. Because all the data will be actually sitting in some temporary area. And then once the write starts, then the primary will decide on a particular order. And then all the after all the mutations that are going on the same time if all the writes are taking place. And then each of these secondaries will actually write those things in the same order.

So, this design is slightly different from the ones we saw before with respect to good (Refer Time: 04:06) systems where the bandwidth issue is not taken that seriously. Because these are large scale file system, massive as system is bandwidth issue is an important form.

(Refer Slide Time: 04:19)



Lease Mgmt

- designed to minimize mgmt overhead at master
- a lease initially times out at 60 secs.
  - primary can request and typ receive extensions indef
  - msgs piggybacked on heartbeats
- master may need to revoke a lease before expiry
  - eg. disable mutations on a file that is being renamed
- even if master loses comm with a primary, safe to grant a new lease to another replica after old lease expires.

So, basically there is way want to continue from here, as I mentioned earlier, there could be a for each chunk there could be a current lease holder. And you want to check, if you want to do a write operation, you want to check who the who is the party has got the lease. And once you get the lease holder that is the primary, and then other people would other replica locations. They will have the chunk replicas, and that service the client gets.

Now, there is some aspect relating to the lease management also. Usually you want to minimize management overhead at the master. And the way the do it is by having a initial lease only for 60 seconds. So, if there are requests that come in within the 60 seconds. So, you do not have to go to the master. 60 seconds was considered to be a reasonably good amount of time. Again, we will see that most file systems. Typically, have leases around same amount of time 30 seconds or multiples of 10 seconds. This terms about the steady longer 60 seconds is it looks a bit steady longer, but I think this makes sense because, these operations are also take longer. Because we are dealing with large 64-megabyte chunks.

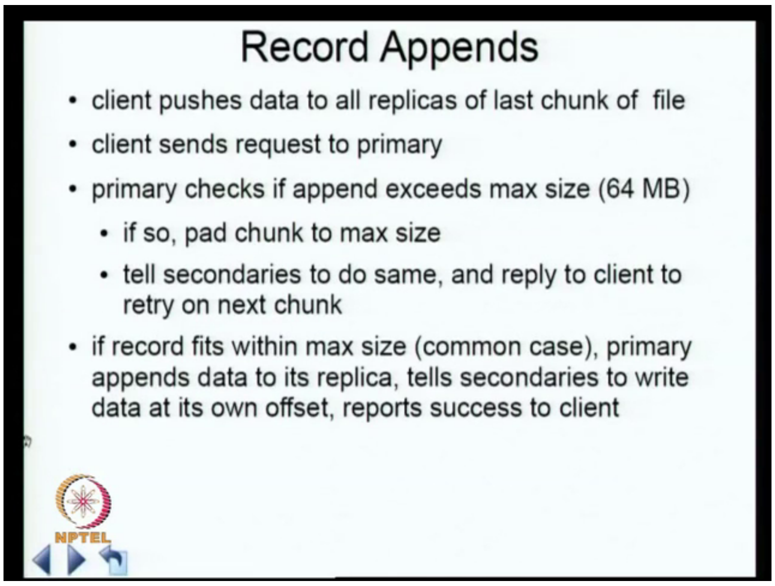
And the primary can request and typically receive extensions indefinitely. As long as it is where actively using it can keep on doing it. And there is not much extra overhead because anyway you are going to use heartbeats for checking the health of the chuck servers. And then basically any lease request can be sent as part of them heartbeats itself. Took basically the you can keep telling that I am alive, but the same time saying that in case you have to get any leases, you can put that in the same (Refer Time: 06:19) send it out.

And basically, you can the master also has some control over things. It can revoke a lease before expiry for example, if there is a file that is getting renamed, you want to prevent mutations on that file, while these misconcurrent operation is going on. You want avoid that concurrent operation. And therefore, you can essentially take back the lease. Once the lease is taken back, then there is no way anybody can modify that particular file. Again, one advantage of lease management is that is the leases is that even if master loses communication with a primary. It is safe to grant a new lease to another replica after old lease expires is that is basically the standard reason why you use leases. Basically, it is communication through time rather than communication through messages.

So, but this requires that once you have communication through time; that means, you need to have time synchronization as a critical part of this. So, it has to be reasonable, but it turns out that in this particular system, they do cache metadata for some pits of time. So, they can inconsistent that are there anyway are going to be here with respect to metadata, here is some problem again this is similar to what happens in nfs where similar ratios arise.

So, you know the time synchronization is not very, very accurate, it is still manageable. Because the semantics anyway assumes that there is a possibility of looking at still metadata for some period of time.

(Refer Slide Time: 08:12)



Record Appends

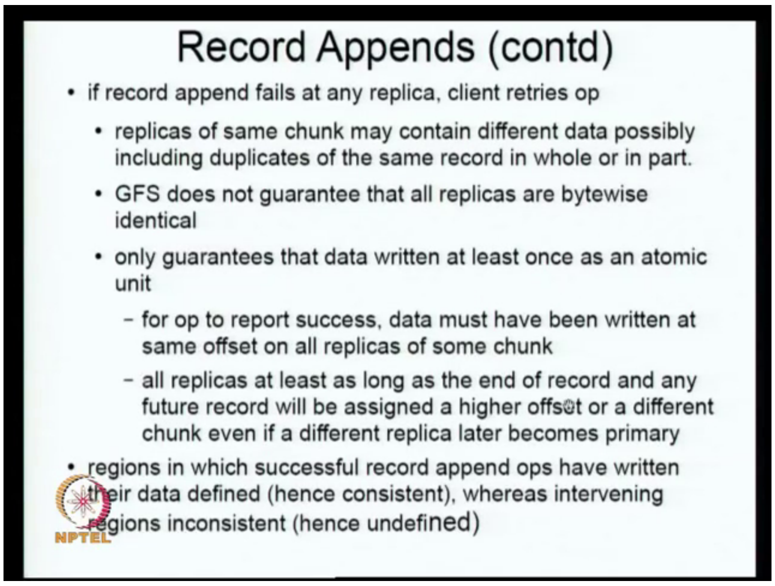
- client pushes data to all replicas of last chunk of file
- client sends request to primary
- primary checks if append exceeds max size (64 MB)
  - if so, pad chunk to max size
  - tell secondaries to do same, and reply to client to retry on next chunk
- if record fits within max size (common case), primary appends data to its replica, tells secondaries to write data at its own offset, reports success to client

Just like the writes, you can also do the record appends. It is very similar, only thing is that you have to worry about the last chunk of the file. In the case of a write it can be spanning multiple chunks and you have to split it. And the various aspect we that had to be done. There could be also let us say read modify write cycles etcetera, that is their part as part of the write. Whereas, in the record appends basically have to look at only the last chunk.

So, the client sends request to primary. Again, just like previously primary checks if append exceeds the max size. If so, pad chunk size pad the chunk to max size. And then it tells secondaries to do the same thing. It says you also I am going to pad it you also pad it. Because and then again it goes back to the client and says I have not done anything for you so far, but please retry it we will proceed after that. Because all it has done is to pad the chunks. Whereas, in the case of for the common case usually whatever is being written usually is fits into the maximum size.

So, it tells the secondaries to write data at it is own offset, and reports success to client, these are simpler case.

(Refer Slide Time: 09:45)



Record Appends (contd)

- if record append fails at any replica, client retries op
  - replicas of same chunk may contain different data possibly including duplicates of the same record in whole or in part.
  - GFS does not guarantee that all replicas are bytewise identical
  - only guarantees that data written at least once as an atomic unit
    - for op to report success, data must have been written at same offset on all replicas of some chunk
    - all replicas at least as long as the end of record and any future record will be assigned a higher offset or a different chunk even if a different replica later becomes primary
- regions in which successful record append ops have written their data defined (hence consistent), whereas intervening regions inconsistent (hence undefined)

Now you have to wonder about what happens if there is a failure. Record append fails at any replica client retries operation. So, that is how you may find that the replicas can diverge in the content of what they have replicas of same. Chunk may contain different data possibly including duplicate was same record in whole or in part. Basically some because of this issue GFS does not guarantee that all replicas are byte wise identical.

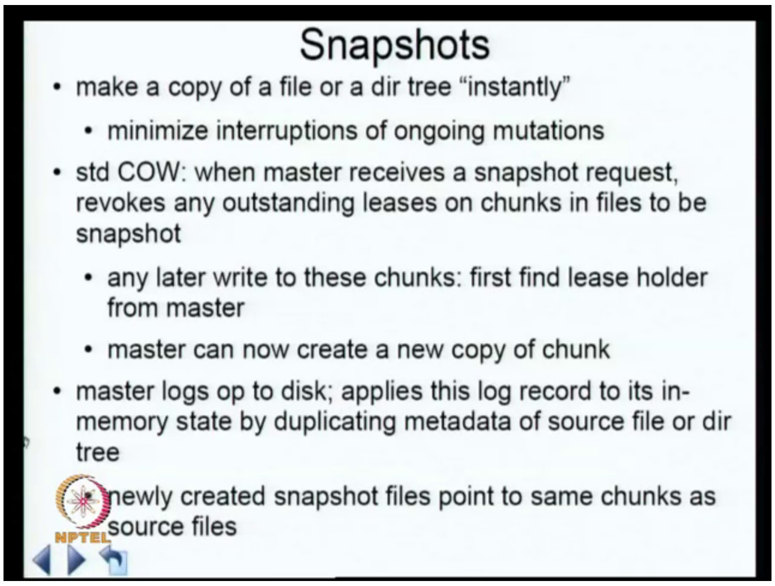
So, basically what happens is that you retry some guys have might succeeded some guys may not might not have succeeded. Those guys have succeeded have written something. And those guys have not succeeded have not written it completely or partially written it. So, they will be diverging. Again, when retry the whole operation. What will happen is that everybody gets a new offset the new offset, but whatever chunk that they are put in previously whatever has failed is still remains there. That is reason why GFS not guarantee all replicas are byte wise identical it also can happen because of some such chunk server is down. Or so, because it is down it is not able to take this particular, let us say data. And so, it can become stale. So, you have to have another protocol by which this tale replicas are detected.

So, what GFS somewhere and guarantees is that the data written at least once as an atomic unit. You are trying to obtain something it is giving you at least one semantics; that means, in a possibly written multiple times, but all the previous times are failure somewhere. And then, but finally, if it succeeds, there will be at least once it is returning the whole thing properly at all the replicas. The appends succeeded all the replicas. So, for op to report success data must have been written at same offset on all replicas of some chunk. All replicas at least as long as the end of record, and any future record will be assigned a higher offset or a different chunk even if a different replica later becomes primary. So, even if when the primaries change and whatnot we can guarantee, but that any future record will be assigned higher offset. Or it has to be it will be in a different chunk all together.

So, regions in successful record append op record append ops have written their data defined they become consistent. Whereas, intervening regions inconsistent. So, wherever the successful record appends have taken place, they will be first of all consistent, because all of them are writing in the same order. And order is fixed. And it is made sure that all the data is available to all the clients, all the primary secondaries only when everybody has got the data, then they start writing it. And they all write at the same offset. That is why it should be that particular region will be both defined. And consistent it is consistent, because they are writing the same things is defined because all of them also will have completed their thing. Otherwise it is not reported this success.

Now, as before this particular successful append. They could have they could have been some problems that is why the intervening regions are inconsistent. Some are written it some are not written it does have for old information some are new information.

(Refer Slide Time: 13:50)



Snapshots

- make a copy of a file or a dir tree “instantly”
  - minimize interruptions of ongoing mutations
- std COW: when master receives a snapshot request, revokes any outstanding leases on chunks in files to be snapshot
  - any later write to these chunks: first find lease holder from master
  - master can now create a new copy of chunk
- master logs op to disk; applies this log record to its in-memory state by duplicating metadata of source file or dir tree
  - newly created snapshot files point to same chunks as source files

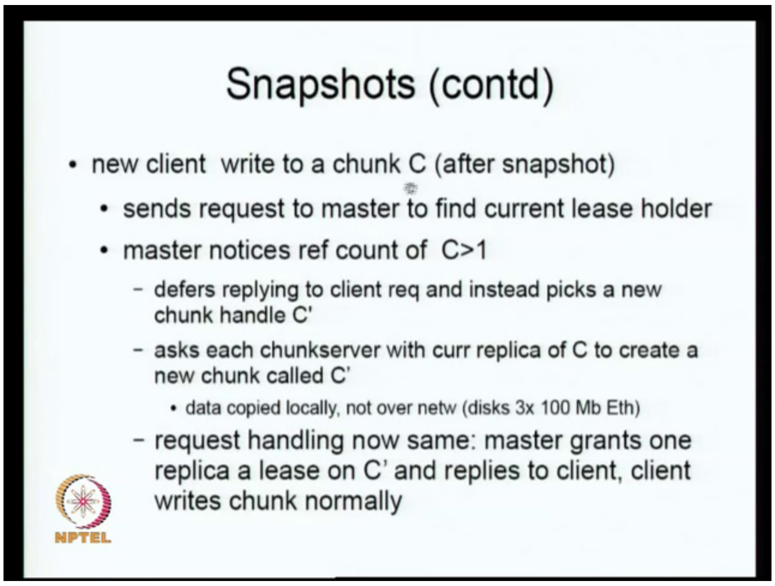
So, another thing that GFS does this is the provide something called snapshots. And the basically this is neither for standard reasons why snapshots are useful. You want to make a copy of a file or a directory tree. And it could be for backup, it could be for you want to make some modifications, and it is not clear that the complex modifications may go through successfully. So, you want to make a snapshot of it, and do it on a branch and later decide what to do with the update whether it is if it succeeds.

So, because you are going to make a copy of file or a directory tree especially directory tree, you have to make sure that you have to minimize interruptions of ongoing mutations. This snapshot should not basically stop the ongoing mutations, ongoing changes. So, basically standard copy on write can a mechanism. When a master receives a snapshot request. It revokes any outstanding leases on chunks in files to be snapshot. Basically, if you have going to snapshot, you want to make sure that no client is actually writing to it therefore, you basically try to get back all the leases. So, that you it is clear for you to start doing a snapshot.

So, if there is any later write to these chunks, first find the lease holder from master. And then master can now create a new copy of chuck. So, after any covers taken place, after covers taken place if there is any later write to these chunks. You first find the lease holder from master get a lease because the client has to find the lease holder from master. So, master will actually allocate a new lease for it and then the master can now create a new copy of chunk.

Now, the master also logs the operations to disk. First it applies this log record to it is in memory state, by duplicating metadata off source file or directory tree. Basically, in snapshot you have a source and the destination. So, basically you duplicate the metadata of the source file, or the directory first. And then actually you log all these things to disk. Just like in any standard snapshot newly created snapshot files, point to same chunks as source files. Once the update happens then they will be that connection this is broken.

(Refer Slide Time: 16:41)



Snapshots (contd)

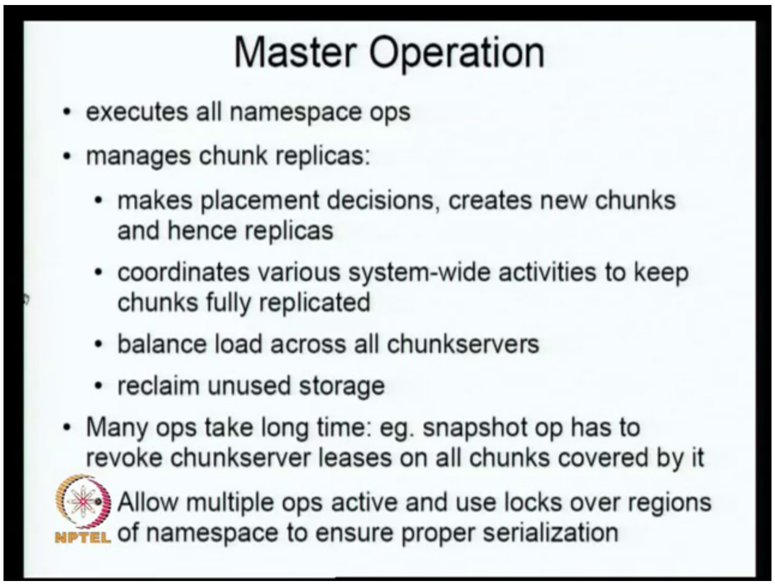
- new client write to a chunk C (after snapshot)
  - sends request to master to find current lease holder
  - master notices ref count of C>1
    - defers replying to client req and instead picks a new chunk handle C'
    - asks each chunkserver with curr replica of C to create a new chunk called C'
      - data copied locally, not over netw (disks 3x 100 Mb Eth)
    - request handling now same: master grants one replica a lease on C' and replies to client, client writes chunk normally

So, if you a new client was to write to a chunk c after snapshot, sends master to find current lease holder, right. Basically, a snapshot has been done, and there could be other things going on. And that is somebody wants to do a client write to chunk. So, sends request to master to find current lease holder, somebody else might be have in the current lease. Then the master notices the reference counter the chunk being greater than 1.

So, if the master notices that the reference count is greater than 1, then it knows so that it is a snapshot it is connected with snapshots. So, instead going back to the client immediately, it picks a new chunk handle c prime. And then ask each chunk server with current replica of c to create a new chunk called c prime. Because now the write is going to go to this c prime and they have some interesting optimizations they make sure that the data is copied locally not over network. In their systems when was designed this for certainly far more faster than ethernet. That is why it is done locally not over the network.

So now why this has been done? The request handling again is the same as master grants one replica a lease on c prime. And now c prime is write on c prime. Very difference in the beginning when the client wrote there was this chunk c, which was higher chunk count of greater than 1; that means, it was also part of that. Same chunk was part of 2 views one of the old file system on the new of the snapshotted file system. And because the reference counts greater than 1, you had to allocate a new chunk c prime, and now the client actually can write to c prime directly. Now that that particular chunk has been decoupled from the previous file system.

(Refer Slide Time: 18:57)



Master Operation

- executes all namespace ops
- manages chunk replicas:
  - makes placement decisions, creates new chunks and hence replicas
  - coordinates various system-wide activities to keep chunks fully replicated
  - balance load across all chunkservers
  - reclaim unused storage
- Many ops take long time: eg. snapshot op has to revoke chunkserver leases on all chunks covered by it
- Allow multiple ops active and use locks over regions of namespace to ensure proper serialization

Now, this let us just look at all the one standard what are the master operation. I think as we discussed so far. It executes all namespace operations. For example, if it question of adding a file, deleting a file, deleting a directory, etcetera. The master also manages all the chunk replicas, makes placement decisions. Because it is the only single entity which knows all about the whole system. It can make all this fairly quickly and everything is done using information in memory. So, it is quite fast also it also creates new chunks and also new replicas as necessary.

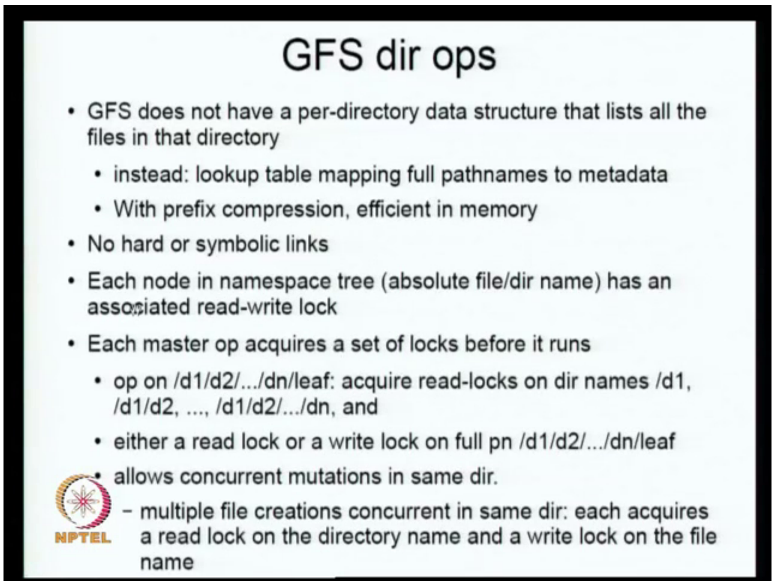
So, if the chunks sometimes if there is a replication factor of some amount, and some of the chunk servers may go down. So, it may turn out that the number of replicas may not be as much as necessary. So, it does some system wide activities to make sure that, the replication level again goes up those kind of things also it does it has to balance load across all chunk servers and also reclaim unused storage.

Now, one thing of interesting about this particular file system we will discuss again later is that the deletes are not done eagerly; that means, as soon as you tell a delete you do not immediately reclaim this space. It is done lazily. So, idea being here is that you have a scan of the whole system at regular intervals the rather for housekeeping operation during that time, you notice those deleted files and their corresponding chunks, then you reclaim you essentially do some current garbage collection. So, they have a garbage collection face is does that.

So, that we will discuss this part of it brief later sometime. So, again it turns out there are some operations take a long time, and just like a new standard file system and concurrent file system, you take basically locks over regions of the file system. If you want to get concurrent operations on multiple parts of file system, there has to be locks taken on different portions in the file system that is what. This also does example snapshot takes a long time, and it has to revoke all the leases, it will take some time.

So, if you know if you are taking or if you are not taking for concurrency, you need to ensure that other party is not connected with this snapshot operation they should able to proceed, but it should be able to stop those guys who are encroaching on the part with the snapshot register. So, you need to take a lock the correspond. So, want allow multiple operations to be active at the same time. And therefore, you need to use multiple locks to over multiple regions to ensure proper serialization.

(Refer Slide Time: 22:18)



GFS dir ops

- GFS does not have a per-directory data structure that lists all the files in that directory
  - instead: lookup table mapping full pathnames to metadata
  - With prefix compression, efficient in memory
- No hard or symbolic links
- Each node in namespace tree (absolute file/dir name) has an associated read-write lock
- Each master op acquires a set of locks before it runs
  - op on /d1/d2/.../dn/leaf: acquire read-locks on dir names /d1, /d1/d2, ..., /d1/d2/.../dn, and
  - either a read lock or a write lock on full pn /d1/d2/.../dn/leaf
  - allows concurrent mutations in same dir.
    - multiple file creations concurrent in same dir: each acquires a read lock on the directory name and a write lock on the file name

So, let just look at some aspects relating to this directory operations. One thing about this particular file system which is somewhat different from a regular file system, like for example, ext 3 file system in Linux. GFS does not have a per directory data structure, that lists all the names in that file in the directory. So, all that has is a lookup table mapping full pathnames to metadata. So, because of that it turns out a per directory data structure. So, you cannot go to a directory and say list all the file.

So, it uses prefix compression to store the names. That is why sufficient unit. Basically, idea here is this that suppose you have a file d 1 d 2 dn finally, the leaf node. This whole thing is thought of a single name. In a regular file system, we first go to d 1 locate the directory structure corresponds to d 1, and then search for d 2 in it d 2 is there then again you search for something etcetera, right. Here the whole thing is looked up in one shot.

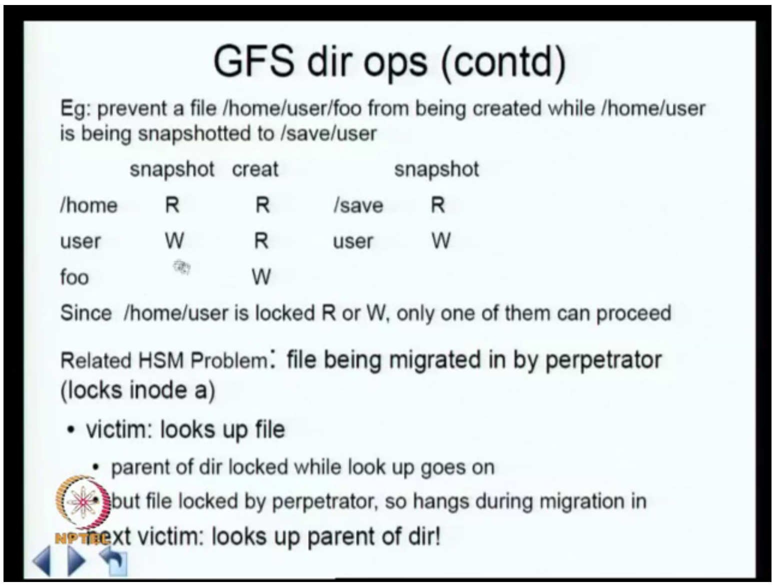
So, there is at lookup table. Full pathnames are mapped to meta directory; that means, you give it this it just goes under some hashing or something immediately finds a corresponding entry and it gets sign out equal I know. And because of this it does not have any hard or symbolic links. It turns out when you are doing this kind of stuff, you can not have hard or symbolic. Because if you want to have hard links, then you need to have the notion of someplace where it each of those new names for these hard links are present in a particular directory, you need to have those kind of notions. This one is not there. Because it does not have a per directory data structure.

So, each node in namespace tree, where there is absolute filename or directory name has also a read write lock. Because you want to allow a lot of concurrency, and normally when you have a tree structured file system like this; given slash etcetera, some operations down below can actually cause some condition the top. Where look at it is own why that is it is. So, it has both read and write lock. Read lock is use so that nobody can change it. Whereas write lock is 2 can you got the only person who can change it both things were taken.

For example, if you are doing an operation on slash d 1 d 2 dn df sorry leaf, then you take all the read locks up to this point. And depending of operation want to take a write lock on the full pathname. So, you take read locks only on that each of the directory names. So, this allows concurrent mutations in the same directory. So, if you are doing for example, multiple file creations in the same directory, you can acquire a read lock on the directory name. And the write lock on each other file names, and that is perfectly fine because basically all you are saying is you are not going to change the pathname. It is intact as it is, but you going to change the file. That is why you going to take a write lock on that what is that in the case of a regular file system. You need to take a each party has to take a lock on dn. Because they are going to add something to the directory. You modify something in the file you are adding a create for example you are doing a create for example, you are going to add something per directory.

So, if you want to the add to the directory that has to be exclusively written, the directory itself has to be written. That means, that if you want to do multiple file creations, they will have to take a write lock on the directory entry also. Where is in this case there is basically taking a read lock on the directory name. And a write lock on the file name. So, this allows for a slightly amount of concurrency, but then the thing is that they have do a you need to have a map which maps full pathnames to metadata. Unless you use tricks like prefix compression it is going to be efficient.

(Refer Slide Time: 27:18)



GFS dir ops (contd)

Eg: prevent a file /home/user/foo from being created while /home/user is being snapshotted to /save/user

	snapshot	creat		snapshot
/home	R	R	/save	R
user	W	R	user	W
foo		W		

Since /home/user is locked R or W, only one of them can proceed

Related HSM Problem: file being migrated in by perpetrator (locks inode a)

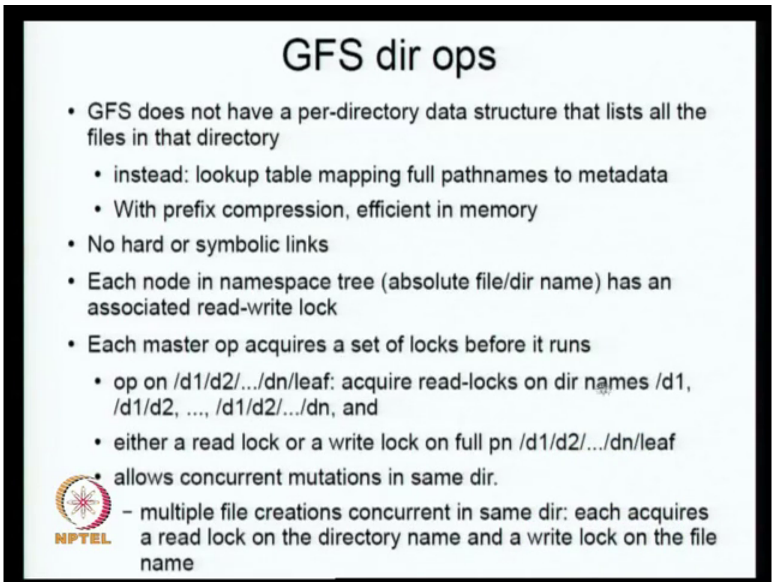
- victim: looks up file
  - parent of dir locked while look up goes on
  - but file locked by perpetrator, so hangs during migration in
- next victim: looks up parent of dir!

Let us look at one example of how they achieve this one. I will also relate this 2 a different problem and some other file systems. Suppose I have the following situation. I am going to do some snapshotting, and then while this is being snapshot become of there is some slash home user is being snapshotted to slash save user. There is a the intent of this being of course, that somebody’s saving or backing up the current state of your user directory exactly as it existed, we just want to back up back that is being backed up to save user.

Now while this snapshotting is being done, I want to ensure that the user is not able to create a file. Let us say I want to do that. So, the way this is done here is the product. So, let us look at the directories slash home user and foo. So, if you want to create a snapshot you are supposed to take the your read lock on slash home, and a write lock on the user. Whereas, when you are doing a create, you want to create a foo file for example, what is going happen is that you are going to have a read lock on this, a read lock on this, because the idea is that nobody should be changing the pathname while I am creating a foo. That is why this is both are read and then this will be write.

So, suppose I look at the snapshot itself. In snapshot slash save should be r, because it is read only again the pathname should not be change that is why I am taking read on block. Whereas, I am modifying a user because I am copying whatever is here into this one that has to be write. A notice that for create or taking a read for snapshot you take an already a write, because you taken a snapshot on write. Nobody can get a read lock. Again, this if you think about it the idea is that when you are doing a snapshot, what did we say it was you are going to; let us see, where was this?

(Refer Slide Time: 29:54)



GFS dir ops

- GFS does not have a per-directory data structure that lists all the files in that directory
  - instead: lookup table mapping full pathnames to metadata
  - With prefix compression, efficient in memory
- No hard or symbolic links
- Each node in namespace tree (absolute file/dir name) has an associated read-write lock
- Each master op acquires a set of locks before it runs
  - op on /d1/d2/.../dn/leaf: acquire read-locks on dir names /d1, /d1/d2, ..., /d1/d2/.../dn, and
  - either a read lock or a write lock on full pn /d1/d2/.../dn/leaf
  - allows concurrent mutations in same dir.
    - multiple file creations concurrent in same dir: each acquires a read lock on the directory name and a write lock on the file name

So, if you are doing a an operation write, you are going to take a read locks on all these directory names. And then you take a read or a write lock on a full pathname all right. So, because of this when you think about it, you are going to create on user a read lock is required for the create that a snapshot to ensure that nobody else can modify it at the time it is being created you take a write lock, and the 2 things are let us say not compatible.

So, only one of them can proceed. So, if you have already started the snapshot nobody else can create a file.

Now, if you think about it, this solution essentially helps you to avoid some complicated condition problems. And this illustrate this with take an example from a regular file system, which does not have this notion of a full pathname to metadata mapping. And does not have per directory data structures. Suppose we have a system called hierarchical storage management. What is this? These are some are basically you can have a system, that has got some files which has stolen tape. You have some disk files and also on tape. And then once in a while when some files have become very old, you want to send it out to tape. And then when we require it, it is picked up from the tape and brought back.

So, suppose some guy wants a migrates somebody there is already a file in the tape now, when the migrate it inside to the disk. So, the party who is doing is what we call the perpetrator, this is a guy is going to bring it in to the disk. And to do that you basically have to lock that particular things to make sure there is no other concurrent operation on that object. It is not admit the disk, but you want to make sure that nobody else also attempt some more activity of that kind of whatever right. So, that is why you are taking a lock on it.

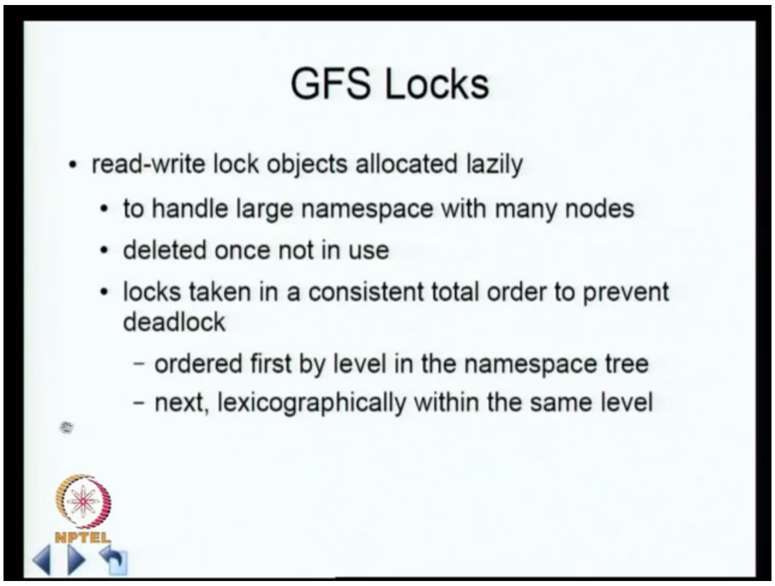
Now, while this is going on somebody you know centrally it tries to look it up, let us say. So, in conventional file systems when you are looking up a file you have to lock the directory where it is existing. Because otherwise somebody could concurrently delete it. For example, and then. So, therefore, you cannot know you do not know what answer to give them. So, basically when I looking up you try to lock the directory. So, if you lock the parent directory, while they look up is going on now the problem for us is that the file is getting migrated it is coming from tape it takes multiple seconds let us see 5 seconds whatever or even 50 seconds; that means, that the parent of directory is locked up for that amount of time. Because if this like the guy was trying to look it up cannot succeed till the migration has completed.

So, this guy has got a lock on the directory. And so, it keeps hanging there. If suppose somebody else tries to look up the parent of directory. Then that they also will hang and if you did just a few of the by sheer bad luck, if all of them tries to opt a slash 4 5 of them, right. If there is a the depth is only 3 4, let us say and the bad luck. So, migration is going on and each of the guys some there other 3 or 4 other processes which are looking up. Those comparison the directory path. And the whole system is essentially can hang and with whole system because slash is stuck and everybody is stuck now.

So, this kind of things can happen, costly because you are trying to take a lock and a directory per say. Because a directory has the notion of having those files and they have to be if you are updating anything you have to take a lock on a directory. Whereas, in this kind of systems, you are taking a lock on the pathname, not on some directory structure called directory. And that is why it is slightly easier it will be less problematic, but the problem with this solution is that you do not you do not have hard or symbolic links. Basically, because you can not have hard links because, hard link by definition has 2 different names for the same thing; that means, that you have to create a some way of hashing 2 different names to the same object which is not possible.

So, because you can create arbitrary a hard link. So, there is no arbitrary hash functions which can map 2 different names to the same thing, with arbitrary pathnames given. Therefore, you cannot have hard links. So, that means, that the kind of saving etcetera some kind of savings etcetera that you can do with regular file system are not possible. But the advantage of that is that your directory now can let us you do not need a directory per say, only pathnames you actually are locking on a pathname, rather than on a directories actual directory structure.

(Refer Slide Time: 35:48)

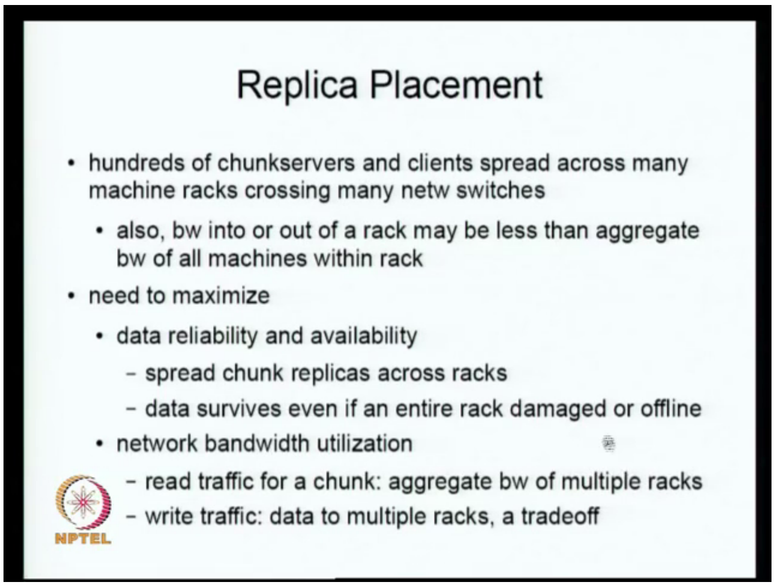


GFS Locks

- read-write lock objects allocated lazily
  - to handle large namespace with many nodes
  - deleted once not in use
  - locks taken in a consistent total order to prevent deadlock
    - ordered first by level in the namespace tree
    - next, lexicographically within the same level

So, basically this so; because you are taking a you are going to have some lock objects, and your read write. And there is a large namespace with many nodes, and you do not want to create a read write object is for each one of them. So, this read write lock objects are located lazily on demand. And if there not in use they are deleted. Again, locks are always taking in a consistent total order to prevent deadlock they basically standard the attempt at 2 phase locking kind of models. Ordered first by level in the namespace tree, next lexicographically within the same level. Basically, is the multiple people walking up and down the a directory namespace the namespace, you have to always have the notion that when you lock it you take it in same order. You start from the top go to the next one etcetera rather than some guys going up and some guys going down. If you do that you can deadlock situations, that also use there.

(Refer Slide Time: 37:07)



Replica Placement

- hundreds of chunkservers and clients spread across many machine racks crossing many netw switches
  - also, bw into or out of a rack may be less than aggregate bw of all machines within rack
- need to maximize
  - data reliability and availability
    - spread chunk replicas across racks
    - data survives even if an entire rack damaged or offline
  - network bandwidth utilization
    - read traffic for a chunk: aggregate bw of multiple racks
    - write traffic: data to multiple racks, a tradeoff

Now, coming to a few other aspects of the system. You also need to handle how to do a replica placement. Each of the single each of this is where in a let us lot of errors are there. I am just highlighting some aspects of a they have done. Now there are 100s of chunk servers and clients spread across many machine racks crossing many network switches.

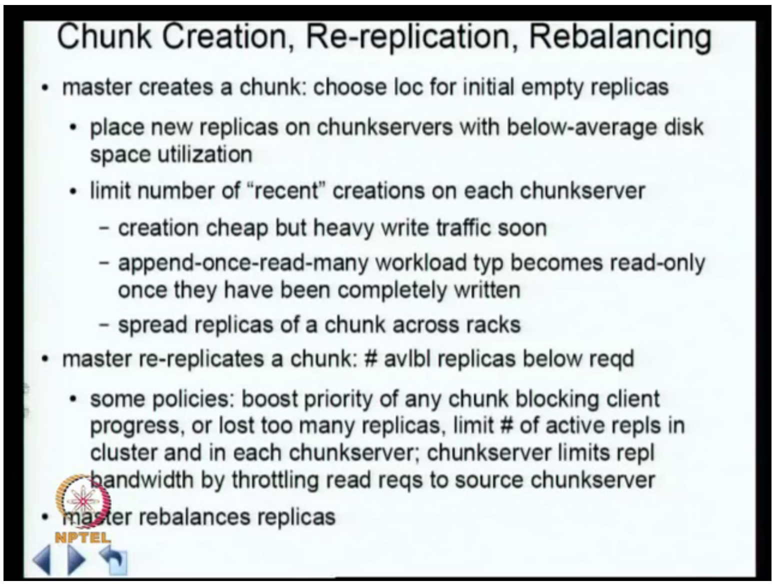
So, your how you place your replicas where actually determine how well you perform. Not only that there are issues like, we have bandwidth going into the rack or out of a rack. May be much less than the aggregate bandwidth all of all machines then rack. For example, if you are having something like forty machines on a single rack, or 20 machines. Then it turns out that the bandwidth between them can much higher than if you are again taking the let us say some network connection out of the rack. That will be basically the maximum possible the technology at that time for example, when this people did it because gigabit Ethernet.

So, out of the rack will probably be one gigabit. Where as if we have some 20 or 30 machines these are the with 100 megabit per second. Let us say that is certainly bigger than the one gigabit connection. So, because of this, your replica placement has to be managed in such a way that, you maximize data availability and reliability, and also network bandwidth utilization. And the way they do it is by making sure that this the chunk replicas are spread across the racks. There is some positive things about this. In similarly thinks about this first you think about spreading chunk replicas across racks is that the data survives even if an entire rack damaged or offline. In real big systems typically, there are many correlated failures. For example, one power supply in spite of redundancy in work node. The power supply can be problematic and the whole rack might be powered down.

So, in that case, if you do not spread your chunk replicas across racks, you might lose a whole the availability is compromised seriously that is why it has to be done across. Other nice thing about doing this is that if you want to do a read traffic for a to look at the if you want to do a read for a of a chunk. You have essentially now have the aggregate bandwidth of multiple racks. You are not actually fighting to get 3 chunk reads going to the same rack and all of them flowing out of the same connection from the rack to the client. You basically now have 3 different connections from 3 different racks which are coming to the client.

So, basically if it is read traffic for a chunk it is actually a good solution. Whereas, for write traffic it is a problem, because if I want to write to multiple racks, actually I have to send multiple copies to each other racks. I can not forward it from 1 play one machine inside the rack to another machine is same rack, I can not do it. So, there is a problem here with respect to write traffic, but again they have decided there is a proper trade off to make because they are usually more into heavy reads rather than heavy racks. Basically, again with the design optimization. Basically, depending on your the way you are using the file system you have to choose and they have decided that it is better for me to be able to exploit the aggregate bandwidth of multiple racks for reads then suffer it is the suffer with respect to write traffic having to make 3 independent writes with different racks. That will essentially reduce the bandwidth of other things because all the 3 racks are getting messed up with respect to the format.

(Refer Slide Time: 41:18)



Chunk Creation, Re-replication, Rebalancing

- master creates a chunk: choose loc for initial empty replicas
  - place new replicas on chunkservers with below-average disk space utilization
  - limit number of "recent" creations on each chunkserver
    - creation cheap but heavy write traffic soon
    - append-once-read-many workload typ becomes read-only once they have been completely written
    - spread replicas of a chunk across racks
- master re-replicates a chunk: # avlbl replicas below reqd
  - some policies: boost priority of any chunk blocking client progress, or lost too many replicas, limit # of active repls in cluster and in each chunkserver; chunkserver limits repl bandwidth by throttling read reqs to source chunkserver
- master rebalances replicas

So, there are quite a few other operations that have to be done. When you are doing chunk creation, you have to think about where to place the chunk for example, right during chunk creation this is a re replication rebalancing some of example a master creates a chunk. You have to choose locations for initial empty replicas. Look at the beginning before there is nothing be returned to the chunk. Somebody let us say is doing a append. So, you have to figure out where to choose it. So, there are various heuristics fairly straightforward place new replicas on chunk servers with below average disk space utilization that is first number, that is one kind of heuristic. At the same time, they want to ensure that limit number of recent creations on each chunk server. Basically, the issue is that the creation may be cheap. But if you create too many things just behind it all kinds of write traffic also will come. Because usually what they have in their kind of system, if append once read many basically the append once read many workload typical pattern. And once it has been completely returned, it becomes read heavy.

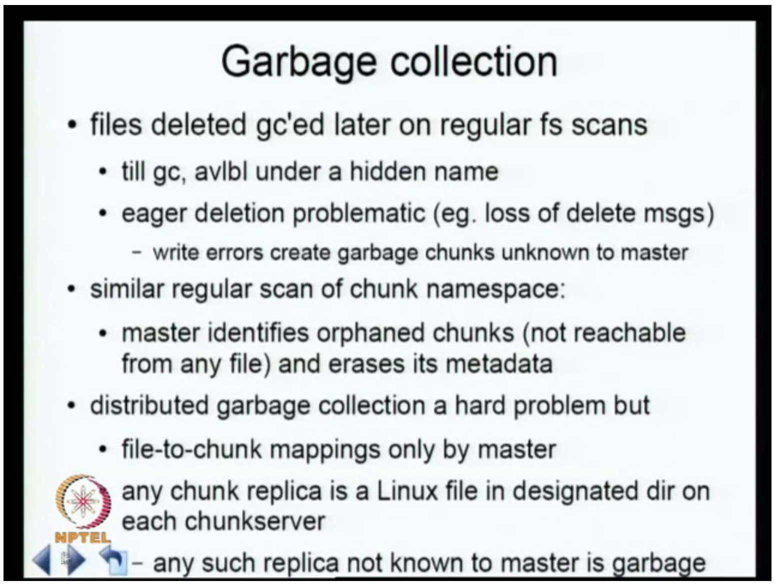
So, basically the thing is once you the reason why you are creating a chunk is because somewhere wants to write. Once you are finished writing it, it now starts becoming read only heavy traffic can start after that. So, that means, that it becomes a likely that if you create too many of them on the same chunk server. Then there will be 2 much write traffic going to that particular chunk server and that particular thing might be overload overloaded.

So, and you also just like before we have mentioned, you spread replicas of a chunk across racks also. Now this one case master creates a chunk. Then there are similar issues like master replicate re replicates a chunk for example, it turns out that one chunks of a dice. So, you wanted 3 copies now there only 2 copies. So, you want to re replicate. So, that number of available replicas if back to normal.

Now, again there are lot of interesting possibilities. This particular system has certain policies for doing this. For example, it turns out that there are some chunks which are blocking client progress. Because they are taking too long they have overloaded. If you notice this kind of situations, you boost the priority of that chunk. So, that you can have a new replica for that. That is if you are also lost too many replicas for example, you wanted have 4 copies, now you may have lost already 2. Then that means, you are lost more than you expected typical at the most you want to be at the most one away. For example, 3 or something like that you already lost 2. Then that is becomes very high priority other thing that want to do is to make sure of the bandwidth is handled properly, a limit number of active replications in cluster. And in each chunk server you have too many active replications then that also will have a interfere with actual work that is being done. Because these things are all basically management operations. You are just so that your comfort level is good, that I wanted 3 copies and 3 copies. There actually not doing anything with respect to the actual client request.

So, these management operations can essentially suck away all the bandwidth. So, idea is to also try to see if we can limit the number of active replications other thing the chunk server does is it limits replication bandwidth by throttling read request to source chunk server. So, you want a chunk server also can essentially play a role if it can throttling read request in case basically the number of replications require to high thus throttling concept. There is also the system also the master also rebalances replicas again from with respect to load balancing. Again, here most of ideas are similar to what we already discussed. Basically, a you want to ensure that there is not too much bandwidth taken up while you are it balancing etcetera those kind of thinks.

(Refer Slide Time: 46:22)



Garbage collection

- files deleted gc'ed later on regular fs scans
  - till gc, avlbl under a hidden name
  - eager deletion problematic (eg. loss of delete msgs)
    - write errors create garbage chunks unknown to master
- similar regular scan of chunk namespace:
  - master identifies orphaned chunks (not reachable from any file) and erases its metadata
- distributed garbage collection a hard problem but
  - file-to-chunk mappings only by master
  - any chunk replica is a Linux file in designated dir on each chunkserver
    - any such replica not known to master is garbage

I already mentioned this a bit where basically this particular file system is slightly unusual. Because files are deleted lazily and garbage collected on regular file system scans.

So, all you do is you write down the delete time, and then leave it as leave it, there you do not delete the file. And interestingly also in the system, they have even if something is deleted till actual garbage collection. The file that is deleted is available under hidden name. Again, these have a similar to even nfs for example, when we have the open, but delete situation, right. The server will actually rename that file with a in accessible name, right. Which the clients on that particular client which deleted it can actually access.

So, the similar is going on here also. And I am told that you can even rename this hidden name to the what you need in case you find that. There is a reason why you want to get back the those kind of user possible one thing they are an interesting thing about this particular design is that they found eager deletion to be problematic. Basically, because when you want to delete something, the master has to send delete messages and they may be lost. And the it may be that the machines also die; that means, that if you think that you are deleted somewhere strictly pluck off I have told them to do it, but they might not have done it, all right the master has to keep on thinking about all the type if you do eager deletion which is a bit of a big overhead.

So, the idea is that why going to that kind situation. You just do it lazily and do it when you have to anyway scan the file system for the reasons. It could be for many other many for example, it could be that you are scanning it, because you want to see if any disk block has gone bad. Normally what happens is that disk suffer from something called late intervals. So, to catch those things you scan it disk every so often. So, the various scans a that kind in the system.

So, why not piggyback on those kind of scans, and detect those things that have to be deleted, because you have marked it as deleted at that time when it is deleted you just a at the time when a do this can you release it. Of course, this is a problem because this can be a problem, because if you are very close to the end of your storage availability space availability. This kind of systems also will create some problem. Because that at the you have to actually be not lazy about it you have to be active eagerly have on certain things at that time.

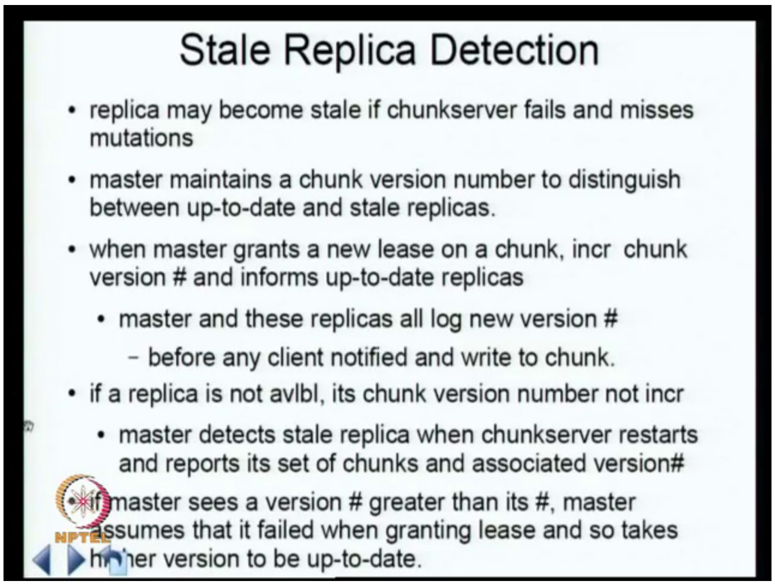
But the reason why eager deletion is problematic is because write errors create garbage chunks unknown to master. Basically, the chunks are kept by the chunk servers and certain parts are the disk can suddenly go back. Again, I have mentioned latent secondary also. And these are not known to the master. So, you need to have some if really you want to do a eager deletion proper one, right. Then what will happen is that do it then there will be some chunks which are not known to the master as having deleted

So, because of all these things they have decided that it is best to delete files lazily. Now they have a similar regular scan of chunk namespace also. The chunk namespace for example, is kept in the as because each chunk is a Linux file, and you keep various mappings between chunk identifiers and the chunk file names of the Linux files file names.

So, basically master identifies orphaned chunks not reachable from any file and erases it is metadata. So, in their design it turns out a very hard problem called distributed garbage collection, a it is sorts like the simpler in a simple manner. Basically, because the file to chunk mappings are kept only by the master. It is an memory actual as I mentioned earlier as I say discussed earlier, the chunk master is responsible for all the chunks that is there. And the chunk to file mappings are kept in memory by the master, because once the chunk server comes on line it sends it all the mappings and then it keeps a consolidated list of all the file to chunk mappings. And any chunk replica is a Linux file in designated directory on each chunk server any replica; that is, not known to the master is garbage. So, that can be deleted.

So, this way they actually solve a slightly tricky problem in a slightly simple manner.

(Refer Slide Time: 51:43)



Stale Replica Detection

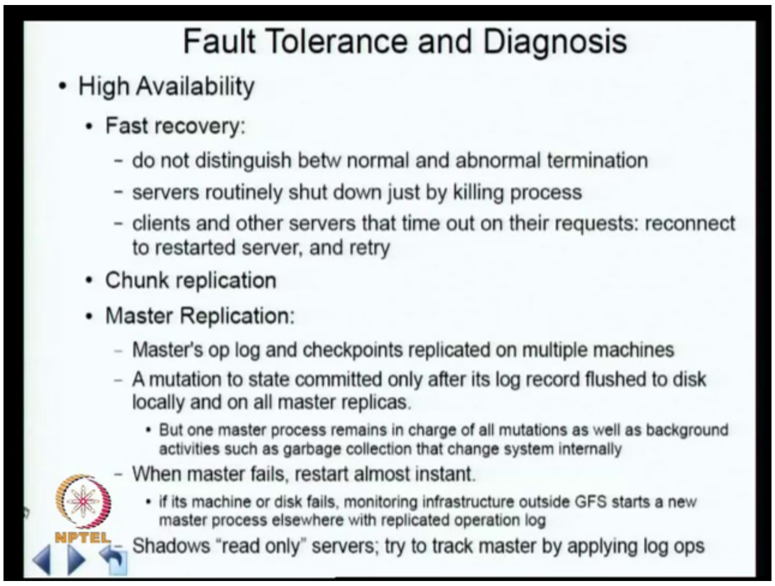
- replica may become stale if chunkserver fails and misses mutations
- master maintains a chunk version number to distinguish between up-to-date and stale replicas.
- when master grants a new lease on a chunk, incr chunk version # and informs up-to-date replicas
  - master and these replicas all log new version #
    - before any client notified and write to chunk.
- if a replica is not avlbl, its chunk version number not incr
  - master detects stale replica when chunkserver restarts and reports its set of chunks and associated version#
  - if master sees a version # greater than its #, master assumes that it failed when granting lease and so takes higher version to be up-to-date.

They also have the issue of a stale replica detection. Replica may becomes stale if chunk server fails and misses mutations. Therefore, the is something is something called a chunk version number to distinguish between up to date and stale replicas. Again, this version number is similar idea what you lot of file systems do if you reuse a inode number you have a inode generation number, and that is used to figure out if you have you are reusing a particular inode block.

So, that you do not get into the stale situations. When master grants a new lease on a chunk, you increment chunk version number informs up to date replicas. Basically, at the time at chunk is being used for write for example, right. You increment chunk version number and tell all the up to date replicas, that there is the new version number. Master and these replicas all log new version number before any client notified and write to chunk. If a replica is not available it is chunk number will not increase, questionable will not increase.

So, when this is not available because the chunk server is done. So, when the chunk server again restarts, and reports it is set of chunks numbers associated numbers it can figure out stale replica. It is possible that chunk server is done or the master itself can be done. If the master sees a version number greater than it is own number for the corresponding chunk. The master assumes that it failed when granting lease. And so, takes higher version to be up to date. Because it keeps all the information in memory. So, it has lost it and now it can rebuild it from the chunk servers.

(Refer Slide Time: 53:41)



Fault Tolerance and Diagnosis

- High Availability
  - Fast recovery:
    - do not distinguish betw normal and abnormal termination
    - servers routinely shut down just by killing process
    - clients and other servers that time out on their requests: reconnect to restarted server, and retry
  - Chunk replication
  - Master Replication:
    - Master's op log and checkpoints replicated on multiple machines
    - A mutation to state committed only after its log record flushed to disk locally and on all master replicas.
      - But one master process remains in charge of all mutations as well as background activities such as garbage collection that change system internally
    - When master fails, restart almost instant.
      - if its machine or disk fails, monitoring infrastructure outside GFS starts a new master process elsewhere with replicated operation log
    - Shadows "read only" servers; try to track master by applying log ops

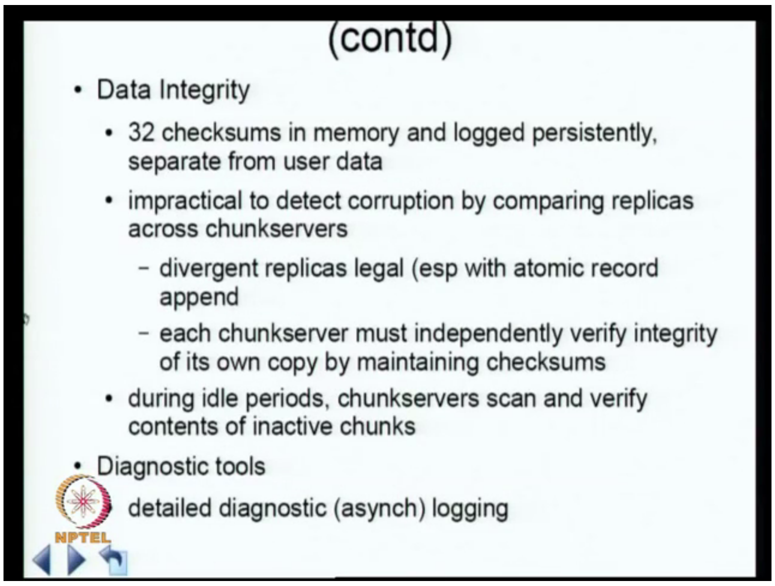
Now, this particular system has as I discussed earlier has designed in fault tolerance and diagnosis systematically. But high availability is one aspect data integrates another aspect and it also has certainly supports lot of diagnostic tools. For high availability what is important is fast recovery. So, basically clients and servers they make requests the time out they again reconnect to restarted server and retry that is a normal operation is there write the way we use a web. We try something that does not work do you retry the think, right. Because the if the state is minimum then this is very reasonable thing to do and the servers routinely shut down and just by killing the process.

So, first recovery is one chunk replication chunk replication is another we already discuss this part. We not discuss this part the master application. Basically, what we have here is the master keeps also some shadows. These are not mirrors. Mirrors are those in which exact copies there. These are shadows. What is the shadow? It is got most of the stuff the master has got, but it is slightly delayed it might be behind by few operations.

So, masters operation log and checkpoints replicated on multiple machines, the shadow machines. The mutation to state committed only after it is log record flushed to disk locally and on all master replicas. All the shadows should also get the log, but the it is only in the log it is not the shadow things do not have the actual only the log is there, but not the actual data. So, the important thing as we discussed earlier is that one master process remains in charge of all mutations, as well as background activities such as garbage collection and that change system internally. Only one master still there. When master fails restart is almost instant, because you detect the failure and you restart a new process new master process somewhere else using the replicated operation log.

Essentially the shadows can also be used as a read only is servers. If for bandwidth purposes this shadows can also be used for getting certain operations done, but they will be slightly stale. So, the application has to be careful about when it can use the shadows for because the metadata can be slightly stale. Other thing that this particular design does is to avoid this stale less problem with metadata in the shadow servers, it tries to track master by applying log operations as things are coming in it tries to do it also.

(Refer Slide Time: 56:50)



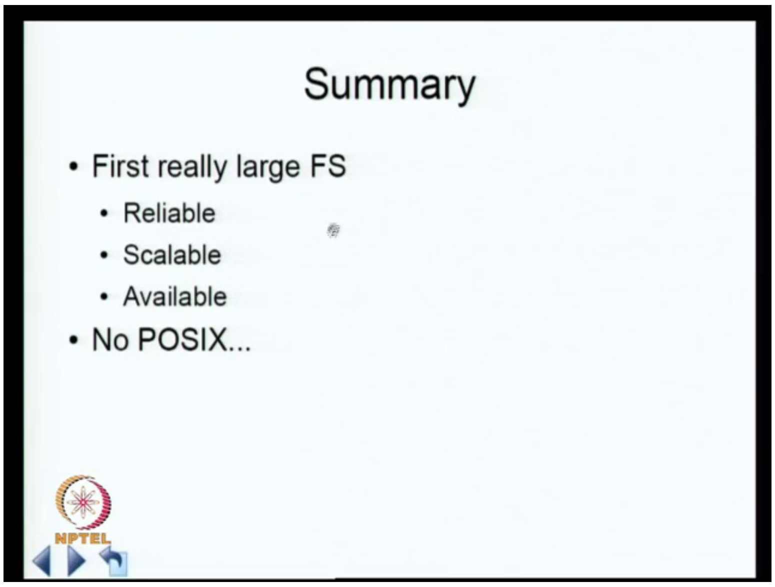
(contd)

- Data Integrity
  - 32 checksums in memory and logged persistently, separate from user data
  - impractical to detect corruption by comparing replicas across chunkservers
    - divergent replicas legal (esp with atomic record append
    - each chunkserver must independently verify integrity of its own copy by maintaining checksums
  - during idle periods, chunkservers scan and verify contents of inactive chunks
- Diagnostic tools
  - detailed diagnostic (asynch) logging

So, there is also data integrity there is taken care of seriously in the system. It has got 32-bit checksums. If I am sorry 32 bit checksums in memory. And there logged persistently separate from user data. Now it is impractical to detect corruption by comparing replicas across chunk servers. An actually chunk server must independently verify integrity of it is own copy by maintaining checksums. So, each chunks do it. Reason being the replicas can definitely go become divergent, because it is part of the semantics. Because we notice that with atomic record appends things can go divergent therefore, each chunks over should maintains (Refer Time: 57:37) for it is own checksums and checking it. Again, during idle periods chunk servers scan and verify contents of inactive chunks. We do not have too much time, but basically, they also have fairly substantial diagnostic tools, with detailed diagnostic logging. Mostly done asynchronously to avoid performance what are loss.

So, with this I want to conclude this today's discussion about GFS.

(Refer Slide Time: 58:03)



Summary

- First really large FS
  - Reliable
  - Scalable
  - Available
- No POSIX...

I think basically it is GFS when it first came out that probably was the first really large file system, which had reliability, scalability and availability as core parts of it is design. And because it was designed with application mind, it avoided the Posix issues. And therefore, it could scale with the way it had to be scaled. So, I think I will conclude here with respect to GFS here. In the next class I will briefly talk about what is write some of the issues of GFS currently as perceived by the designers. And then from there we proceed to other types of large scale storage designs.

Thank you.