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

Highly scalable Distributed Filesystems
Lecture – 40
Highly scalable Distributed Filesystems _Part 2: The GFS Model

Welcome again to the NPTEL course on storage systems. In previous class, we started looking at in the google file system. It is as we discussed before, it is a highly scalable system, and basically it is geared for performance, scalability, reliability and availability.

(Refer Slide Time: 00:50)

Google File System

- Non-Posix scalable distr file system for large distr data-intensive applications
  - performance, scalability, reliability and availability
  - high aggregate perf to a large number of clients
    - sustained bandwidth more important than low latency
- High fault tolerance to allow inexpensive commodity HW
  - component failures norm rather than exception
  - appl/OS bugs, human errors + failures of disks, memory, connectors, networking, and power supplies.
  - constant monitoring, error detection, fault tolerance, and automatic recovery integral in the design

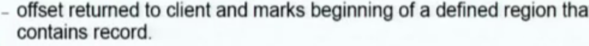
NPTEL

So, is so, we looked at some aspects of the google file system, and we notice that google file system has a specific idea about how to do things. One of them is that it takes care of faults in the system quite well, and allows in that it takes care of very well is. It has a somewhat relaxed consistency model. And the idea are the here also is that you want to design the applications on (Refer Time: 01:28) file system. So, we are not assuming Posix file system, we are assuming somewhat different, and for that reason this is the it has got a relaxed consistency model. So, we will try to understand what this is today.

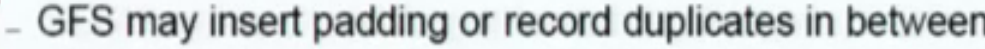
(Refer Slide Time: 01:41)

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.

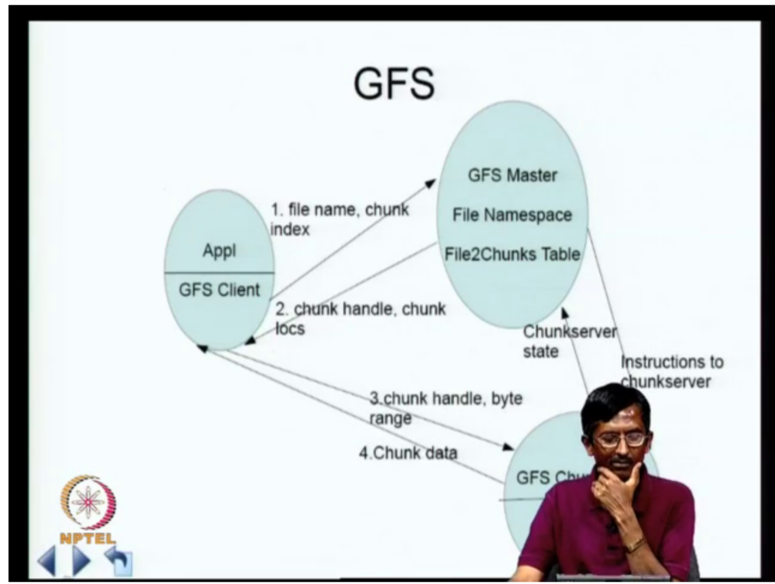


    - GFS may insert padding or record duplicates in between



So, you notice that somethings have to be atomic for example, if you are doing file created etcetera, that is you create a file and also it has to be certained with the directory atomic model. If you do not do this, then it can create lots of complexities. So now, the file name operations are namespace operations, or done exclusively by the master it is not done by the clients sorry, by the yeah by the clients. So, you need to guarantee atomicity and correctness of these operations and this going to be locking. And this particular design as you mentioned early is a masters there masters and there are clients. So, assume is about client now basically there is a master, and you have what you call this (Refer Time: 03:11) this it is a yeah various GFS clients. This (Refer Time: 03:24) that is about sorry, not well remove the exact terminology.

(Refer Slide Time: 03:22)



So, we will calling a GFS master and GFS client. And basically, the masters operation log defines a global total order of these operations.

Notice that ordering is an important issue because we discuss ordering sometime in the past. And what this particular system does is, it defines a global total order and then it is expected that all the GFS clients also notice this same. So, first well let us define what a mutations. It is an operation that changes contents or metadata of a chunk. And the types of mutations that we talked typically talk about a writes or record appends. So, again in regular file systems, you we were just call everything a write even a appends is called a write, but in this particular system they have given 2 names, one is a write and a record append. Write is what?

Write the datas written at in application specified file offset. Whereas, a record append it is appended automatically at least once even in spite of concurrent mutations, but at certain offset of the file systems chosen. Again, we discussed last time that we have the similar to touch and nut situation. Remote that in touch and nut instruction many concurrent people are trying to improvement something, and each party gets their own motion of incremented value.

And it is there is no 2 log 2 people will get the same incremented value. Some is similar is what is going on here. So, there can be multiple appends, each append happens at a particular offset, and then this also is chosen by GFS. And order is not up to the client. So, the also returned to the client and marks beginning of the defined region that contains a record. Now what (Refer Time: 05:52) also is the GFS may insert padding or record duplicates in between. What is it insert a padding? Because it may turn out the file that is been; what is being written is going to span chunk boundaries.

When you span chunk boundaries, then it create some complications if you are to proceed buts you have to do work with proof chunks now. So, that case complications. So, the idea here is that if it is happens at the right is going to straddle 2 chunks, you do not try to do it you think that chunk we basically pad it, saying that some based just empty space I move to a new chunk. That way the logic come to that simpler. The idea is not to work with complicated things based if there is anything which seems complicated just differ the problem make it simple and work with it. So, you insert padding in case there is a right that straddles 2 chunks and work with only a chunk deposit.

Also, there could be some duplicates because of if some error takes place, then you can detract operation and sometimes there can be duplicates, because of earlier attempts to write. And this have to be or detected. So, basically the idea here is even if there is something which is a duplicate, you need to recognize the applications of the application logic through a library code there figure out these are duplicate. So, again this makes it first like a simpler design. If you look at a Posix kind of systems you do not have this kind of options. Posix cannot already this kind of things. Because the application is not is not been told that this kind of things can happen. Where as in GFS there is the model and the applications have to figure it out it out. And so, that is why they will actually check for certain consistency before they proceed.

(Refer Slide Time: 07:57)

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
- concurrent successful mutations leave region undefined but consistent
  - all clients see the same data, but it may not reflect what any one mutation has written.
  - typically, fragments from multiple mutations.
- a failed mutation makes region inconsistent/ undefined
  - different clients may see different data at different times
- applications can distinguish defined regions from undefined

So, it is responsible that application. So, if you do this mutations, then the state of the file after this mutation depends on the kind of thing that happen was it success or failure was it concurrent. So, GFS they define some terminology, something called consistent and something called defined. A file region is consistent, if all clients see the same data, regardless of which replicas they read from; that means, there are always in the same data. So, inconsistent means they see different data. Usually you have failures that can happen.

So, consistent means all the clients will always see the same data, but this may not be what really is what you want also what you want is something called defined region. It means it is consistent and clients see what when attempted to write may it is entirety. So, in the sense what is happening is that, there will be multiple writers and you are getting inters persed write from various writes, that will be called an example of a if all of those things succeed, it will be defined, but it will not be it will be consistent, but not defined.

So, for example, when a mutation succeeds, without interference from concurrent writers, the region is defined and consistent. So, basically client see a mutation writes in entirety in a without any break etcetera. Whereas, a concurrent successful limitations leave region undefined, but consistent. There is all the clients is the same stuff, but they do not see the write in it is complete entirety that will be interspersed. So, all clients see the same data, but it may not reflect what anyone mutation has written. Typically fragment some multiple

mutations. Multiple writes are going on concurrent writes and then they can each everybody see the same data every client, but what is there on the file is written by different writes in some order, but everybody see the same data. It is not one write incompletion in completing in one entirety follow over some other write may be (Refer Time: 10:56).

So, there is a failed mutation, it makes a region inconsistent or undefined. And different clients may see different data different times. For example, in one let us say that there is some write go multiple writes are going on. And some write was done, but one replica could not update it is result is, because it failed at that time. Because it is failed it has the information of the previous content whatever it was. So, I write has gone to the replicas properly, but third replica before it could be done might be part of it has been done it died. And then right now it is possible that there is the proper write in 2 replicas, but partly only in other replica is that way that (Refer Time: 11:57) we have seen different clients may see different data at different times.

The one thing is again is in a with this example for cross layer design, and here is that applications should take control about the situations the semantics whatever the consistency guarantee where look at the thing and say again this distinguishes from Posix. But this is manageable because it turns out that mostly in this particular file system appends are taking place large files are being written there is also ca there is also check pointing going on.

So, the checkpoint happens; that means, that you are on a very well-defined state. So, readers can always look at checkpoints and then proceed with that, they do not have to look at the current updates are going on. So, again this script I think we go through a bit more carefully. So, the idea basically is that you can have it something consistence means everybody see the same data, but it is not it may not be defined, it is defined in case a particular write made it in it is entirety and it is you can see very clearly that this write has without being interspersed other writes where is if there is a failed mutation. What is happening is that some clients might see some information at updated because the (Refer Time: 13:25) chunks are was down and it was now updated when it came up no other clients can actually in principle you see it (Refer Time: 13:38).

(Refer Slide Time: 13:38)

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

NPTEL

So, this is a model they I have given in the paper. So, you can have 2 different types of mutations wtites and record appends. Now if it is a serial either it is a serial write or a serial record append; that means, one after (Refer Time: 14:01) solve completely serialize in way, then if it is a success then it is going to be traditional is the normal the defined part of it. That is most consistent and it is also depend.

Now in the case of record append, the reason why that is there because a required appends are used as a way to attach something to a file without going through locking. One of the interesting thing about this design is that that try to avoid things like distributed lock managers (Refer Time: 14:32) as you notice (Refer Time: 14:35) are somewhat complex, and they slow down the system called a bit. So, the idea here is just like in the case of (Refer Time: 14:42) you basically want to allow multiple updates to happen in parallel. And the way they do it is the without a DLN, you are going to have concurrent appends going on. And this concurrent appends also can result in a situation where it is defined, but certain portions of the system of the file contains could be inconsistent again because of failures.

So, basically what happens is that you can have a record append going on, and some of the replicas they do not get updated with the append because again the failure kind of situations. So, they might have different piece of information, because there is

concurrence here see, there is no concurrency here therefore, there is no issue of it being any other than what has to be (Refer Time: 15:45) case. So, all these things are concurrent situations. Now this is the serial lock. This one says a serial success, but actually this required append, because it is essentially trying to give you a concurrent way of a appending things. That is why these 2 could have the same problems like what concurrent writes also do.

Now let us look at current what is concurrent you are writing multiple writes. And here you can have everybody sees the same thing consistent, but it may not be defined in the sense that what you see on the file is not the write of a single write in it is in it is entirety it could be partial here and there. So, if this is a failure you can have inconsistency, because some guy has not some particular replica has not updated it is part of it is got old information where the other guys have updated it. So, there is that inconsistency is there.

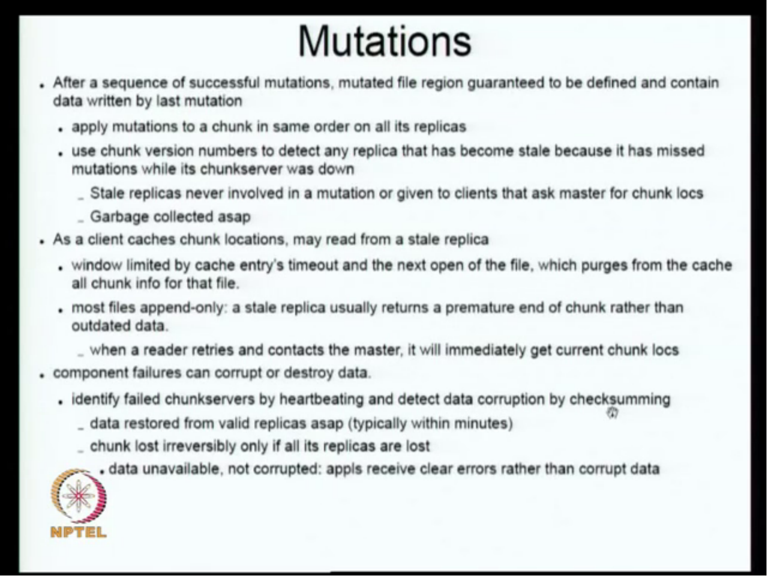
So, again we look a bit more into this exactly how this happens and what are the ways to resolve the issues. The most important thing to remember is that failure is very clear some of not updated it there is some old and new mixed up old information and new mixed up.

In the case of serial, it is very clear it is basically only one part is updating it and will see how we can ensure that every if there are no failures everything happens in the right way. This is concurrent things then again this can happen because of record appends or it can happen because of writes unit concurrency. So, what happens is that you retry the operation, when you retry the operations you might have what happens is that you have not given the offset the client is not told if it fails right it was told that it is, something has failed. We do not give them the offset. So, there time to retry them. So, whatever has been written is actually chunk. So, will have interest because chunk here and there, but finally, when it succeeds you get the offset, when I get the offset you know where exactly is a complete write.

So, that is why it is defined here interspersed with some inconsistent information. So, in the case of this write, you may find that it is consistent, because all of the chunks are basically written on the same order. Basically, that master actually has a order in which it is written and all of them actually write in the same order. And that is why it is consistent. It can be undefined, because the mutation is not written completely. Because if there is

interference, then what will happen is that different writers will write different portions the file. And therefore, the region may not be completely the what is say the applications can figure what is going on, but it is not exactly the mutation write in it is completely entirely.

(Refer Slide Time: 19:43)



So, see if you are having many mutations, the mutated file region is guaranteed to be defined and the contented written by the last mutation.

So, basically what is the way this is guaranteed is by making sure that all the changes to a chunk are done in the same order on all it is replicas. And that means, that there is some kind of a primary and the primary decides the order and everybody else follow the same order. So, it also uses something called chunk version numbers to detect any replica that has become stale, because it has missed mutations while it is chunk server was down, because we because there are possibilities of failures. So, replica may become stale because it has now missed some changes. And to detect these things you have version numbers, the version numbers will be able to you can be used to figure out this replica stale.

Again, this similar to again if you look at NFS, you have some notion of what is called stale file handles there (Refer Time: 21:01) you called new generation numbers, and from

there you can figure out something is stale. Straight different model here, but it is something similar.

So, one some stale replica is there it is never again involved in a mutation are given to clients that ask master for chunk locations; that means, once some replica becomes stale the master will essentially take it off. As something to be given to (Refer Time: 21:32) stale as to a client. And the idea also is that these stale replicas are a problem and they will be garbage collector, depending once we will discuss this in next class possible, I am going through it in detail because this particular design has got lots of interesting let us from various you know all let us say well known ideas, but they integrated very well. So, this some garbage collection also going on for those stale replicas that have to be removed.

So, there is a small problem also just like NFS as client caches chunk locations may reaf from a stale replica is possible. So, the windows limited again just like in NFS where windows limited by cache, entries, time out and next open a file. When you open a file (Refer Time: 22:25) function purges a cache from the cache all the chunk into for that file. So, if you really want to be (Refer Time: 22:34) about something you just open the file and make sure everything is cleaned up. So, most files are appendable, I think this again we mentioned is important part of the design. A stale replica usually returns a premature end of chunk rather than outdated data. So, basically what is happening is that if you have a stale replica you are not updated somethings, you have not taken care of certain updates, because you because the chunks are washed out.

So, usually what happens is that you think that the file is this big actually turns out to be slightly smaller. So, if this kind of error takes place premature end of chunk. It transfers some size and it has been not there you get end of end of whatever you can retry the reader can retry and get the results from get the information on the master about exactly the character issues. So, the client may actually is a stale replica, but if it so happens that it tries to do something with it gets another.

And therefore, it can get a current situation correct situation there is also other possibility that component failures can corrupt or destroy data. So, for example, some chunks others might have failed, and these chunk servers feeling or detected by heart beating. What is

heart? What the heartbeats? Basically, these are messages sent from a master to other nodes to figure out if there alive or not. And that is so, identifying failed server is done through heart beating, but if you did data corruption is handled by checksumming.

So, every 64 another by it is has got a checksum, and it has to match, when you read it has to match otherwise you know that something is wrong. So, again if there is any invalid replicas, you need to restore them and they typically I am told happens with 10 minutes. To only why get loose something is if all the replicas are lost. Now important idea here is that you can determine the unavailable, but you not given corrupt data. Under all circumstances they try to guarantee that it is possible that you may not have access to (Refer Time: 25:25) data.

So, the application knows that something it has not worked out, but it can cannot work with wrong data or corrupted data, that is one thing which they try to guarantee. So, ris application receive clear errors rather than corrupt data. And it is quite important because and the scale (Refer Time: 25:40) it is done. And if I write (Refer Time: 25:44) level it typically finds that there are about, at least about 10 to 100 errors per day. These are what are called undetected errors if you do not do checksumming and things.

So, if this kind of errors happen in critical part say system then you might get a file system might get completely inconsistent. So, it is a very serious issue; that is, why at this level you need to ensure that there is already that disc is already in lots of reed solomon coding all those things. In addition, you doing checksumming so that these kind of issues are taking care. And again, one other thing is that the GFS is at happening at the application level all, right. This guys closer to an application level system. So, the checksum is happening at close to application; that means, that the checksum is happening with respect to both discs as well as network. And all that transfer that is happening.

So, this is closer to a n to n design of course, is only at the application level file system level not at the current level, but anything upto till application wherever it is come from days network HDS storage and network networking hopes whatever it is right they all get all the data will be let us say at the end only you check the checksum. So, any error somewhere in between could be caught. And since there is a application and the file system working together, the checksumming aspect also in a sense almost at the

application level.

(Refer Slide Time: 27:23)

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 & record frags using checksums.
      - occasional duplicates and non-idempotent ops: readers ignore using unique ids in records, needed to name corresp appl entities such as web docs
  - 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) in library code linked to appls
- simple approach: avoids consistency and concurrency issues
- no DLM needed with record append

So, there is some kind of end to end guarantee. So, let just to see how this what is the kind of applications that you have to design with GFS. First you have to depend on appends, rather than overwrites. You may depend on check pointing and you have to depend on writing self-validating self-identifying records. Because things can go bad, when you write you should write down in such a way that you read the data, and then figure out what can what has to be done again this happening at the application level it is not happening yet much lower levels.

So, let us see what are the kinds of issues that appears. What is the typical use of a write, a writer generates a file from beginning to end atomic at renames a file to permanent name after writing all data. And why is this talked about basically because the GFS itself is running on a Linux file system. And so, it might be writing parts of a big file and to various small files. So, that recovery is easier to something breaks a (Refer Time: 28:44) recover from right. That is why finally, after you probably might want to take the chunks and make it to one single whole let us by that atomically rename is sometime may be (Refer Time: 28:58) after writing all the data.

(Refer Slide Time: 39:03)

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 control flow to improve perf by scheduling expensive data flow based on network topology regardless of which chunkserver is primary
- primary picks a serial order for all mutations to chunk
- all replicas follow this order when applying mutations
  - error: write may have succeeded at primary and some subset of secondary replicas
  - modified region left in an inconsistent state
  - client code handles such errors by retrying failed mutation: try resending data
    - no success: retry write from beginning
- file region may end up containing frags from different clients, but replicas identical
  - file region consistent but undefined state
- global mutation order defined by
  - lease grant order chosen by master
  - (within a lease) serial numbers assigned by primary

And one thing about reason why appends are used extensively is because, appends more efficient and resilient to application failures than random writes. Why is it more efficient? For the same reason fetch ended is efficient. So, without fetch ended what happens, every time you have to some of serialize every person and everybody has to go through locking right one by one. When fetch ended what happens is that lots of people are concurrently updating it and everybody gets updated version one round trip. Something here also.

That is why it is more efficient. So, that is not a severe lock manager and it is the file system takes care of responsibility of taking all the updates and giving it some order. So, the applications are not aware that they are doing any locking that is very strong point that is why it is more efficient. And is resilient to application failures than random writes. So, basically random writes, it turns out have to do read modify write (Refer Time: 30:06) if you are doing anything in between, you have to do lot of read modify write.

With appends; the issue is slightly simpler, because there is no the read modified circles are not really necessary. So, we typically uses one writer generating a file from beginning to end, and append it. Or there could be many writers concurrently append to a file for merged results or as a producer consumer queue. Now we will go and read your exactly how the append happens, but the important thing is the recall append guarantees a following, which says at least append at once it is a preserves now basically each writers

are pretty preserved.

Again, this will look at it pretty similar to in the case of NFS, you have this issue about what is called at least in a at least once right. Again, that goes back to RPC if you see this RPC calls, right. In the RPC basically, a semantics either you can say at most once or at least at least, once at most once means either it is 0 or one time typically in RPC people try to give you at least one semantics. And NFS is based on at least one semantics; that means, that you try to do and NFS operation, it is at least done once at least done once. So, is a problem with I think all of us we have this discussed before with NFS you do at least once then for non-idempotent operations you get it probably right. So, special logging has to be done to catch those non-idempotent operations. Here we also have append at least once, similar to what he saw in RPC's at least one semantics you have up append at least one semantics.

So, in the case of NFS, we handle the problem using using logging. Here it is done by looking at if you see that there is something called self-validating self-identifying records right. So, basically you write some sequence numbers. And that sequence number tells you whether this is an extra copy setting of the thing that has been written. So, the application actually has to understand this things. It actually knows that this kind of things can happen and has to prove actively handle it; that means, if it leads a chunks it can look at the some metadata, and nodes know that this part of the file is this part this region is not this region is undefined. And (Refer Time: 33:03) discard it.

And then there is because at a self-validating self-identifying records, it knows where the offsets are. So, again a chunk in there will be some information somewhere by which it can figure out that actual offset is here even though physical aspects something else. So, that some kind of a this kind of notions are all there. Again, the application has to keep resonance this that something readers identify or discard extra padding and record fragments, sorry. Readers identify and discard extra padding, and record fragments using checksums. So, they have to do this systematically.

So, occasional duplicates and non-idempotent operations readers ignore using unique ids in records needed to name corresponding application untitled such as web documents. So, again this an example of an application driven file system, if you have you want to detect

as I mentioned duplicates etcetera, in there are some unique ids and records and these record these things are basically connected with the application level entities like web documents. So, basically it will have the information saying there is a web document x part number 4 etcetera. So, if you see 2 chunks or participate chunk with same part number 4 they know it is the duplicate. So, just like we number pages if I have duplicate pages you can figure out that is duplicate we have to do the same thing here also. So, they so, in a sense you do not do only by offset. You go by the offset and also look at some metadata and you define that something is a duplicate.

So, the typical use is concurrent writes or single writer, and then there are duplicate. So, that things you have to handle. Or the other thing we can do is by check pointing. Periodically checkpoints how much has been successfully written basically when you doing when lot of concurrent writers are going on appends are going on, because of possibility that this the thing can be undefined state, right. You want to use only checkpoint information. The checkpoint essentially is a consistent and defined part of the file. So, basically if I have a check points it keeps readers from processing successfully written file data that is still incomplete from application point of view. So, even though it has successfully written it is inconsistent. Sorry I made a mistake. It is undefined it is consistent, but undefined.

And readers can be can avoid this kind of situations, they can essentially look at only things which are defined not necessarily consistent. Of course, other areas of course, the standard thing advantage of check pointing is that keep something as where you can restart from where you know last checkpoint it that is basically you can restart incrementally if I want restart from. This consume very long write very large write, right. Then it something has broken you go from the check point (Refer Time: 36:17) and these are logics importance, because it is a very large-scale file system, and you are talking about multiple gigabytes or terabytes at one go.

So, there is no way to provide atomicity. There is no way to provide atomicity, because you do not have terabyte, memories there is no way to do it. So, by definition any large file of that kind we have to break it up into pieces. And write one at a time the minute you say break it up into pieces then atomicity or the whole thing is not possible. We cannot guarantee it at all it is not possible. So, you have to handle it. So, application whatever

has checkpoints can include application level checksums.

Readers verify and process only file region up to last checkpoint which is known to be in the defined state. So, defined means again consistent as well as the way a complete track has been written. So, all this semantics and how to handle all these things is in the library code linked to applications. So, I am not clear, but they say that in the paper that except duplicate removal everything is handled in the library code some of this is not handled I am not clear. So, this particular design is simple, because it avoids concurrency issues because the; appends basically take care of without locking it does it.

So, it avoids some other concurrent issues. Because the file system guarantees atomicity of appends and the consistency is avoided. Because you have a relaxed model you do not have this stringent const requirements that process will have. So, you basically such that some other complicated things are there, but the application become slightly more complex, but the idea is that the library code handles all of it. So, the file application itself may not have to struggle that much. And according to this paper which again, I am not very clear that duplicate removal this mat handled by this library code. So, that is the only thing that the application handle, but that one can be simple, because as I mentioned there are unique ids in records.

So, as long as this unique id is or something anyway you need to name application entities. And therefore, if there is a duplicate these unique ids can be used to figure out that this a duplicate. So, again as I mentioned there is no distributed lock manager required for this got it. For especially means you have atomic appends. And let us look at how this because you have multiple replicas, we have to figure out how to apply those changes to all the pla replicas there has to be some order, how is this decided each mutation is per performed at all junk replicas, right. Replica by definition should have the same information like any other replica (Refer Time: 39:26) therefore, any operation that you do should have at all synchronous. If it is a very large write or the write that straddles chunks, right. That is there is a write which begins in some one chunk and ends, in some other chunk that is of the boundary of 64 megabyte. To simplify the system, you break up the client code. Sorry, it is broken up by the client code, sorry.

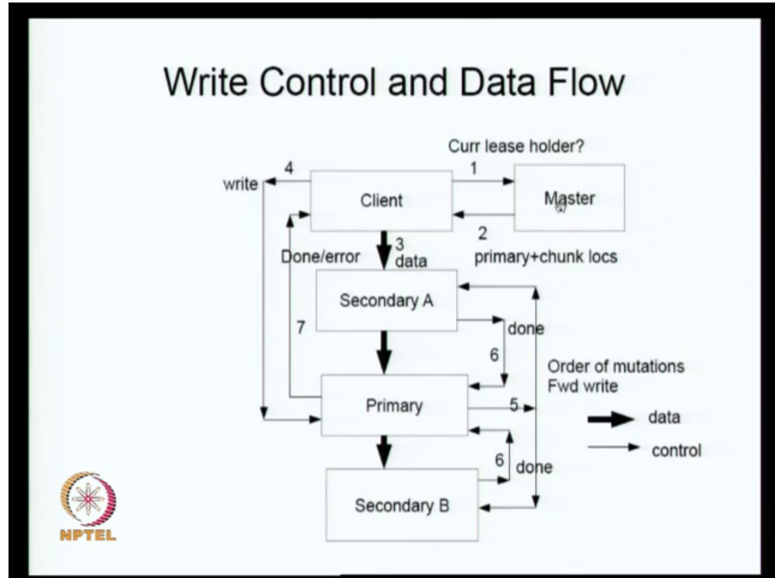
The client can breaks up that write so that each write happens to be on the same chunk.

So, one way to do it is by padding. So, basically what it do is padding. So, that the chunk becomes full. And then you can start with the new chunk. Or of course, you can break up the write itself. For example, if it is a if padding is not is not efficient, it is not a good thing to do then you can break up the write itself. So, it uses leases to maintain a consistent mutation order across replicas. Basic idea here is you have a primary replica, and the primary decides how the updates are done in which order. So, and basic idea is that if in case of primary fails, then lease will get broken. Because what is a lease it is for the master gauge one of the primary chunk servers, right. A certain pre-language it can do the primary for all the replicas.

And if it does not respond back, or read basically try to get the lease again. It can give the lease somebody else, or it possible that a primary it has failed. So, because you have the lease model, even if the primary fails, then it would not reconnect with the master to remove the release. Therefore, the master can actually give it have it a somebody else. You do there are it is a different, I think important thing is that it is not acid it is a kind lock it is take a lock. Then you have to do something you have to have to figure out how to ensure that the lock is broken which is a bit of problem. Whereas, the lease it is only (Refer Time: 42:00) of time. If it is not renewed it goes away automatically, and the master is absolutely that clean with the clear idea it can give it to someone else.

So, master grants a chunk lease to one replica, primary and chunk locations basically what it does is it grants a chunk lease to one primary, and it also gives you where the chunk locations are there and these are cached by the client. So, basically what happens is when you write when you write is requested, we have to find out who was the lease, and where are chunk locations. And this information is cached by the client. And the client pushes data to all replicas, and this data is cached. And then all the clients I acked in it, let us look at a diagram here.

(Refer Slide Time: 42:50)



So, this is a client here just trying to do write. And so, first if it is writing it has to find out who is the current lease holder if there is one.

So, if there is a current lease holder, then that information is given by that is a primary or if this none it will take one more the replicas, the chunk server corresponding to that replica and make it a primary and send it back. And thus, they receive a all the chunk locations of all the replicas. So, the client must to write. So, first it has to query the master to figure it out how the primary could be and the primary save the master. Either it is a lease holder or the primary the master decides who could be the prime primary. And then all the chunk locations (Refer Time: 43:47) and then as soon as the client gets it starts pushing it immediately. Now what is that they similar to do in the case of if you remember we were doing virtual synchrony models. There you notice that are 2 different things. There is a the some order in which some message goes to nodes, but there is a delivery order also, the 2 are different.

You get to the information, buffer it somewhere and then that delivery order tells you in which order it has to be given to the application. Same here also as soon, as we get the information from the master, about the primary and chunk locations, the client can push data to those replica chunk servers directly, without waiting for a particular ordering. But once the data has been pushed out as soon as possible because these are all large files lot

chunks 64 megabyte, that is why the idea is to push it to the clients as quickly as possible. Replicas chunks as quickly as possible. And then later what happens is once the data is in all the (Refer Time: 44:57) in all the replica chunk servers, then the write actually write operation actually the is committed in some case.

We will look at that how that happens. So, client pushes data to all replicas. So, immediately without waiting for any pertaining or anything of that kind, as soon as if you have got do the replicas send it out immediately. Basically, is decouple data flow from control flow to improve performance by scheduling expensive data flow based on network topology regardless of which chunk server is primary. I can one of the things are decided in this particular design was that through pot is important very critical. So, the idea is that there is a control flow, control is what it basically tells you in which order something has to be updated, but there is also data flow. Which basically tells you how to move this 64 megabytes or whatever amount of information move across the network. So, if you want if you are doing big writes, you have to make sure that your network and is used to the best it doing best possible way. And the idea is that the topology of the updates.

Should be determined independently from who is a primary. So now, the way this is the data is updated is based on the network topology, somebody else is deciding. This guy is near to this guy therefore, you should go through him for example. It might be that the primary is 2 steps away from the client. Whereas, one of the replicas is only one step away. So, the way this they do it is they send it to the first replica even though it is not the primary. And then that replica will actually in term forwarded to the primary.

So, it is not by it is not the pay they you sent to the primary and primary will send it to the replica even though the replica is nearer. So, this is not a good idea for when you are transferring a large 64 megabytes chunk (Refer Time: 47:07) or big writes choose writes. So, that had to decouple the data flow from control flow to improve performance by scheduling expensive data flow based on network topology regardless of which chunk server is primary. Now what happens is that. So, at this point the client has already figured out who the primary and to chunk locations are the data has been pushed to, all the primary is a secondaries it has been pushed all in. Now that write has not at actually commenced all that is happened is that whatever that has to be written is now sitting all

the secondaries. And primaries one primary or secondaries. The data is already there.

Now there at actually (Refer Time: 28:04) the acknowledgement happens comes back to the client saying it has written, these date that had to be written. It has not at made to the file, this has gone into buffer in some sense. Sometimes LRI buffer is sitting there. That is all it has gone to. So, once all the auto acknowledgments come back to the client saying of the data has been written, then the write actually starts, this is part. Then it says the write command is sent to the primary. Primary picks a serial order for all mutation to chunk.

Now the primary the client is telling the primary I have sent all the data to everybody, now you decide the order and actually do the actual write. So, primary picks a serial order for all mutation to chunk, all replicas follow this order when applying mutations. So, write acts come back after the data is been transferred to all the secondaries and the primary a client, now sends a write. A primary now decides order write it has to be that A has to be done first followed by B etcetera.

Sorry I made a mistake. There are multiple chunk writes are going on. And those that is what is being ordered, sorry I made a mistake. So, you are having multiple writes concurrent writes are going on, and you are ordering them, and each of those secondaries will actually of try those updates in the same order. So, everybody every client will actually see all the updates in the same order. That is basically what it is. And this is possible because all the datas already sitting there. This like in the case of our group communication systems, right. In the virtual synchrony model.

There is a messages are sent, they are going to be let us say buffered, and then based on the ordering model weather is total order or it is causal order, then those messages are given the order that is specified by that model something you assume that it is happens here. So, and then once the secondaries have been told to write, then they will again respond by saying that I am done, that is what this is. And once the primary is done, the primary get acts from all the secondaries that they have done updates of the chunks, then the primary again comes back to the client saying (Refer Time: 50:41).

Now, this can incorporate either done or something happened bad here, that is going to be

an error. Out here there as you can have. So, all the replicas follow this order when applying mutations. What there can be errors? Basically, write may have succeeded at the primary and some subset of secondary replicas. If they does not succeed at the primary cell the whole thing is off, that is the obvious case. So, it may have happened at the primary, but a few of the secondaries not everywhere modified region. Left in an inconsistent state, that is the reason why you mentioned that it can be inconsistent because some places have the new information completely some have partly old informations some have partly old information that is why it is inconsistent modified region left in an inconsistent. So, client code handles such errors by retrying failed mutation.

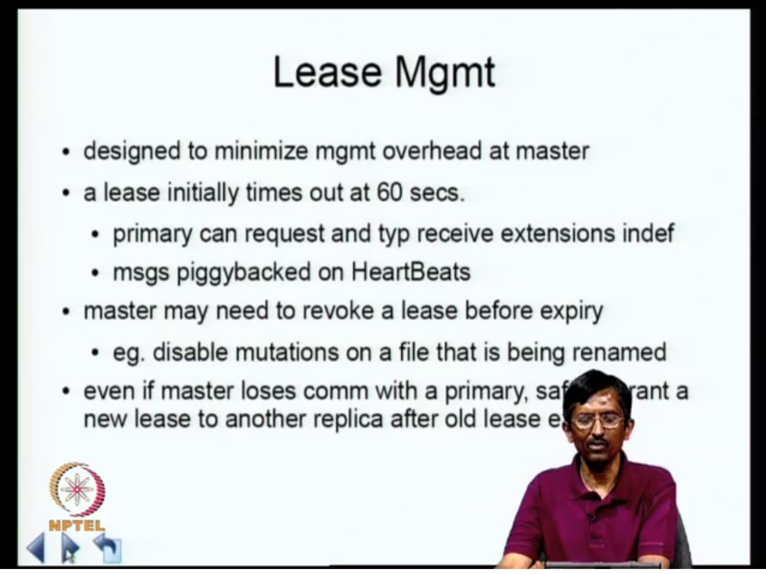
So, the client was the party who initiated the write it again tries to write data. So, what is the what is the model here? The minute there is an error because only some party succeeded in doing it you can retry the whole thing again. You it maybe because some data could not make it. So, it tried re sending a data or it could be because some other errors. So, if you e even after re sending data if it does not succeed, then you retry write from beginning. Because initially how did with they started they started with a client write that is where we started. And you it could have been broken up for (Refer Time: 52:37) of reasons. So, you are doing one thing at a time, and then again if they are unable to continue, because of repeated errors right. Who try and see if you can start a new write.

So, the file region may end up containing fragments from different clients, but replicas identical. So, our basic problem is that because of some of the errors, right it may contain fragments on different clients, but the you do not report success until all of them act back same definition writing it. So, the that is why the replicas are identical. So, region maybe consistent that is everybody say same information. But undefined state in the case of failures this will not even be consistent, it could be inconsistent. So, here is the case when you talking about concurrent writes happening in which make the file region in undefined state. So, basically the global mutation order is this chosen by the lease grant order chosen by master. So, it tells you who actually gets a lease. Therefore, that decides the order, and within a lease serial numbers assigned by the primary because of primary decides which order each of those mutations have to take place that serial numbers here.

So, basically if you have multiple concurrent writes, the way the master gives the lease

decides who actually that started. And then within the release the primary orders various updates various mutations, and then that same order is followed by all the secondaries. So, the 2 levels at which things are attempted to be controlled is been with respect to order.

(Refer Slide Time: 55:07)



So, I think I will continue next with in next class I will try to finish the rest of the google file system. There is lot more things about this management, then issues about garbage collection, and how to handle failures. Those things these kind of things also issues. So, we will talk about all this later. And hopefully we are done by next class, will be done.

Thank you.