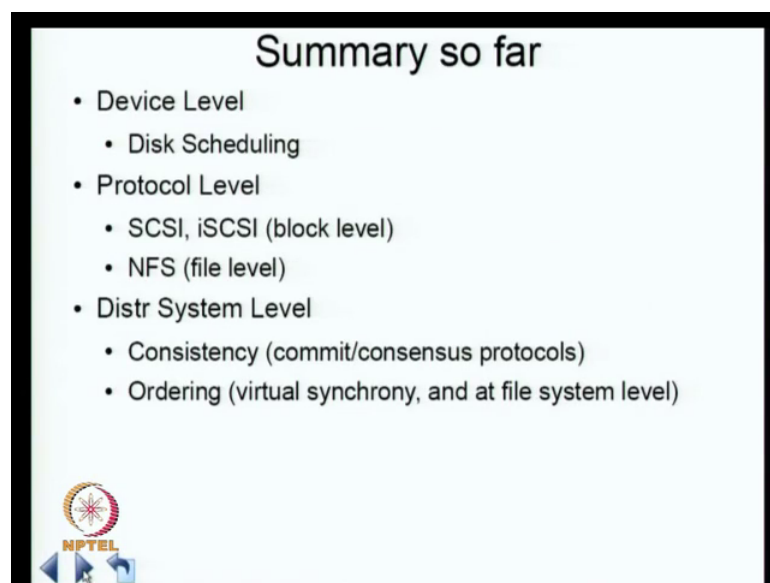


Storage Systems
Dr. K. Gopinath
Department of Computer Science and Engineering
Indian Institute of Science, Bangalore

Highly scalable Distributed Filesystems
Lecture – 39
Highly scalable Distributed Filesystems _Part 1: The Google Filesystem (GFS)

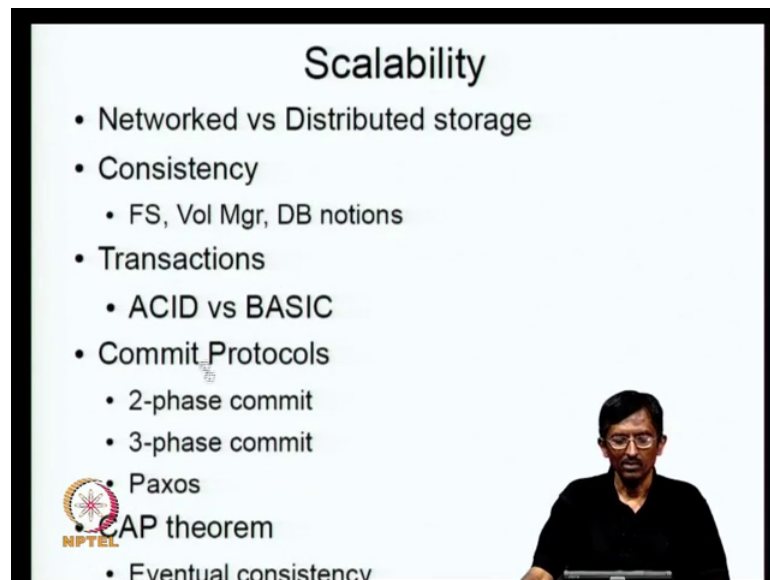
Welcome again to the NPTEL course on storage systems.

(Refer Slide Time: 00:27).



So, I think basically into summarize.

(Refer Slide Time: 00:34)



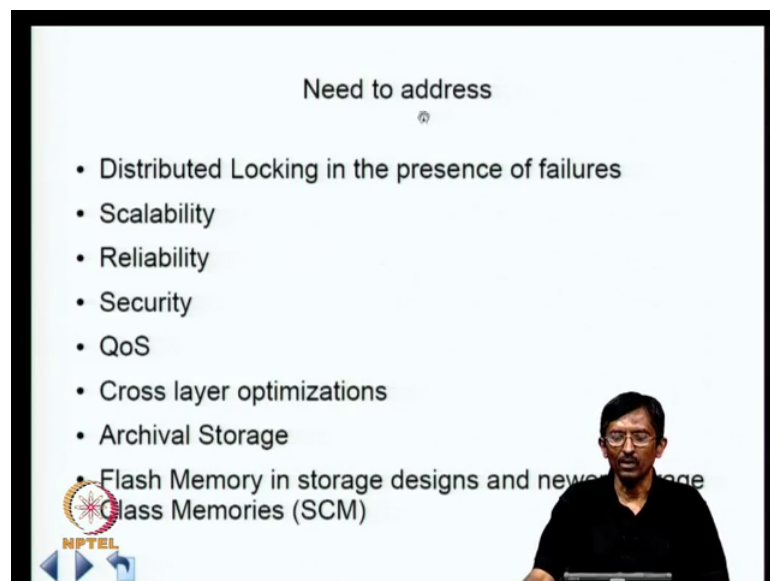
Scalability

- Networked vs Distributed storage
- Consistency
 - FS, Vol Mgr, DB notions
- Transactions
 - ACID vs BASIC
- Commit Protocols
 - 2-phase commit
 - 3-phase commit
- Paxos
- CAP theorem
- Eventual consistency

The slide includes an NPTEL logo in the bottom left corner and a video inset of a man with glasses speaking in the bottom right corner.

We looked at some aspects relating to consistency, some commit protocols.

(Refer Slide Time: 00:40)



Need to address

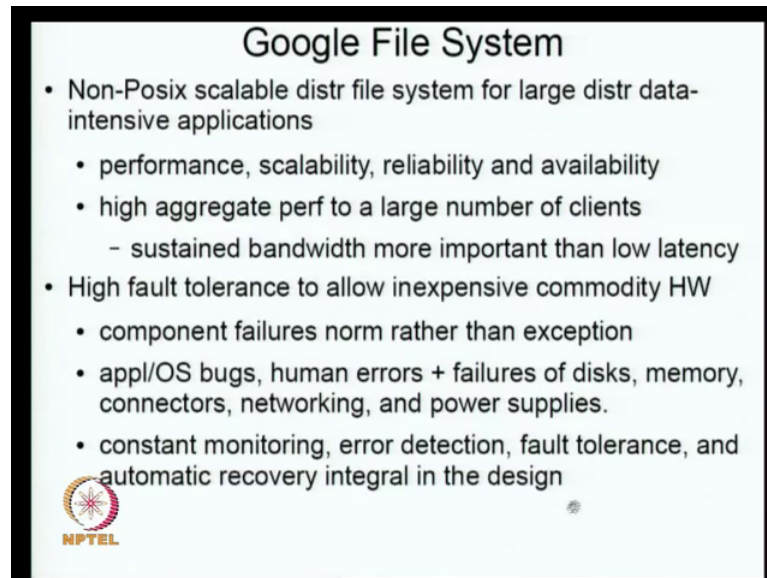
- Distributed Locking in the presence of failures
- Scalability
- Reliability
- Security
- QoS
- Cross layer optimizations
- Archival Storage
- Flash Memory in storage designs and new storage Class Memories (SCM)

The slide includes an NPTEL logo in the bottom left corner and a video inset of a man with glasses speaking in the bottom right corner.

Then we said, we had to address some issues; like distribute locking scalability reliability etcetera. So, today we will try to take a look at this part scalability, reliability. We took some notice of this particular problem in the previous class. I think we will revisit again later, but right now let us look at these two parts.


So, as a way to address this issue, I thought we will try to. We started looking at this.

(Refer Slide Time: 01:15)



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

And I want to really today talk about one good example of a reliable and scalable system, and this is the Google file system one can say that this is, as far as I know this is a largest file system to let. I am not sure Facebook what exactly their system, the size is, but at least for some time, at least Google file system of the largest file system.

So, we will try to look at how the design of this and this particular file system, we look into some detail. So, we will at least take two or three classes to go through this, because go into some reasonable detail may not really examine any file system in sufficient depth. So, I think, I thought I would take Google file system as an example. First of all, the Google file system is a Non Pos Posix system; that means, your application has to beware that, it does not have the kinds of things that Posix file system will support. For example, the semantics of, like deleted, but open files. All those things are certainly not going to be supported here.

The most important thing is, it has to be scalable distributed, and for data intensive applications critical things are performance scalability reliability and availability performance, because this particular file system is supporting one major application, that of searching the word. And since a critical part of this service should be. It should be highly performant, it should be able to supply the search results really quick, in spite of the massive amount of data that is there. So, performance is very critical scalability, because the web is growing at a very high rate. I am pretty sure that when the web

started, but 94 or something, probably you could have got some hundreds of megabytes under cross, taken care of everything very soon it became gigabytes and terabytes and petabytes ok.

Now, it is certainly in multiple petabytes. Now tens of petabytes probability, because there is so many disks, so many components. The chances of it working completely correctly are extremely small. So, you need to have make sure that the system is reliable just available, because otherwise the service or such service will be unavailable randomly, which is a problem. So, it has to be high available.

Most important thing is, it has to high aggregate performance to a large number of clients, because population which is using the Google file system, could be at any point in time, some multiple millions, tens of millions. So, you need to figure out how to supply this, and sustain bandwidth is more important than low latency. This also is important low latency, but critical thing that they have tried to do, is to have a sustained bandwidth from the file system. The low latency is handled through some of the mechanisms. For example, by using 7 parts, a system cooperate from memory to replicate. There is other things that the file system, per say the more important issue is sustained bandwidth, and one of the important thing about this file system is that high fault tolerance to a low inexpensive commodity. Hardware is a very critical part of the design, because when Google started out, they were a small company. They did not want to go for extremely expensive commodities, expensive storage systems. For example, if you go with a storage area network, it is going to be fairly expensive. They did not want to go the talk. They wanted to do it slightly cheaper ok.

So, the idea here was to find a way of in designing applicational file system together. So, that you can use fault tolerance, to able to use inexpensive commodity hardware; that is a basic thing that I write down, and I think they succeeded quite well. Let us say transaction, what they did and how they did it, that is that. This is very different design space then a small businesses, or let is enterprises, because they do not need a distributed file system per say. They can deal with reasonably centralized types of systems. The certain interesting performance and their interested not so much in scalability. I think it is not they do not grow that rapidly, its I think there is some of them might, but many of them have. They grow enough sort of around 10 percent growth per year, which is sort of tolerable you can handle it ok.

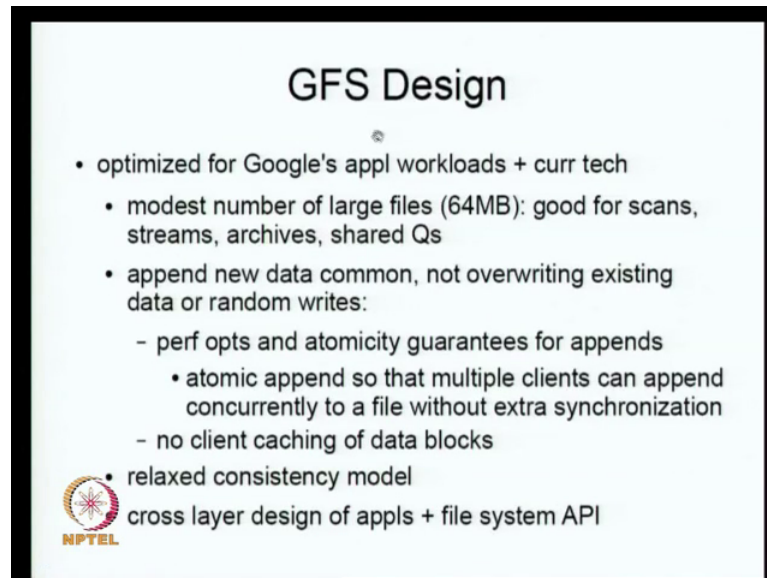
So, they do not, might not require this. For example, you do not need to scale from gigabyte to petabytes that they might not, that might come to some manipulation. So, for those kinds of systems, I think well engineered science solutions could be useful. Here the design space is different. So, the idea was to take inexpensive commodity hardware, and some are merit with a well-designed highly fault tolerant system. So, that the fact that inexpensive, or it is not specialized, components can still deliver the goods.

Just let out understand what we mean by inexpensive commodity hardware. You will see that in the case of disks, you will find certain conditions, you can find what are called enterprise disks or the common one enterprise or consumer disks. Typically the enterprise disks are much more expensive. They are carefully selected for vibration tolerance and with other kinds of things. Whereas, inexpensive disks, they are let us say, they not checked for some of these things. So, idea is to say that, let us use inexpensive disks or inexpensive components in general and build a highly fault tolerance system. So, that you can get the design space the design work.

So, one of the important assumptions in this particular file system is, component. Failures are the norm rather than exception, and it can be due to many reasons, it can, because of bugs in applications or operating systems kernels, wherever human errors failures of disks, memory, characters, networking, power supplies all kinds of things.


So, only solution to this problem, is to constantly monitor things. Do a like detection fault tolerance, in case things go bad automatic recovery etcetera, all those things observed critical. So, if you have a very large file system, very large systems, planning the web and. So, it is basically you need to do this, some of the systematically. So, what will do is. So, start looking at some of this aspect.

(Refer Slide Time: 08:46)



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

So, what is the high level GFS design goes. It was designed for Google's application workloads, and current technology, and possibly try to accommodate future trends also. That is what they try to do.

So, one let us look at one major decision that may talk modest number of large files. We are not talking about a trillions of trillions of files. They talking about millions, or slightly closer to hundred familiars of files, rather than trillions or 10 to the power 15 times of file that, because the web is big. So, can in principle imagine that there are trillions or hundreds of trillions or even more a files, but they decided that. That is not right way to do it. they decided that you will have a modest number. What I mean by modest is, hundreds, some millions or about billions, not much more, not in some terms of trillions, or you know 10 to the power 15 tons is of say, and large files 64 megabyte each.

So, basic idea is that, if you go for large files, its good for scans, streams, archives, and what are called shade Qs, that are there between producers and consumers, basically often dense. What happens is, that is searching for something, and you want to do what is called scans, and if you are able to get a large file in place, you can do much better when it comes to read ahead all this kind of issues, it comes out to be much better than scans. Then you have a large file then scattered files here and there.

Lots a times data is produced in streams. So, want to be able to. I am going to access them in a particular sequential manner typically. So, writing it in large files situation, archives by definitions are large, and they are also sequentially return or write. So, shared qs are those systems, of which multiple producers and consumers can put stuff into a particular file and these. So, this also requires large files typically ok.

. So, one basic, and you can see that this is quite different from a regular Posix kind of file system, that we use normally. The file system sizes are not 64 megabyte. Typically we assume that it is closer to, let us say multiples of 4 kilobyte or 8 kilobytes only. Nowadays with widespread use of music files. Like Mp 3, the size of files has a typical file, but muscular nowadays using, because a lot of people are listening to music, its closer to 4 to 8 10 megabytes that kind of stuff, but in the past, before the music point became common, most of the design space was to handle 4 kilobytes and 8 kilobytes, or somewhere in that region; that is the size the factory before.

Other difference is that, append new data is common, not overwriting existing data or random writes. So, basically you are adding new data, rather than modifying data; that is why it is designed for handling appends better than over writes, and they do lot of performance optimizations for appends, and also they give you atomic guarantees for appends there is, when you are concurrently appending multiple parties are appending at the same time, then you want to give some atomicity guarantees. So, they are designed a specific operation for atomic append. So, the multiple clients can append concurrently to a file without extra synchronization.

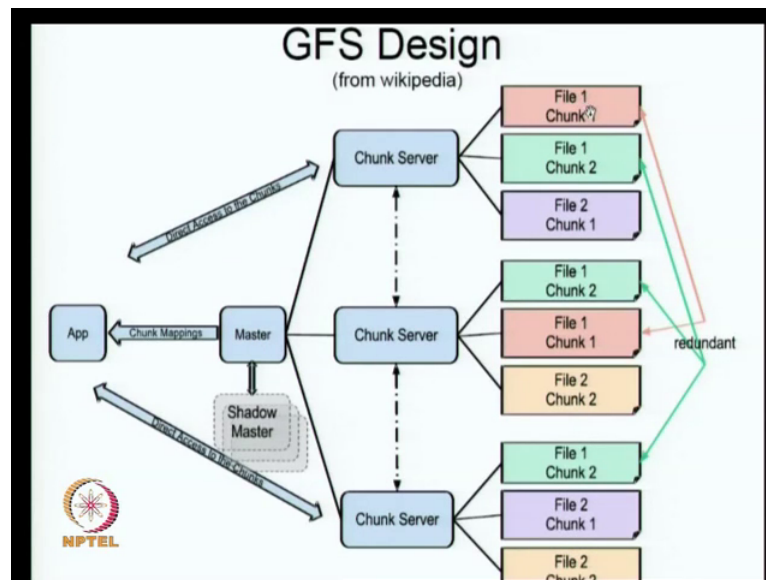
I think there is somewhat similar to, if you have studied computer architecture, there is you have some synchronization, sometimes you have to do. It turns out to be expensive, sometimes people have come up with slightly different models of synchronization. Like for example, you might come across, fetch an ad, a fetch an ad, it allows multiple increments simultaneously, and then each client will get a consistent version of how it was. I did took that shared variable, everybody gets different word, different values, but there all happen in some order, its not guaranteed which order it happens, but everybody gets some serialized version of the ads, and each one of them gets a distinct value of the ad here also. Similarly what it means here is, that any number of people can be trying to append you, get some serialization of the appends. Some serialization we do not guarantee which one it is ok.

And, but everybody sees that they appended a particular offset every, but every party gets a different place, where they appended. So, that way you can essentially do any extra synchronization. You just can go ahead and start appending, and the file systems taking care of it again, because you are talking about a large files. It makes no sense to catch them, because by definition you have a large file, and you are doing mostly scans. Some those kind of things; the Larry part of it, is quite the output best is not to do any caching of data blocks. You just pick up big chunks, use it and then let us say throw it out. I think there is some caching going on, but this is below the, there GFS file system, because they use Linux as a node, where the GFS is working on top of it.

So, those cache that caching that happens, happening through Linux, not to GFS. They also have a failure relaxed consistency model. We will discuss this again with respect to appends. The second important aspect of the design, we will look at this particular consistency model also, and fundamentally I said discussed earlier. These are cross layer design of both the applications, which are going to use this file system. And the specific file system capabilities are there. It is basically they have designed the file system api, knowing fully well. What kind applications will be targeted? Of course, this is a, in any, only file system the set application skip changing, but they had a certain idea about the kind of application they wanted target. So, they attempted it inter plus split some of the design decisions, especially supporting only large files. It has been found to be problematic, and they are actually revisiting these issues, and they changing things here, there change quite of a things, some in some areas. Here I can, we will have at the end of this particular discussion on GFS. We will study exactly the kinds of a problems that GFS has encountered, then kind of remedy that people are looking. It will just look at doubts.

Most important thing is that, it is a cross layer design. What is that mean. It means that knowing fully well the application space, the kind application that I written, you try to tailor their file system API for those kind of things. So, that there is some synergy between both these things. Whereas, Posix; for example, is a general purpose thing, it does not. It has several moral applications, but typically desktop applications. So, that its design space point is somewhat different from what these people are trying to do.

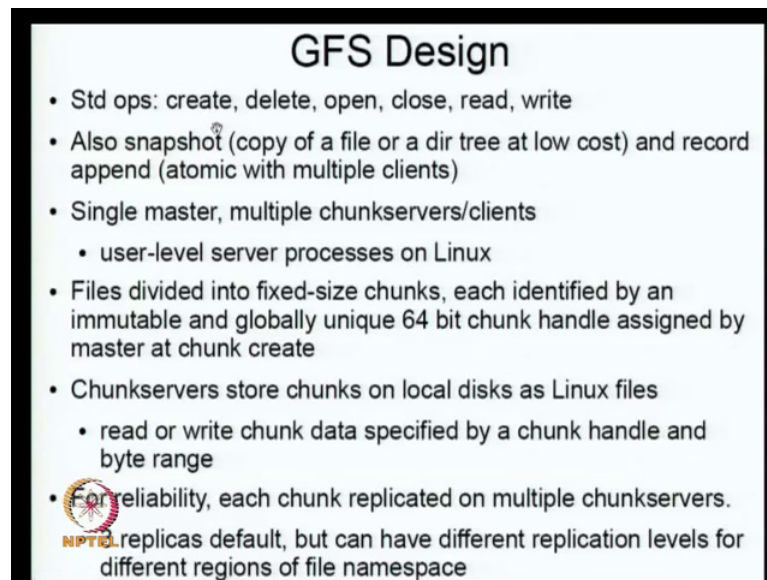
(Refer Slide Time: 16:42)



So, will also look at how this is done. Again we look at this figure in the very first lecture I believe. So, again we just quickly look at what is going on. You had an application, It could be search application for example, and basically the application talks to the master, it is a single master, and you will not discuss this right now. You will just think about later the single master to makes, to avoid consistence issues single master, and application will ask details about a files chunk locations; that is what this is basically an application. Tries to figure out a given file is chopped up into multiple chunks, and they are stored at multiple nodes. And the first thing to do is, if there is a file you talk to the master, and find out where the chunks are. An application can directly talk to those chunk servers without going to the master, to get the file chunks, and because of reliability reasons, we need to have some replicas, and they are done a replicas. For example, file one, chunk one. This also replicated here. File one chunk two is replicated three times. This is two times etcetera.

Since (Refer Time:18:20) the high level design, we will go into more details soon.

(Refer Slide Time: 18:25)



GFS Design

- Std ops: create, delete, open, close, read, write
- Also snapshot (copy of a file or a dir tree at low cost) and record append (atomic with multiple clients)
- Single master, multiple chunkservers/clients
 - user-level server processes on Linux
- Files divided into fixed-size chunks, each identified by an immutable and globally unique 64 bit chunk handle assigned by master at chunk create
- Chunkservers store chunks on local disks as Linux files
 - read or write chunk data specified by a chunk handle and byte range
- For reliability, each chunk replicated on multiple chunkservers. **NPTL** replicas default, but can have different replication levels for different regions of file namespace

First of all, it does support a certain set of common operations that everybody users finding use the file system like create, delete, open, close, read and write, but does not support all the Posix ones, even the semantics of all these things, also is quite different. For example, at least in the written documentation as seen about GFS, they were never talk about a link. For example, command link or symbolic link all over, they could be having it's, but not, I am not seeing it ok.

So, those things are all post you can call, it is you need to, you need to have those things for Posix compatibility. They also have certain other types of operations. For example, you can create a snapshot, the idea being you want to able to take a some kind of a copy of a file or directory tree at low cost. It is done through what is called copy and write. Its not really a physical copy, it is basically a logical copy and. So, there is some, sometimes you can take a snapshot. So, that you want to be able to see the particular file or a directory, I said exists a particular point in time and then, but this particular operation has to be low cost.

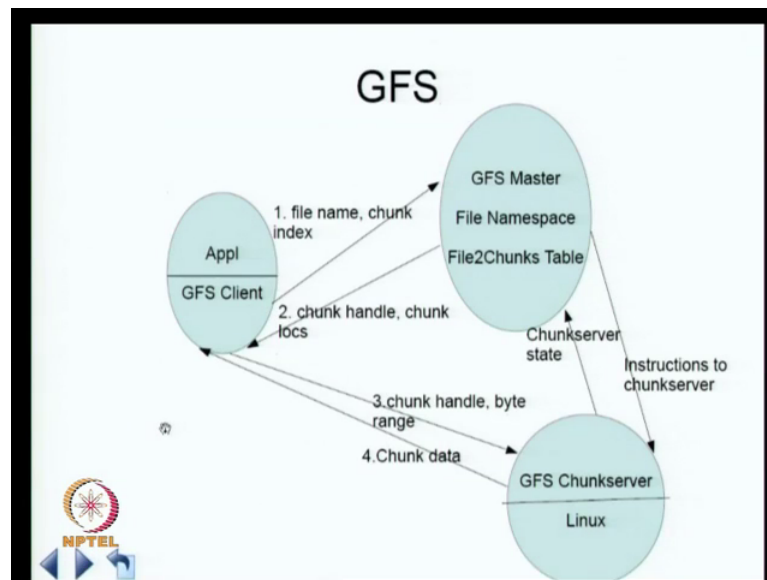
So, for it to be low cost, you have to use some techniques, like copy and write; otherwise this is not a feasible solution, because that directory tree can be quite arbitrary large as a measure, already they have some specific design components for handling appends, and it has to be atomic, in spite of multiple clients, these one, other critical thing that they provide. So, In one critical part of the GFS design is a, it is a single master and multiple

chunk servers and clients, that n number of clients, there are multiple chunk servers, but this one is a single master, and all these things whatever stuff that this running on. All these machines is, all user level server process on Linux, typically you will find a file system, is in the kernel especially at the desktop, but here is a case, where everything here is user level, everything is running on Linux, and the extensionally using Linux as a platform on which all these masters and multiple chunk server clients are, all running on top of it.

So, files are divided into fixed size chunks, and each is identified by an immutable and globally, an 64 bit chunk handle assigned by master at chunk create. So, when a create a chunk, you have a unique name for it, and it is not reused again. So, I guess it is 2 to the power 64. So, if you run out some day, but looks like it will be not soon. So, that is why they can handle this. So, chunk servers store chunks, and local disks as Linux files. So, various chunks are there, and each of them gets a number, that basically that you make 64 bit chunk handle, and that is how the number has, and they are stored as Linux files ok.

So, each Linux is basically, each chunk is basically a file name with a unique chunk handle. So, when you want to read or write a chunk data write, you got basically what you have to do, is you have to specify a chunk handle and a byte range. So, if you look at NFS, there it was a file handle, and if what is the offset right, here basically what have you do a chunk handle. So, again for liability, each chunk replicated on multiple chunk servers, typically three replicas default, but you can have different replications levels for different parts of the file name, space, it is up to you to decide.

(Refer Slide Time: 22:41)



So, let us just quickly look at how this works. So, the application it, there is a library by which the application can make calls to the, give his client. And first you request a file name and a chunk index, it was the GFS master, and this GFS master given a file name and chunk index. It gives you a chunk handle, and a chunk locations where they represent. So, basically you are essentially saying, given a name of a file and which chunk you are interested in right. Then the GFS master will give you a unique set of the unique name for the particular chunk, and were located, because they replicated. There could be 3 4 places where its replicated, you get the locations.

Now, you basically take the chunk handle, and the byte range, and send it to a GFS chunk server. And then GFS chunk server talks to the Linux system below it. And then it is a regular local Linux file system, to pull out a chunk, and it goes into user space process, and that is basically copied out to chunk data, and then sent out to GFS client, from where the application gets it here. You find that the GFS master has completed control over that file name space, essentially the directory structure. And it also keeps track of the metadata corresponding to where the chunks are. So, there is a table which is file to chunks table ok.

So, again there is some communication between GFS master and chunk servers. I just had only one server chunk server. There are many of them. There is only one of these, and there are many of these, there are many of this, many of these produce, only one of

these. So, there is some communication between the GFS master and multiple chunk servers with respect to the state etcetera. This roughly the high level idea about the system. The basic idea is that, you go to the master with the request about where the chunks are located, and you also get a chunk handle, and then you go to the chunk locations and present the chunk handle, and you get the data back.

(Refer Slide Time: 25:35)

Large Chunk Size

- reduces clients' need to interact with master
 - reads and writes on same chunk require only one initial request to master for chunk loc info: appls mostly read and write large files sequentially
 - for small random reads, the client can cache all chunk loc info for a multi-TB working set.
 - with a large chunk, a client is more likely to perform many ops on a given chunk
 - reduce netw overhead by keeping a persistent TCP cnxn to chunkserver for an extended time.
 - reduces size of metadata on master: all metadata in memory!
- but hotspot if a file a popular single chunk executable
 - popular executables need higher replication factor
 - also, stagger application start times
 - (future?) allow clients to read data from other clients

Let us just see one of the important design aspects of this large chunk size. As I mentioned this large chunk size has certain difficulties. We will discuss it later what the difficulties are, but in their design, they decided that it was good, because it reduces, clients need to interact the master, because it brings it in one big shot, then you do not have to keep interact with the master, because there is only single master. We keep on interacting with the master, then master will be over loaded. So, that is one part of it ok.

So, it reads and writes happened to be on the same chunk, there of course, only one request to master for chunk. Basically in some sense that is read ahead going on, in some sense. That's also reason being applications mostly read and write large files sequentially. For small random reads, client can cache all chunk location information even for a multi-terabyte working set, because the size is big. You can cache all the information even for a multi-terabyte working set, because each chunk is big 64 megabyte. So, the number of even if I multi-terabyte working set, the number of chunks is going to be manageable. It might be in that region of, let us say, not about millions at

the most. And if you have million kind of chunks right, if you have a multi-terabyte working set, it have million, let us say chunks. Each chunk location information, let us say I say some tens of bytes. Then it still feasible for it to cache it. Its definitely possible to cache it.

So, again there is other reasons also why this is useful, with a large chunk a client is likely to perform many operations and given chunk, and there are issues some about networking also it make sense, because you can have a large with persistent TCP connection, for longer period of time. It helps in efficiency of TCP itself. Most important is, to have a large chunk size, it reduces some size of meta data on the master, and actually now you have the luxury of keeping all meta data in memory. I can look at this in some detail. This is one of the critical aspects of the large chunks size. You basically can keep all metadata in memory. Once you keep all metadata in memory, then you can do lots of interesting things fast in memory itself. You want to rebuild something, everything is in memory. So, its fairly trick. you are not now limited to disputes

So, even one sense, large chunk size is connected with all metadata in memory. And it turned out the current technological solutions, it made it possible. The same design would not be feasible some other term. Nowadays memory is reasonably cheap. Therefore, this is sensible thing, but there are some, at the time when the design the system itself they came across certain problems. I will later talk about the kind of problems that exist as of now, because designed about 2000 or slightly earlier. At that time itself the came across certain problems, but post this particular paper was published in 2003 S O S P, all the details. And around that time already they had seen some problem, but later also there are some other problems which the designer sound come across, we will discuss it later, exactly the kind of forms they came across. What is one problem they came across at the design time itself ok.

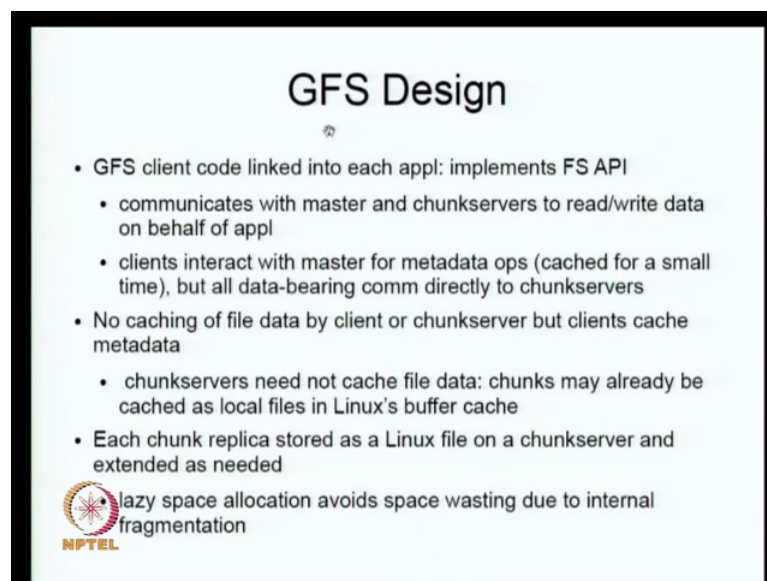
Suppose you have a an executable, the very popular executable, and it is smaller of that its fixed into one single chunk. Then it turns out that if there are many clients, wanting to execute that particular, popular executable, then all of them will start going after that particular node with that particular chunk. So, that is why they essentially had to ensure that, popular executable are given higher replication factor. And also when applications are getting started, you make sure that some of sense, their some kind of back off. Not everybody starts in the same time. You need to ensure that, even if there are hundred

clients wanting to, when they just started. Let us say there is some system, and all the client start at the same time for some reason, might all the go try to get it from one node, then we will get some more load of the particular system.

So, one thing to do it is to somehow recognize a situation and stagger how they actually access it. Of course, the sensible thing would be to, have some kind of a tree structure. So, one guy has caught a copy, some other guy should be able to get a copy from them. So, basically create a, instead of going to the same guy, if you have somebody has read it, some of the client is read it. Other party should able to get it from the other clients rather than slightly from the node that has to chunk.

So, what this, they decided was to cost me them have done, it its possible one could think of doing it. They have left it as a possibility.

(Refer Slide Time: 31:05)



The slide is titled "GFS Design" and contains a bulleted list of design points. At the bottom left, there is an NPTEL logo and a note about lazy space allocation.

GFS Design

- GFS client code linked into each appl: implements FS API
 - communicates with master and chunkservers to read/write data on behalf of appl
 - clients interact with master for metadata ops (cached for a small time), but all data-bearing comm directly to chunkservers
- No caching of file data by client or chunkserver but clients cache metadata
 - chunkservers need not cache file data: chunks may already be cached as local files in Linux's buffer cache
- Each chunk replica stored as a Linux file on a chunkserver and extended as needed

NPTEL lazy space allocation avoids space wasting due to internal fragmentation

I can as we mentioned before, this GFS is a user level design. So, EFA GFS client code is linked into each application that implements the file system API. So, this client code communicates with master and chunk servers to read write data on behalf application. Again, this kind of design has been attempted in other areas. For example, if I parallel file system you find that this current design is quite common also, and the reason why that is useful there is, because if you do it add up at the user level, then you can do what is called user level networking. So, it does not go through the, you do not have a multiple copies if you go through the kernel ok.

So, you can have also a user level file system, where things are directly mapped across the network interface cards. So, that you can avoid the copies and touching up all the data, data bias some other kernel code. So, this is a not a unusual design it is for many people have done this, basically the fact that the file system actually is in user space, and that client code actually is linked into each application. So, the clients interact with master for metadata operations, and these metadata are cached for small plates of time. I can just like NFS there is some element of caching that goes on, but it is only for metadata. So, there are still here also there are some issues of consistency, basically they have a time of mechanism, but it is possible that you can indeed get into the kind of problems NFS had, but the applications are aware of all these things. Its not as, as mentioned this particular system, the application has been designed, fully knowing what the file system is capable of them. It is not like a Posix system where, the applications written Posix model in file system has to implement it. Here the application has to take the trouble figure out, what the file system is there.

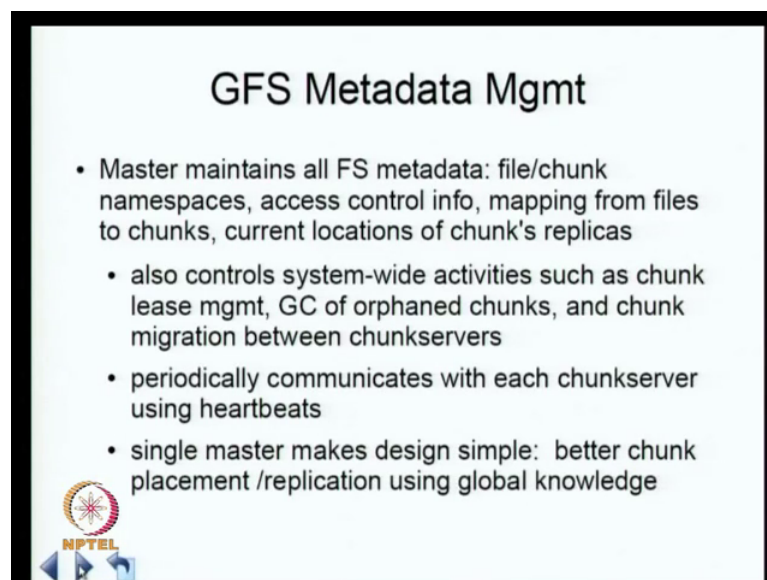
So, clients interact with master of metadata operations, but all data bearing communication directly to chunk servers. The most important thing is not this since is, this single server you do not want all the communication to go through the master. For example, here if you think about it right, you do not want all the data to go through this. You want to get the metadata of here this side, but once you got the metadata, you want to directly go to chunk servers. You do not want to go through the master, whether this one is single, but serious battle and (Refer Time:33:53). There is only a single master, maybe only single node.

No caching of file data by client or chunk server, but clients cache metadata. So, chunk servers which a user process does not need to cache file data, but they may already be cached as local files in Linux cache buffer cache. The cache server these a, as mentioned this is a, a chunk server is basically a user process and it is not doing any caching of anything. And it may be there as local files in LINUX's buffer cache. Each chunk replica stored as a Linux file on a chunk server and extended as needed.

Now, as a mentioned it is 64 megabyte, each file is, each chunk is. Sorry each chunk is 64 megabytes, but they do what is called lazy space allocation. So, it is essentially, it is given space, but it is not allocated is that, in some sense, means that it is not backed by disk. It is notionally there, and that is lazy space allocation avoids space wasting due to

internal fragmentation, because a very, another critical part of the data, but this is more makes it slightly more complicated also. Once you have lazy space allocation, then it you need to have some other metadata to keep track of the fact that you not, you do not have very kind, you have to allocate now and then, and the blocks may get scattered here and there, those condition, this is also will (Refer Time: 35:16), because when you are trying to get blocks at the time then you want it, if may not be contiguous. So, there could be some issues here also, but if you are that efficiency willing with large files right, you will be requiring in large requests also will come. So, when that fragmentation or a scattering of files is slightly to the also less in problem.

(Refer Slide Time: 35:40)



The slide is titled "GFS Metadata Mgmt" and contains a bulleted list of responsibilities for the master node. In the bottom left corner, there is an NPTEL logo and navigation icons.

GFS Metadata Mgmt

- Master maintains all FS metadata: file/chunk namespaces, access control info, mapping from files to chunks, current locations of chunk's replicas
 - also controls system-wide activities such as chunk lease mgmt, GC of orphaned chunks, and chunk migration between chunkservers
 - periodically communicates with each chunkserver using heartbeats
 - single master makes design simple: better chunk placement /replication using global knowledge

NPTEL

So, let us just look at the GFS metadata management. So, the more important thing about this design is the master maintains all file system metadata, all of them. The file and chunk namespaces. For example, all the file in, because there is going to be a directory structure of all the files is current, that metadata for all the files, means file structures that all be sitting out here, something about chunk namespaces. Basically the file has got multiple chunks. Somebody has to keep the track of which file, now what chunks are there in each file. There is mapping from files to chunks.

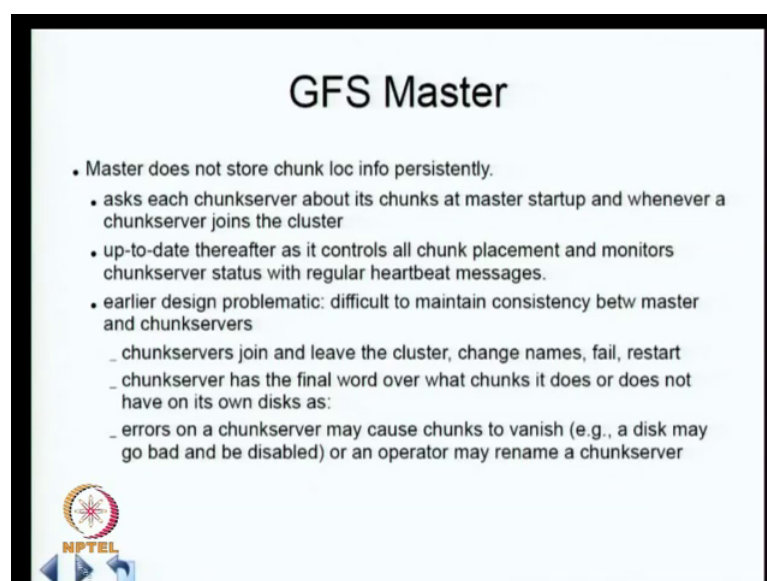
Again things like access control information also has to be kept here, and current locations of chunks replicas. All the metadata is going to be kept here. So, because only master keeps it, there is no need to have in distributed coordination, that makes induct

into right. So, that is why it, the single master design, is well known to be a single point of failure. There is a master of dies, everything goes down right, but decided the single point of failure problem is serious, but you get so much advantage by, not worrying about this aspect, that you can probably do better. So, I think that some of these is what they gone for it. There is several problem that is why there is, will come to how they handle the single point of failure issue later, by having the shadow master etcetera, will talk about it later.

So, because you have only one place, its easy to do system wide activities. I mean a simple way, because everything is happening in single place. The full state of the system is known to the master. So, its somewhat easier to design algorithms. For example, chunk least management, garbage collection of orphaned chunks, will come to why this happen later, chunk migration between chunk servers. So, this part of it we will discuss it later, all these things later.


So, other thing it also does is, it keeps track of all the chunk servers that are, whether the alive or not by sending heartbeats. So, basically the reason why everything is in the master, is because single master make design simple, better chunk placement replication using global knowledge. The global knowledge everything is sitting in one place. So, its somewhat easier to do it.

(Refer Slide Time: 38:21)



GFS Master

- Master does not store chunk loc info persistently.
 - asks each chunkserver about its chunks at master startup and whenever a chunkserver joins the cluster
 - up-to-date thereafter as it controls all chunk placement and monitors chunkserver status with regular heartbeat messages.
 - earlier design problematic: difficult to maintain consistency betw master and chunkservers
 - _ chunkservers join and leave the cluster, change names, fail, restart
 - _ chunkserver has the final word over what chunks it does or does not have on its own disks as:
 - _ errors on a chunkserver may cause chunks to vanish (e.g., a disk may go bad and be disabled) or an operator may rename a chunkserver



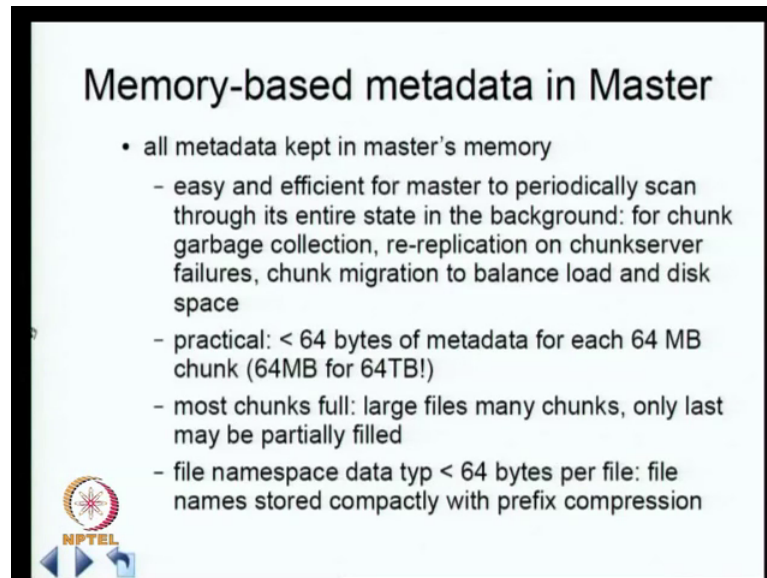
Let us look at some other design aspects. Master does not store chunk location information persistently, everything is sitting in the memory middle. Means the basic idea here is that the division of labor between master and chunk server. The chunk server is a party who actually stores things. So, is a better entity which knows about what its storing rather than the master. Master can always get a stuff from chunk server; that is idea.

So, basically when each chunk server comes up right. It can figure out what chunks it has got, and then whenever a master starts up, it can request all chunk servers to tell them what it has got. Again whenever a chunk server joins a cluster, you can as the chunk server to supply many information, because the chunk server is sitting on a Linux system, and it has persistently kept information about all the chunks it has got. So, whether it is crashed or whatever, it can always come up with its set of chunks it has got. So, whenever chunk server joins a cluster, it will request. The master will request chunk server to supply whatever chunks sequence got. And then. So, as long as a tracks, each of the chunk servers with how they come in and how they go out right. It can always keep a consistent notion of what chunks represent where.

So, basically it controls all the chunk placement, and monitors chunk servers status with regular heartbeat message. So, it exactly knows, who is out there, and since it manages all the chunk placement it knows exactly what is happening. That is why he does not need to keep chunk location information persistently in its own node. Whatever it keeps it is kept in memory. So, its very fast also in terms of sublime information. Before this they had a, they actually was storing it persistently, and that cared lots of problems what are the problem you have to maintain consistency between, what a chunk server thinks it has got and what the master has got. You need to keep on reconciling these two motions; that is a bit of problem. And because chunk servers join, leave the cluster change names fail restart whatever they are do the kind, what algorithms. And the basic argument why this makes a lot of sense, is that a chunk server has final word on what chunk it does got. Only it looks nobody can know, Proix, because you may find that the chunk server make, they made be some error some chunk server, and suddenly certain this may go back. Only the chunk server which is attached to that, which is running on the Linux system can figure out what is going on there, because normally these chunk servers, they are running on some nodes, and that node might have multiple disk; 2 or 3 or 4 disks.


So, any particular disk can go back, and this complete local at local events, which the chunk server can track. So, even if it has, in the past had some chunks, suddenly it may disappear. I only chunk server less about it and. So, it may be better for chunk server will keep track of this, and let the master get the information chunk or from a chunk server. So, they designed the redesign this particular aspect to not keep it persistently.

(Refer Slide Time: 42:10)



Memory-based metadata in Master

- all metadata kept in master's memory
 - easy and efficient for master to periodically scan through its entire state in the background: for chunk garbage collection, re-replication on chunkserver failures, chunk migration to balance load and disk space
 - practical: < 64 bytes of metadata for each 64 MB chunk (64MB for 64TB!)
 - most chunks full: large files many chunks, only last may be partially filled
 - file namespace data typ < 64 bytes per file: file names stored compactly with prefix compression



In other next issue is, as it mentioned before, all the metadata is memory based in the master.

So, therefore, it is easy and efficient for master to periodically scan its entire state in the background. It has to for example, we will look at how some chunks become garbage. So, if chunks become garbage you have to, do garbage collection. So, all these things can be done in memory, you do not have to do it to disk. So, it can be very fast. And if there is any chunk failure, and chunk server failure you might on to, again replicate whatever has been lost and some other machine. So, for doing all this kind of stuff, you will find that information being presenting in the memory is useful. They also might want to do chunk migration to balance load and displace.

So, all these things, all the metadata kept in memory, you actually is, you have a completely consistent global knowledge about where things are. So, you can just do it this go to memory and pick it up and these run some quick algorithms, and it will be very extremely fast. If I do all the same disks, you can be probably having serious trouble,

now and there. The most important reason why this all is possible is, because it turns out to be practical. Why, because for each 64 megabyte chunk, it turns out you need less than the 64 megabytes some metadata; that means, that for 64 terabytes you need on the 64 megabytes.

Even in two thousand 64 megabyte was not considered very big. So, in 2000 I decided that, if I were able to have a web, which is about 100 terabytes right. I just need about two a, what is say, I just need to have 128 megabytes of disk that is all, that takes care of memory right now, its not too huge. Of course, with now, the size is the what this much bigger, its not 64 terabytes are 128 mega terabytes, its going to be closer to some hundreds of petabytes; that means, that if it is hundreds of petabytes you are talking about a scale of this by a factor of thousands, thousands are you and who knows probably about 10,000. If you scalar by factor of 1,000, we are talking about 64 gigabyte, which is still quite reasonable. The scale of a factor about 10,000, then you are talking about very big memories, they talking about half a terabyte and memory or something or that, and that can be a bit costly, but even today is possible, its nothing difficult to imagine, because already many machines come with ninety six gigabyte standard, many of them come with.


. So, that is even why it turned out this design decision was quite practical, and it gave lots of amazing capabilities. So, one other thing what is interesting is that, most chunks are full, because they are during with large files. And therefore, only one chunk at the end might be partially effect. Therefore, it makes a lot of sense to. Actually its not, there is not too many chunks with incomplete information. There is not fully 64 megabytes. So, that helps also, in terms of reducing the metadata node.

So, I can, they use some other interesting techniques to reduce the file name, because file name in the file names have to be stored, and the method, and the master, and they made sure that it is not too big for file, and they use things like they can store names, with some compression techniques. This is also helps.

(Refer Slide Time: 46:26)

Logs

- Namespaces and file-to-chunk mapping kept persistent by logging mutations to an operation log stored on the master's local disk and replicated on remote machines.
- a log allows reliable upd of master state without inconsistencies if master crashes
- serves as a logical time line that defines order of concurrent ops
 - files and chunks, as well as their versions, uniquely and eternally identified by logical times of their create
- changes not visible to clients until metadata changes persistent
 - else can lose whole fs or recent client ops even if chunks survive
 - respond to a client op only after flushing log record to disk both locally and remotely.
 - master batches several log records before flushing to reduce impact of flushing and replication on system throughput



So, let just. So, since its a file system which gives you some guarantees, as I mentioned appends are atomic, and the, but the consistent model is a bit complex. I can, it come to that, but once you try to make things atomic, you have to ensure that operations can be either roll forward or roll backwards. So, you need some logs. So, what this data base, important metadata aspects; like namespaces and file two chunk mappings, you keep it persistent by logging changes to an operation log, and the masters local disk, and replicate on remote machines ok.

So, hopefully these things are not too frequent, because you are logging it to disk. So, its a master crashes, you can look at the log and figure what to do. Other thing that they do is, these logs, the serves as a logical time line that defines order of concurrent operations. And basically any file creation or chunk creation right. There identified by then the time of their creation. So, given that log is basically sequential. So, depending on the way in which order the operations, essentially you get a timeline, that helps also, because we notice that all the metadata operations are getting done on the server, on the master minute, nowhere else.

So, if you are somewhere able to sequential as it you got a timeline automatic timeline. Now one critical thing is, that the changes should not visible to clients until metadata changes persistent. Now one thing one should remember is that, even though certain things are getting updated, there are consistent versions of the files to the checkpoints.

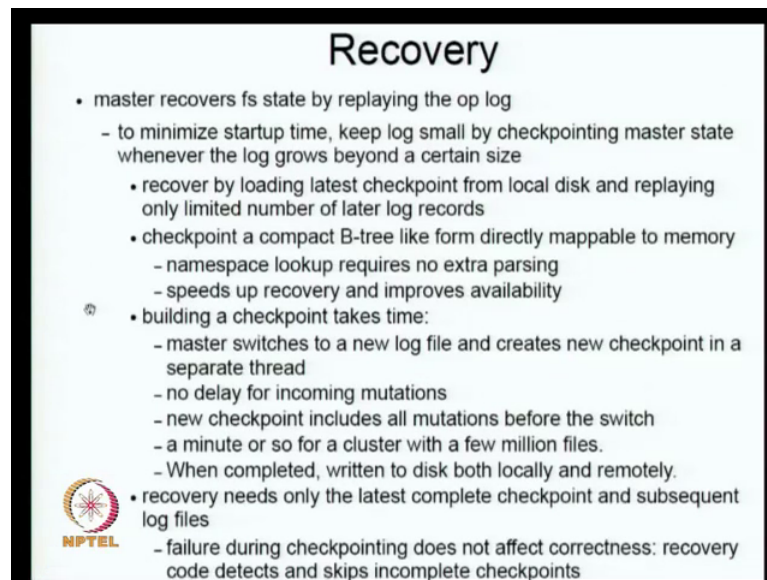
So, there are there are some readers, they can look at the consistent checkpoint rather than the current versions as they are getting updated, that reason why changes are not visible to clients until metadata changes persistent. Basically what happens is that, the clients can actually use checkpoint at versions of files for reads, and they do not have to look at, currently being modified files, appended files ok.

So, there is, clients are always able to get some information to keep possible. Of course, the reason why this kind of stuff makes sense is, because you are talking about such application, where it is not absolutely critical right, that every single data be available. There is a tolerance for certain small amounts of lack of completeness. Of course, we don't, we do it only for data, we do not do it for metadata, is only for data.

So, one important thing is, if you, if data becomes visible to clients before the metadata becomes persistent right. Then if there is a cache or whatever right, then essentially you are going backwards in time, in some sense. So, that is basically the, you can essentially lose some operations, if that happens, even if the chunk server, stresses in why they have a strict full that changes should not visible to clients until metadata changes persistent. And then basically respond to a client operation, only after flushing log record to disk both locally and remotely.

So, these things are essentially client consuming operations. So, it is a problem, that is not why the check pointing other kinds of methods actually help you a lot. Essentially are limited the latency is limited by, how fast you can log records to disk, both local and remotely. So, it is not going to be something insignificant. Of course, the way they avoid the performance bottleneck is to batch them, and such them batching it server have also happens. This of course, to be care of most file systems.

(Refer Slide Time: 51:16)



Recovery

- master recovers fs state by replaying the op log
 - to minimize startup time, keep log small by checkpointing master state whenever the log grows beyond a certain size
 - recover by loading latest checkpoint from local disk and replaying only limited number of later log records
 - checkpoint a compact B-tree like form directly mappable to memory
 - namespace lookup requires no extra parsing
 - speeds up recovery and improves availability
 - building a checkpoint takes time:
 - master switches to a new log file and creates new checkpoint in a separate thread
 - no delay for incoming mutations
 - new checkpoint includes all mutations before the switch
 - a minute or so for a cluster with a few million files.
 - When completed, written to disk both locally and remotely.
- recovery needs only the latest complete checkpoint and subsequent log files
 - failure during checkpointing does not affect correctness: recovery code detects and skips incomplete checkpoints

Now, when it comes to recovery. If you, if a master fails, you have to replay the operation log. So, it turns out it can take some amount of time, because you; now you are limited to discretely. So, to minimize startup time, keep log small by check pointing master state, whenever the log grows beyond a certain size. So, you want to, you have to replay the log, but the thing is, you cannot play the, all the operations are taking place, that is to do a checkpoint. Checkpoint means that the particular file system is that a consistent state. If you just go back to that consistent state; that means, you only have to do a small portion that metadata operation that happened after the check point, that is why we might have to do ok.

Recover by loading, latest checkpoint from local disk and replaying, only limited number of later log records. So, checkpoint is in critical portion part of the thing. And again most important is, the checkpoint is like compact B tree directly mappable to memory. So, I can, this is kept in memory itself. So, next phase look up requires, no extra parsing speeds up recovery and improves availability. So, the basically there, again they are exploiting the memory design, using the memory as a quick way of looking up things, its not on disk. And it is an of course, why this happening is, because we have a lot of memory, and it can be used for all these process, its not a design, exploits a current technical advantages. Again even this checkpoint itself is a problem, because it has to be, let us say it will take time. So, the idea is to switch to a new log file, and creates new checkpoint in a separate thread.

So, basically in some sense, as your check pointing, you create a new log file, and the new mutations, new changes are taking place in a new log file; that is why new delay for incoming mutations, whatever changes, whatever metadata operations are taking place, they are not going to be delayed by this. And the new checkpoint includes all mutations before the switch. They claim that it takes only a minute or so for a cluster with a few million files. Then completed, written to disk both locally and remotely in check point.


So, basically the idea is, the checkpoint is going to be in memo, is going to be on disk, but you can map it to memory directly if ((Refer Time:54:05)). So, there is a problem of building a checkpoint takes time, that is why you want to have, one while one is being made other there is new things are going into a new log. And if you have to read the checkpoint, you have to take it from disk, but you can map it to memory, and you can do all the operation maybe fast. So, idea basically is that, you want the on disk format to be similar to the memory format. If you do that, then there is very little work you have to do to get go. And recovery needs only the latest complete check point and subsequent log files. And again check point itself can fail, because it takes time right. By definition if something takes time there is a possibility is some error right.

So, failure during check pointing does not affect correctness, and recovery code detects and skips incomplete checkpoints. Basically idea is that, whenever you do any check pointing, you keep adding some inline metadata in the check point itself, and then when something, when the recovery code comes in, it checks was whether signatures of their. In the signature not then we can figure it out, that these is an incomplete check point something is wrong with it. So, all these things are done ok.

(Refer Slide Time: 55:25)

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: 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



I think I will stop here, I will talk about consistency manage it in next class. So, we will look at how, what is the concessional model GFS has got, why it is like the weaker model than. For example, Posix kind of models, and why there append for example exploits this aspect so that you can get high throughput for concurrent appends, you look at that point

Thank you.