

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

Theoretical Foundations

Lecture - 38

Theoretical foundations of Distributed Storage systems_Part 7: Distributed locking problem in the presence of failures, Introduction to design & semantics of highly scalable Distributed Filesystems


Welcome again to the NPTEL course on Storage Systems.

(Refer Slide Time: 00:31)

Summary so far

- Device Level
 - Disk Scheduling
- Protocol Level
 - SCSI, iSCSI (block level)
 - NFS (file level)
- Distr System Level
 - Consistency (commit/consensus protocols)
 - Ordering (virtual synchrony, and at file system level)

®



So, let me first summarize what we have done so far in the previous fifteen classes. I first introduced; what storage systems it typically deal with, and then we started looking slightly into some more detail. In the beginning I looked at the device level for example, a disk, what kind of scheduling is useful (Refer Time: 00:58) without thinking about what goes on up, just purely from at the lowest level.

Then because since most of the systems have multiple components and they have a protocol by which they talk to each other. We have to look at the protocol level and the types of protocols that we looked over SCSI and iSCSI. This SCSI one typically a dielectrical level iSCSI at the network level or internet level, but this was using the block level protocols. Then we move down and looked the file level NFS level. And we notice that NFS has some problems with consistency that is a first time we notice (Refer Time:

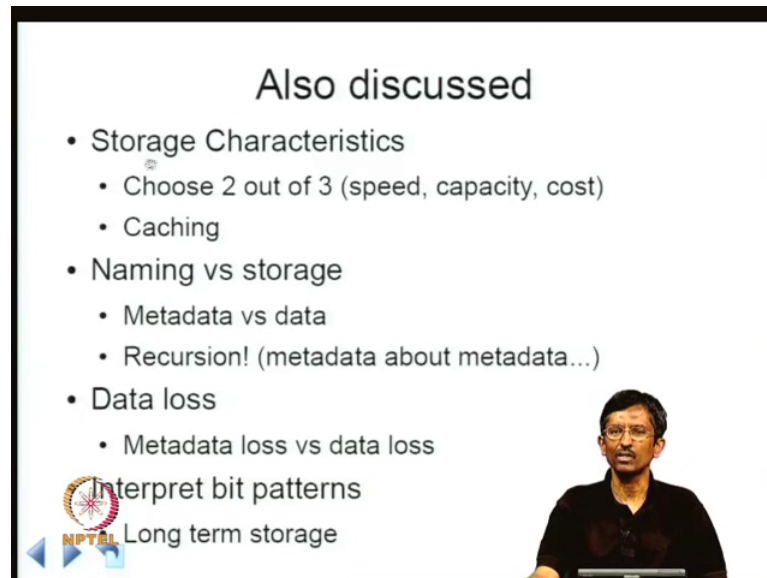
01:43) because we are now coming closer to the applications of understandings.

When you are here at this level, we had no idea about what application wanted as you come up they start thinking about application bit, so that is the reason I again the difference between block and file also is somewhat connected with the application level understanding. Block level is strictly an internal appear of the file system or any other storage system that is below the let us say application slash file level. Whereas NFS already is there is some kind of application understanding, we expect files to have certain information in a particular bit the way I put it in. So, we start thinking about semantics at this point seriously.

And as we go up we are building larger scale systems a distributed NFS also distributed, but it is slightly more simpler model it is basically a server client model. When I am talking about the distributor system, we are talking about not necessarily only a server and a client, we can talk about multiple systems working together to do something that is what I am calling distributor system. And here we have to face important issues that NFS started facing the NFS of issues of consistency. Again NFS because of its weaker model consistency, it went through some iterations a widely used one then became NFS 3 and NFS 4. So, there are two issues that systems at this level try to handle; one is at the level of consistency issues, it could be issues and ordering issues all three things are somehow had to be handled.

We are going to take into account failure systematically later, but we already had to stuck thinking about it that is why in our commit and consensus protocols we talked about failures. And then once we have a distributor system we have to communicate that is why ordering is important. There are multiple types of ordering, at the network level or at even at the application levels, system level, there are understanding things should be ordered. We looked at this also. So, we have all looked at some of the things, but a quite a few more things to look at. So, I will briefly mention what other things we have to also to look at.



(Refer Slide Time: 04:17)



Also discussed

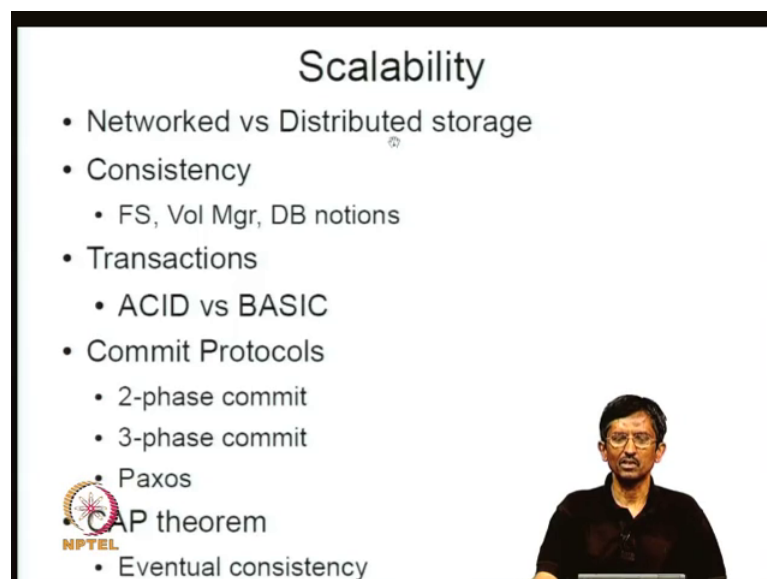
- Storage Characteristics
 - Choose 2 out of 3 (speed, capacity, cost)
 - Caching
- Naming vs storage
 - Metadata vs data
 - Recursion! (metadata about metadata...)
- Data loss
 - Metadata loss vs data loss

Interpret bit patterns
Long term storage



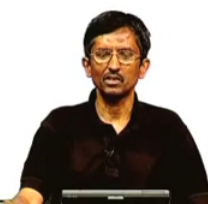

And in passing also I will mention other things we touched upon in the previous sixteen classes. For example, we looked at briefly a storage characteristics, for example why we can only choose 2 out of 4 here, caching naming versus storage difference between metadata and data, there are some recursion issues, whether we value metadata or data, how do we handle those things. This we just briefly touched upon we did not really talk about it, which will talk about in archival storage. So, we as part of what we discussed so far we also touched upon some of the issues except this one in ok.

(Refer Slide Time: 04:54)



Scalability

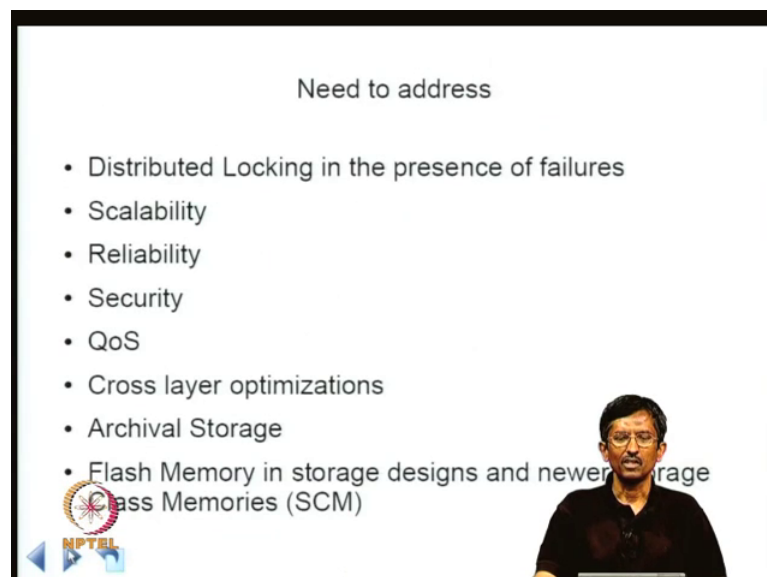
- Networked vs Distributed storage
- Consistency
 - FS, Vol Mgr, DB notions
- Transactions
 - ACID vs BASIC
- Commit Protocols
 - 2-phase commit
 - 3-phase commit
- Paxos
- CAP theorem
 - Eventual consistency



So, what we need to look at also later are issues like scalability; and so far with respect to scalability, we will look at some of these issues. For example, difference between network and distributed storage, network storage a prime example is NFS. And distributed storage is something which is not we used, but you can say for example, GFS 2 is an example of a which (Refer Time: 05:23) had provides that is a example distributed storage or a clustered file systems, which many companies provide like for example, clustered VXFS that is kind of distribution, because they try to give you POSIX kind of storage POSIX kind of model.

Again as part of all we discussed we also looked at notions of consistency as understood by file systems database. I am not really talked about vulnerability consistency we will talk about it later. We also touched upon issues regarding transactions acid versus nonacid kind of models. Again just like a commit protocols we looked at 2-phase commit, 3-phase commit, Paxos. And we also touched upon the cap theorem by and because of this we had to think about slight lash models of consistency eventual consistency.

(Refer Slide Time: 06:26)



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 newer storage Class Memories (SCM)

NPTEL

So, for this is what we have done so far let us see what else we have to look at, there is lot more things to come on this thing. First of all we have to we are not really touched upon distributed locking in the presence of failures, which is an important issue. We just when I building a distributed storage, I have to handle this part; and this have contains

everything in one place issues regarding ordering, in terms of applications understandings, in terms of interactions with file systems applications, all those things failures all those things keep coming here. Again, we touched upon a bit of scalability, but there are newer notions of scalability, which we also have to handle. For example, if I go into scale to web scale systems, will it have even more different modes of scalability that we want to start thinking about.

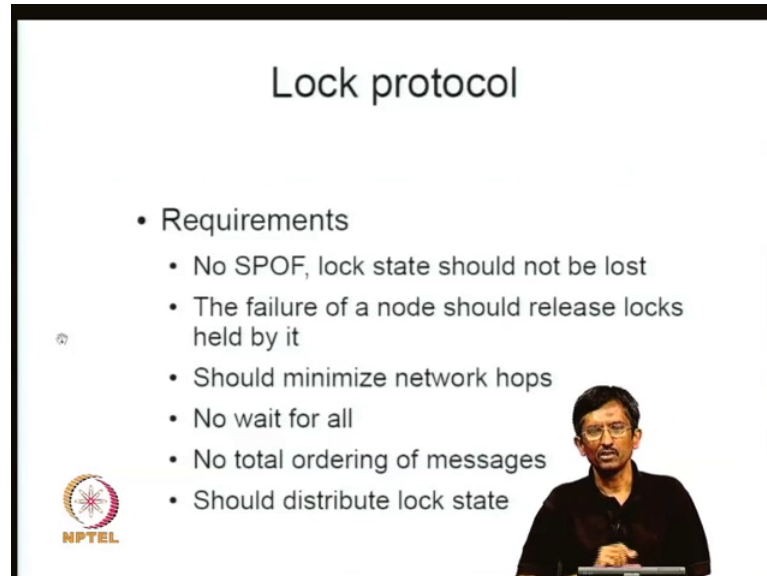
So, again when you build systems of again the web scale reliability becomes very important in that way also we have to look at. Again a good example of something which handles both these things for example, we have discussed again in a very beginning a Google file system does both of this, but it is for a specific application was search only means it is not for other applications. We comparably it is good for read heavy workloads, but not so much for if workloads which involve lot of rights also or if they have small rights, this is usually good only for big objects the Google file system. So, we have to look at this also.

Then also issues on security which are not really touched upon at all it is the big issue, but has. Finally data has to be secured without this the whole thing falls apart. So, appending system give you some motional security, you may need to make it slightly stronger that also is there. Also quality of service we briefly touched upon it with respect to disk scheduling, but not really seriously looking to it, this is a very big issue which also has to be looked into. There are also issues concerning cross layer optimizations, how do we ensure that is it possible that if the more device characteristics, I do my file system in a different way. So, can I optimize across layers, what we do I do about these things that also is a big issue.

And one very big issue is archival storage which is about how do I make sure that if I write something now, somebody can read it 15 years from now. Now, this has to be a salient with respect to device changes, file system changes, operating system changes language changes and very other kind of things which is nontrivial; it is a very, very difficult issue. So, actually there are no serious answer so far. And in connection with that, we also need to look at other storage designs with respect to newer devices, this also we have not looked at. So, it is quite a bit of stuff to go through. So far we have covered a part of it.

So, today I would like to look at distributed locking in the presence of failures then we look at some of this issues which are this in some detail.

(Refer Slide Time: 09:32)



The slide is titled "Lock protocol" and contains a bulleted list of requirements. In the bottom right corner, there is a small video inset of a man with glasses speaking. In the bottom left corner, there is the NPTEL logo.

- Requirements
 - No SPOF, lock state should not be lost
 - The failure of a node should release locks held by it
 - Should minimize network hops
 - No wait for all
 - No total ordering of messages
 - Should distribute lock state

I am going to present one simple lock protocol. Again the real distributed lock managers are very complicated. And if you look at Netware or Sequent some of the companies, they make their name only because they had a good distributor lock manager. And this particular lock protocol is not going to be as complex as those ones actually it is designed by one of my students. So, it is a good simple reasonable protocol which may not take everything into account as the industrial strength distributor lock managers can do. So, we are going to look at this briefly, so that to give you an idea about what kind of issues that cope up.

What are the requirements, it should be no single point of failure, whatever happens the lock state should not be lost, because if the lock state is lost then somebody can try to update something without realizing somebody always got a lock. If you have a failure of a node, you should release locks held by it. Should minimize number of network hops avoid as much as possible. I think we saw some of it in the case of the (Refer Time: 11:03) SFS where they do forwarding of reads and that is why we got a dead lock situation developer right because of that right. So, this was an issue.

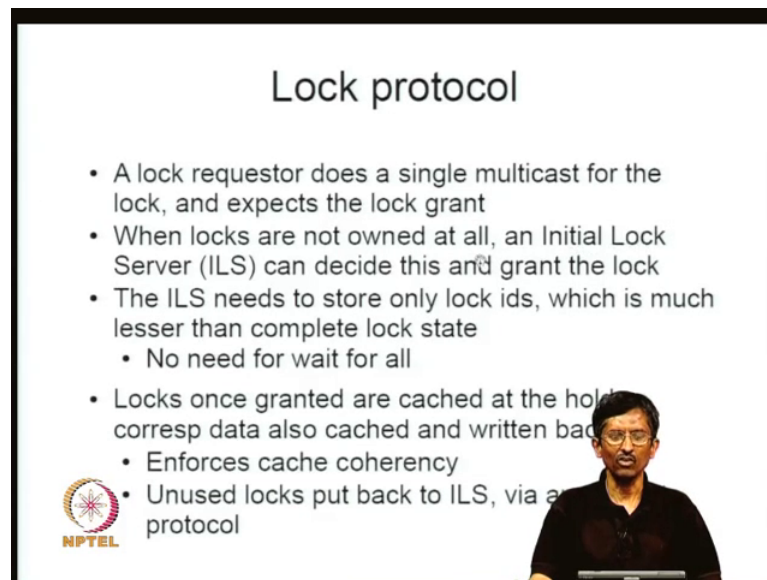
And if you are trying to get if somebody has got the lock state, then you might want to figure out, you may want to revoke it, the question is do you wait for everybody till the

response say I do not have the lock. So, it is preferable that you have a system that is not waiting for the slowest party and the various types of ordering that we have decided we looked at FIFO ordering, casual ordering, total ordering. I do not want total order total ordering is too painful too slow forwards. So, I want something better either it should be FIFO or casual.

So, I wanted my design should handle one of these things rather than the most restrictive form of order. Similar this similar to in the architecture area, we have various memory consistency models and even see that there are (Refer Time: 12:10) sequential consistency then it release consistency etcetera. Typically a sequential consistency is the most restrictive form. If you the easiest for the programmer to write applications and that particular model, but the most restrictive one. So, people of most architecture is do not provided here also we do not want to; however, lock protocol depend on total ordering message. That means, it can only depend either on we will assume that we can write this protocol assuming only either casual ordering or messages or FIFO ordering messages, that means, we are already assuming there is some kind of GCS group committed a system already present.

And the lock state should not be in one single piece it should be distributed of course, it should take a SPOF consideration, it cannot do everywhere. The actual protocols actual details of the implementation or a formal complex and what I am going to discuss is a curious we can look up the thesis written by the student his name is Gowtham.

(Refer Slide Time: 13:14)



The slide is titled "Lock protocol" and contains a list of bullet points. In the bottom left corner, there is a circular logo with a starburst pattern and the text "NPTEL" below it. In the bottom right corner, there is a small inset image of a man with glasses and a dark shirt, likely the presenter.

Lock protocol

- A lock requestor does a single multicast for the lock, and expects the lock grant
- When locks are not owned at all, an Initial Lock Server (ILS) can decide this and grant the lock
- The ILS needs to store only lock ids, which is much lesser than complete lock state
 - No need for wait for all
- Locks once granted are cached at the holder, and the corresp data also cached and written back
 - Enforces cache coherency
- Unused locks put back to ILS, via a protocol

Let us look at the lock protocol. So, what is the issue here, the various clients who are requesting locks they are running on various nodes, and you will find that there are multiple threads on each node any thread can be accessed in a particular lock. So, there is a local agent at each node. And then if the thread makes requests, and it is not available at the local node, the agent will actually make a request basically does a single multicast for the lock and expects the lock, it wants to get it somewhere.

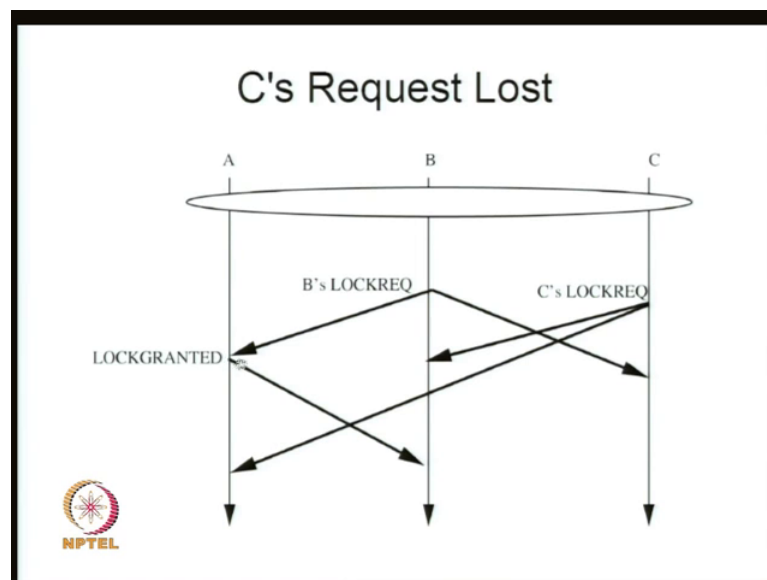
Now, to take care of some bootstrap situations, when locks are not owned at all, you have some kind of a initial lock server. When nobody is there nobody can respond, there is lock is not taken right, there is no one to say in that I have the lock that is why I have a bootstrap situations to have to take care of this (Refer Time: 14:19) ILS which is available which can actually see that nobody has requested this lock. And it can grant it. Of course, the minute we have something initial lock server we have to think about what happens in this ways, but somebody keep waiting for saying that whether the lock is taken or not. So, your group communication system should actually be able to notice any lock fail any server failure and we are able to restart things or come up with a new group that is a new hash leader.

This initial lock server needs only to store lock ids, you do not want to store the complete lock state. So, I will not discuss this further. There is a it turns or this is enough for the time being; it is enough. Locks once granted, but crashed at the holder, but at the node

and corresponding data also crashed and written back, basically you need to have some notions of cache coherency if locks are unused, they are sent back to ILS put you another separate protocol called update protocol. I am not going to discuss some of these aspects that in detail. I want to only concentrate on those aspects which will tell us something about the kinds of issues we discuss about all thing because this distributor lock manger is important component in the storage a distributor file system.

And so I want to look at this especially we are write some reads or you know they are not strude its not only reads and only writes, both have miss of these things. Therefore, there has to be some locking that has to go on.

(Refer Slide Time: 16:15)




So, now let us look at what kinds that can happen. Now, let us say that A has the lock, and B and C are both making request. It turns out B's request comes to A first, and A is sees this request and then grants it to B. C's request is also made, but might say that this C's request happens likes this way. C's request come to B and B's scratches it certain says I do not have a lock, so I just ignores it. Same thing with C, this request to A, A also says I have already given it I do not I cannot do anything with it I will just drop it.

So, what is happen to C's request, C's request is lost? Now, you to do, it is a very simple case trivial case this has to be handled. How do you handle this, you somehow have to buffer the request because the fact that C made a request right somehow has to be buffered by B also. So, there has to be some queue for some time.

(Refer Slide Time: 17:32)

Lock protocol

- Lock holders queue lock requests if they have locally locked the lock
- Requests are granted in FIFO order to prevent starvation
- When a node is in transit, what happens ?
 - The lock could be unowned at prev owner
 - The lock could be unowned at next owner also!
 - Since lock transit can occur only due to requestors, if we queue at requestor problem solved?



So, lock holders queue lock requests if they have locally locked the lock, they have to lock queues, a locks queues a request. So, basically even those guys for example, B also has to queue the request; otherwise, this C's request will be lost. Requests are granted in FIFO order to prevent starvation. Again a summation this is a one particular design the lacks of possibilities here unless showing you one example of what one of my student did as a solution to this particular problem. There could be better ways of doing this. For us the issue for us is when a node is in transit what is the node in transit means basically somebody has a lock and it has been given to somebody else. So, I think it should be on lock is in transit, but this is a mistake, what happens.

The lock could be unowned at the previous owner, the lock could be unowned at the next owner basically because denies that are there can be so substantial that if you go to this guy, this guy already has released (Refer Time: 18:45) state, it has been given to this guy that also has (Refer Time: 18:50) state. So, thing is you cannot really depend on the fact that let us say the current owner or the next owner also will have it, it could be any where we do not know, so that something which has to be figured out. So, as I mentioned before one solution is if we can queue the request even at sides like B. So, what we know that C's request cannot be dropped that is a reason why you do is, but this itself causes some additional problems.


Basically, you can see what is happening is at the similar distributed state, it has to be

properly taken care of, and the messages are coming in the arbitrary order depending on the way the things are getting delayed etcetera so that is where do we have a clear correct snap shot is nontrivial.

(Refer Slide Time: 19:42)

Lock protocol

- The lock requests queued at the requestors may not all be valid..
- With the previous example, one of the nodes (B or C's) lock request will be obsolete
- We flush obsolete requests by having a logical time on each node.
 - Every request that is sent is stamped with the current time
 - Locks carry a timestamp also which says when this lock was last held by a particular node



So, it turns out that some request can be obsolete, you can look at it. You can queue the request, but some request can be obsolete, let us look at this one. See for example, B requested it, A gave it to B, and then what happened was C requested it. And B in the beginning also had requested to C. If I queue all the requests then when C gets it that after B has got it, it will still have B's request still sit here you got notice that also its very obvious thing, but you have to handle it. Because notice what is happened at the very beginning B decided it wanted a lot therefore, it send it to everybody. So, C has queued it because for this particular reason because I did not want C's request to be that is why I did that.

Now, but what has happened is that once B has got it and this you to see some time later an obsolete request of B is sitting (Refer Time: 20:55) here that also has to be handled. I must getting some of the kinds of things that really happened in a proper if you are really looking for consistency in a LAN or a let us say distributor storage kind of system this message can get lot of arbitrary ways and we have to be correct with respect to all these orderings. So, we can see that one of the lock request obsolete. How do you know that lock requests obsolete? You can see that I made a request here, this is if I am using

because if I there is some way of finding out that this B's request to C is obsolete come because I have already got the lock that I can drop this.

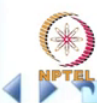
Then when C gets a lock, it uses some technique where it figures out that this request is obsolete because B also already granted a lock. Wherever to drop that one then everything is fine, but this requires you to have a notion of distributor time across all these nodes that is why we need a vector time clock. Again, we discussed it in the previous class, we need a vector time clock model is. Then I can figure out that this request which is sitting here is obsolete because B already had a got the lock it is done and C when it is C is this particular request. And then it when it gets a lock if there is a way of time stamping that lock when it got here it can see this obsolete and throw it out that is how we have to do.

So, what we do is we obsolete we flush obsolete request by having a logical time on each node. Every request that is sent is stamped with the current time. Locks carry a time stamp also which says when this lock was last held by a particular node for example, when B got A there will be a time stamp on it, and then you can see whether C can decide that B all has already got a lock, so I can flush this obsolete request.

(Refer Slide Time: 23:16)

Lock protocol

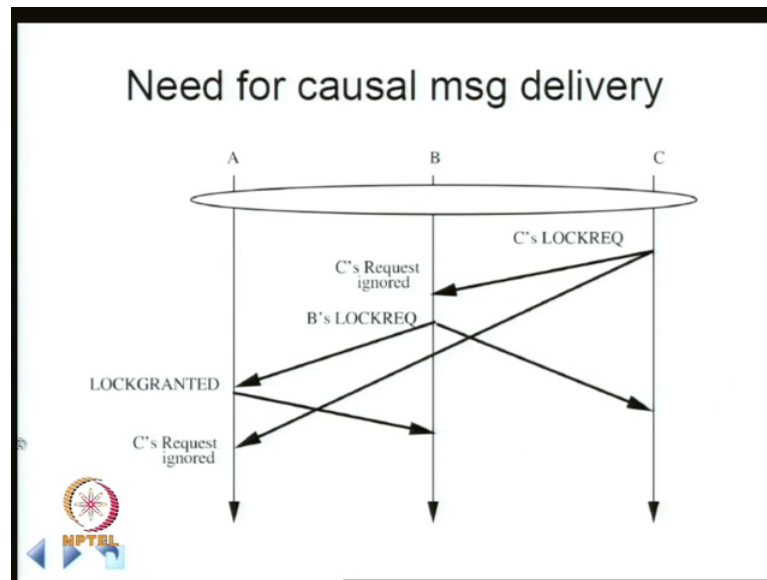
- We increment the logical time on the node and stamp this on the lock each time we grant a lock
- If the timestamp on the request is $<$ than the time on the lock, the request is obsolete
- Lock requests with \geq timestamp are valid
- New node's timestamp on the locks are set to 0 at all nodes



So, basically the protocol, we increment the logical time on the node and stamp this on the lock each time we grant a lock, so that it kick like all these events. The time stamp on the request is less than the time on that lock the request is obsolete lock requests greater

than equal to time stamp are valid, and the initialization time stamp also have to be time taken care off. So, this is one example for (Refer Time: 23:48) this is not enough, there is some more need for.

(Refer Slide Time: 23:50)



So, right now so far we can do FIFO we said, but there has also reasons for doing causal message delivery. Now, let us see what this causal message delivery is about. Again we are assuming a certain type of message ordering that happens. So, C makes a lock request and B also makes a lock request, but by shade bad luck C's request comes to B, before B has made it, but C's request to C to A sorry to A comes after A has granted it. Now, again what will happen C's request to B, B does not have a lock respond, therefore, B will drop it let us say B drops it, or suppose a drops it also because this is a (Refer Time: 24:56) then we have a problem also. If you even if you use a previous solution of buffering there is some work. Actually what you have to notice is that you had noticed that C's request can before B's request and therefore, there is a need for a causal ordering.

We discussed causal ordering and that is what has to be implied here. You have to take care of this particular situation where C's request are essentially not being handled properly. You notice that if you do not use causal ordering there is no the FIFO ordering does not help us because C this C right is starting to these two guys, and B starting to these two guys, and there is no way to order them in a FIFO order. There is no they are

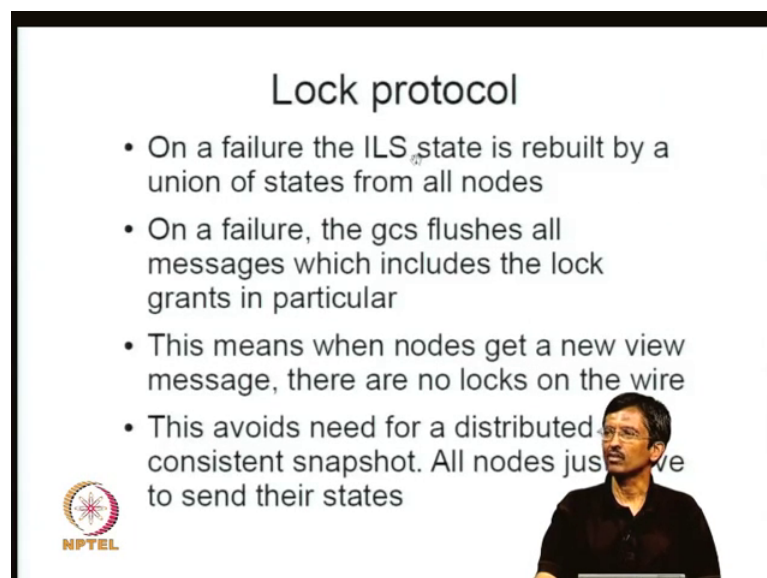
incomparable, but from a causal point of view it is possible. So, this particular design takes into account that you can order messages either in the FIFO way or in the causal way, these are the only thing assumptions, we are not assuming any total order.

Student: (Refer Time: 26:15).

Of course, to be completely sure by this particular protocol works, you need to formally model it and check everything, and this was not done. So, there could be a bug in this protocol system, I do not guarantee it is to completely correct protocol. And there are other issues also which are not taken into account because it turns into real systems there are issues like interrupts, system calls etcetera are there. And they also may actually if you get an interrupt it may not be deliver at right time, it may be delivered only after the system call exits; so various reordering or techniques at different levels in system.


So, a real complete solution has to take into account all these things. In addition to at this level there are levels below where somebody else had could also be doing reordering, unless you are clear about it or unless you take some measures to make sure this kind of miss ordering take place. We cannot really guarantee correct this solution, so that hopefully gives you some idea about why we need causal ordering.

(Refer Slide Time: 27:19)



Lock protocol

- On a failure the ILS state is rebuilt by a union of states from all nodes
- On a failure, the gcs flushes all messages which includes the lock grants in particular
- This means when nodes get a new view message, there are no locks on the wire
- This avoids need for a distributed consistent snapshot. All nodes just have to send their states

 NPTEL

So, if you get causal ordering you can very clearly see that C requested before B and therefore, A can try to attempt to give it to C rather than B. So, there are some things we


can try, if you have a causal order. Somehow there is a failure let us say the ILS itself; we have to rebuild the states, union states from all the nodes. So, the other things that you have to do for example, you have to flush all the messages which includes lock grants. So, basic idea is that you want to get it to a clean state, and therefore, what we are trying to do is that no lock request should be on the wire at that time.

And that is basic idea you want to get it to a particular state where all essentially the state in a known simple state. Otherwise, you need to do something more complicated that is called distributed consistency snapshot, so which makes it even more difficult to engineer.

(Refer Slide Time: 28:43)

Reader Writer Locks

- Uses the same underlying scheme as the non reader design
- Need a policy for readers and writers
 - Readers read till a writer comes in
 - A writer granted next
 - Then all waiting readers so far and
- To perform the lock protocol activities, a primary reader is elected among the readers.
- This is elected by a writer or by the ILS



So, now with respect to you typically also need in addition to this something called reader writer locks. Now, when you have multiple readers and multiple writers, you need to have some policies; and you can follow the some of the kinds of policies that are used in regular mutual exclusion algorithms. Readers read keep reading it till a writes comes in. Then what you can do next is you can prevent any other readers from coming in, you give the access to the writer drain out all the readers. Then the writer gets it, and then all the other readers who came in between there will be queued. And then when the writer finishes, if there are no other writer then the queued read readers come in. If the other writers in between then you have to again decide, whether the queued readers or the next writer who has to get the access to that those kind of things can all be written, that is not


those kind of issues are slightly easier to handle in that policy decisions is can be handled.

Now, it also turns out that to perform some of the protocol activities, a primary reader also has to be elected among the readers, because somebody has to keep track of who are the readers, who are the writers etcetera. Because some distributed state has to be kept about how many readers are there how many writers are there etcetera. Again this particular primary reader can (Refer Time: 30:08), you have to handle that part of it also. So, there has to be some election that can happen multiple times. So once you have this, a primary reader and ILS probably they can watch for each other there has to be some redundancy in some of these things.

(Refer Slide Time: 30:30)

Reader Writer Locks

- The code is substantially more complicated
 - The need to handle multiple readers, waiting for readers to drain out
- Recovery is more complicated
 - Reader failure has to be detected by writers or by the ILS
 - If a primary dies, all non-primaries will have to perform duties of primary till ILS elects a primary.

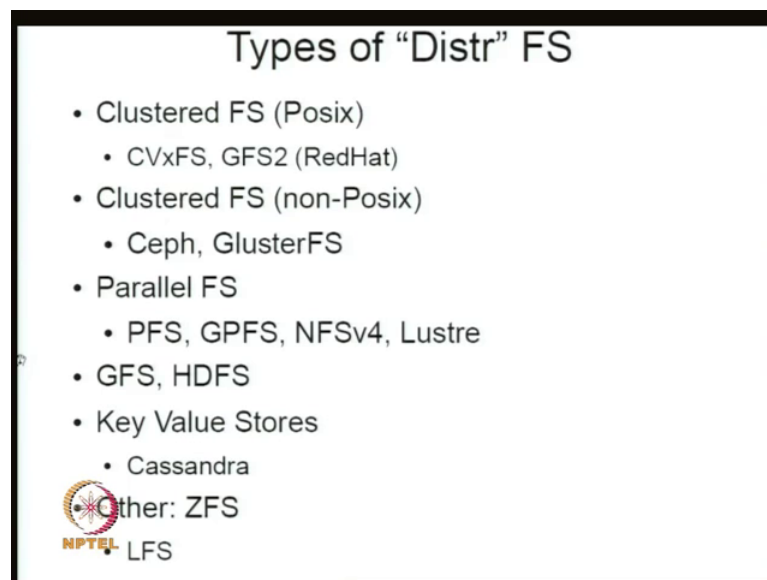


So, again a summation the code is substantially more complicated for this case you need to handle multiple readers waiting for readers to exit. Recovery is more complex reader failure has to be detected by writers or by the ILS, primary dies all non primaries have to perform duties of primary till ILS elects. Now, there is a issue of this kind. So, above I have given some idea about some issues a distributed lock managers has (Refer Time: 31:08) and this is just a simple design. And idea here in this particular design was to use group communication systems primitives like a causal ordering and FIFO orderings to a exclusion of total ordering that has what this particular attempt was. And should be a student was able to come up with a consistent working system not that it has been proved

to be correct, but it works sufficiently that very large programs could be run.

Now, so after this discussion about the distribution lock manager, I would like to move on to few other topics, I am not going to discuss distributor lock manager after this. Probably we need talk about or some point later possibly another lock manager for example, Google's chubby lock manager, information want to look at we will see now depending on the availability of time we will look in to it.

(Refer Slide Time: 32:17)



So, now, again, let us just do a bit of summarization of some other things that we would like to look into, and where we are. So, let us just briefly look at the various types of distributed file systems that are available and we just take a look at some of these things. First of all we are right now we will exclude network file systems NFS, because there the notions strictly are client and server multiple clients, but a single server. What we are thinking about when I say clustered is that there are essentially just know is a single server there are multiple nodes, which together behave like a single server.

So, you can have a what is called a clustered file system and you can either provide a POSIX semantics or a non POSIX semantics. Now, this issue of POSIX semantics, non-POSIX is a very delicate one various parties claim to have POSIX semantics, but turns out they in some specific cases they do not provide it. So, it is very difficult to figure out who is really POSIX who is not really POSIX I have given my idea what POSIX is wherever I understand of POSIX. What they are doing (Refer Time: 33:32). For example,

cluster FS in that documentation say that they are POSIX, fully POSIX from plant, but I am from what I understand there are certain issues they do not handle which is really POSIX related things. So, we do not really know, it depends on the interpretation, but there are some clustered file systems which are definitely POSIX, for example, CVXFS the clustered VXFS is a clustered POSIX, they also have to play around with POSIX a bit to get efficiency.

And for example, one of the kind that things they do is a following. In a regular file system, what is called there are anytime a file is accessed we have what is called A times C time and M times. A time is access time, C time is create time, and M is modified time. Now, if you are in a distributed system and if you are making access is (Refer Time: 34:44) access is then every time you would do access time, then If it is a case that everybody else node should know about the factory accessed it. It is extreme. Basically what it means is that I am the only party who is using it and if there is somebody else who has got it cache, but it is not really using it, he also has to be informed that access time has changed. Now, that is a bit too much, because that basically means that all caching and everything have (Refer Time: 35:14) useless.

Now, basically I have to keep on synchronizing meta data across the whole system for every access I do every single access because every time I access something the access time will change even if only for your repossesses. Now, that is intolerable. So, the semantics of access time update has been relaxed. So, they are they said they are POSIX, but if you think about it carefully even the clustered VXFS cannot be strictly hundred percent posses complaint. So, that is the reason why there is some dispute about how you classify things I have just given you my idea what it is, but this I think I am pretty sure that GlusterFS they will say that they are process complaint and GFS 2 the red hat provides something called GFS 2 which also claims to be process complaint.

The other file systems which avoid certain cases which slow down the system for example, there is a system called Ceph which tries to reduce the time taken for look ups and there they may not be process complaint. For example, normally when you look up a name, the name has got multiple components, so you cannot do a single look up, you have to do multiple iterations to get each of the sub pathenings right, each of them has to be passed independently and finally, you get a full path. If you want to do you know one shot look up then it is not possible. And why is that issue because it is possible that if you

do in one shot, it does not have a same semantics if you have to do it with part by part because it is possible that some other concurrent activities happening, and therefore, if I do it in one shot the result will be one. If I do it part by part it will be some other semantics is possible, so that is the reason why Ceph I would call it non-POSIX.

So, other kinds of file systems like parallel file systems GPFS is the well known parallel file system for an IBM, NFSv4 also has some notions of a providing parallel access to files and Lustre also provides certain notions. So, basically the idea here is that we have very large file, and it is sub divided into small portions and each node will actually work on that particular portion. So, it is sort of implicit partition. You do not take any locks etcetera, it is clear that only you distribute the file across these nodes the application has figured out some non implying ways in some things are the file is chopped into pieces not really chopped into pieces, it is accessed or updated in parallel through a implicit understanding at the file system level or the application level, that is what is going on here.

If it is done through the application of course, then the file system not there in the feature if it is done at the file system level, then the application has to tell the file system that please I want a particular view of this particular file that has to be in 10 let us say parts and I am going to assign various nodes to work on each of these 10 parts independently. So, the various systems of this kind have some notions of this kind again there are other kinds of file systems as you discussed before things like Google file system and Hadoop file systems.

And as you go down this side we come closure to this notions of key value stores, which are basically quite removed from POSIX, they do not have the same semantics. Again we discussed in the very beginning in the first two classes something about key value stores briefly, we looked at some of the APIs. And we want to look we will study this in some detail, we look at this areas a bit. We will find in the other types of file systems which are not really touched upon, which are also important. For example, you can have a file system like ZFS originally from sun and they actually provide very interesting guarantees with respect to consistency the file system in the ability to incorporate the device management as part of the file system itself. So, these are fairly well thought out idea about integrating device management and file management, and it gives you reasonably good reliability guarantees.

So, similar to ZFS, in Linux, we have something called beta RFS, I forgot to include it here there is something called BTRFS, which has got similar guarantees like ZFS. Basically these are designed taking to account reliability as an important aspect. So, one thing they try to do is if you write something, you can either have what is called end-to-end arguments at the file system level or application level. So, these are ZFS or BTRFS you are getting that end-to-end argument with the file system level. Of course, application also has to worry about that because there could be some problems from the application to the file system, there could be some corruption of something exclusively possible.

But if you do it to the file system level already you are now guaranteeing that whatever happens, whether there could be some malfunctioning host bus adapters or great devices or disks whatever it is, you have some kind of check sum. And then when you then you read something, you check whether the what you read as the same check sum it should be present if it is not present you can actually flag in error as quickly as possible that is what ZFS does that is what BTRFS does. So, this one ZFS basically currently is not a distributor file system as we have looked at this ones, but it is distributing in terms of devices it can handle larger in the devices and. So, it is distribution at the device level and not at the file system level. So, it does not have it is not clustered as the way we are talking about clustering here.

There are also other designs like lock structure file systems, which are I mentioned earlier that you can do some cross layer optimizations lock say the file system. As example of a an attempted cross layer optimization, because you take into account the characteristic about disk carefully. If you look at a disk, you will find that disk is very good at giving you good bandwidth, but it is poor at latency. So, if you are designing a system which has got reasonably good bandwidth that read or write bandwidth, but it is poor in terms of latency, then one of the design options is to make sure that you aggregate request as much as possible or you prevent you make sure that you do not have to do any seeks. You avoid seeks as much as possible.

The lock system file system basically has interesting design which basically says that all my blocks etcetera are essentially a lock. In a normal file system, you will find that the blocks can be anywhere on the disk, and you will have to seek to a particular block to be able to write read or write. So, latency is involved in it because you have to seek to it that


seeking basically is a mechanical activity and it takes about 10s of milliseconds.

Whereas in a lock stage or kind of file system what you are planning to do is to ensure that seeks do not take place. So, the head does not move that much idea is to keep on writing without moving the head if possible only under recovery situations that is wherever some error has taken place I use a situation at that time you go back and read; that means, there you do seeks only and failure events serious failure events. So, this is also another design options. So, various designs have been attempted with respect to distribution with respect to device management with respect to trying to do cross layer optimization etcetera, and there are quite a few more if you take into account things like security, faultier service all kind of things. So, we are not really gone it those things.

(Refer Slide Time: 44:29)

NoSQL

- Cloud computing has shown that current RDBMs cannot scale and do not have the reqd perf
 - I have not able to find an example of a large-scale Web application that has been able to meet its needs with a single coherent RDBM system (John Ousterhout)
- Column stores
- “NoSQL” systems: Avoid ACID
 - Amazon, Facebook, Google, Yahoo, and Ebay
 - Bigtable: a sparse, distr multi-dimensional sorted map
 - Apache Cassandra — Facebook’s dist storage system based on Bigtable data model on Amazon’s Dynamo-like structure.



So, what we like to do is continue on what we have discussed earlier. Basically to try to see how if you are trying to scale things up right what kind of options are not possible. So, what we will now do is instead of being let us say looking at POSIX as the model, we will decide POSIX is not that important, let us say that. For scalability I am willing to sacrifice POSIX compatibility that means, it has a specialized applications which do not require process that whoever is reading the application knows that knows what they had in hand that whatever system they are using does not guarantee process kind of semantics it is a special command semantics which you use. So, this is where you will find lot of work happening now.

And this also has been interesting because the issues with respect to whether for this last (Refer Time: 45:33) closed systems whether you should use a file system or a database has also come into pictures in a major role that is why there is a this slogan called a NoSQL. It is basically saying that because I this NoSQL itself I think is a problematic name, but this basically is trying to say that the current relational database models cannot scale. And the things that can really scale or slightly closer to the file system kind of models that is a even my Google file system - the GFS does not really address POSIX which not really part of that design. They are not gone for a database because database will be not scalable.

And that is why for example, if you look at the quarter one John Ousterhout is a well known operating system designer, he was also the author of this LFS this particular lock system file system. He makes the claim like that he has not been able to find single example where RDBMs have actually scaled a good for consistency if it is critical information like monitor aspects have to taken into account. But for the web scale applications where that is that is not the overriding concern this RDBM is have not been able to scale it out. Many companies which started with RDBMs have finally, had to move away from it to a different model I think if I remember right Facebook started with a where RDBMs and then has really became big they have to essentially moved a system away from this data base model.

And if you look at it Google file systems sorry there are something called column stores which are slightly different model compared to relational data base models, because in relational data bases you have the notion of rows. Here you basically keep information in terms of a column and they are some positive things that come because of this instead of having a row orientation basically information that you keep in the row for example, in a column for example, is usually the same type of information and therefore, compression is better etcetera. So, when you are doing very large-scale systems this column stores has been also explored. So, basically this NoSQL kind of model basically says try to avoid the highly restrictive acid model, it may be good for some critical applications, but it may not be applicable in main large in many other situations.

So, the idea was to not go for acid just like that, just carefully think about it you have need to go for it otherwise drop it go for something else. It is its more scalable that is how you will find that most of this big companies they have not real gone for they have

certainly some small scale relational databases, but mostly they have gone for non relational models or disk storage needs. I think Ebay's only the one I think here who is heavily invested still on relational model, because they most of their thing is based on that is auctioning kind of systems where lot of monetary aspects are involved, I think these are one of the things for more heavily into relational database.

And the way they do scaling is by having per let us say what you might call private databases. For example, if there is an auto they will have a database only for autos. If it is a some electronic components like PCs, they will have only a complete database only with PCs. So, they distributed that way. Again finally, turns out to be a single data base not a distributed database because the real problem is that scalability is not there with respect to distribution.

So, the other kinds of systems that people have come up with is a example Google has come something called big table and it is a sparse distributed multi-dimensional sorted map. And we will look at some of these things later some details about this. Similarly, Facebook has come up with a some model Cassandra which is become a apache project it is a distributed storage system based on the BigTable model and. So, we will also look at some of these issues. And Hadoop is similar to a Google file system again open source kind of model. So, what we will do is we will make our way through some of these systems in model.

(Refer Slide Time: 50:33)

Transactional Workarounds for CAP

- "BASE": No ACID, use a single version of DB, reconcile later
- Defer xact commit until partitions fixed & distr xact can run
- Eventual consistency (e.g., Amazon Dynamo)
 - Eventually, all copies of an object converge
- Restrict transactions (e.g., Sharded MySQL)
 - 1-node xacts: Objects in xact are on the same node
 - 1-object xacts: xact can only read/write 1 object
- Timeline consistency: Object timelines (PNUTS/Yahoo)
 - Reads are served using a local copy; may be stale
 - But application can get current version or any vers>N
 - While copies may lag master record, every copy goes through same sequence of changes
- Test-and-set writes facilitate per-record transactions

So, let me just mention some aspects we can look into as part of scalability. Again for most of these things we need to worry about how to guarantee three things that is consistency and availability and called as two partitions. So, I think we already mentioned it first option is to say no acid, use a single version of database. Again this is been followed by as I mentioned EBay. You scale across products not you have sorry you have each product has its own database and that is not distributed. Or other thing it is to have multiple databases and then you reconcile them somehow, manually reconcile them or use some semi automated way by which the things go out of they become inconsistent you do something about it to figure out how to make it consistent.

Other thing is if there are any partitions that happens in the system, you differ the transactional commit until partition are fixed, and in the distributed transaction come down that is one way to do it, but this is basically availability sacrifice state or you can do eventual consistency model I can be discussed briefly this one. So, eventually all copies of an object coverage which is probably weaker than what it might be happy in many situations in some situations, but this is one that is available. Again when you design big systems of that kind you figure out where you can tolerate this and have it on that way where you cannot tolerate you go for acid.

So, for example, Amazon we will do most of the stuff using this model, but when it comes to billing and other kind system they will go use acid model, they are finally, the important aspect like money are been transacted then you probably use acid and you a regular relational database. So, basically you have to this figure out an application where some things are what kind of consistency is appropriate and then do something like that. You can also restrict the transactions and this is what is called Sharded MySQL that is Sharded means we have break it into few pieces and Sharded is basically a piece and then each pieces handle then and put it the place. I think just like I mentioned about EBay one node transactions objects in transactions are on the same node that is do not go across multiple nodes. So, all these problems are scattered across (Refer Time: 53:29).

So, other thing is one of the transactions transaction can only read or write one object. This could be across multiple systems, but there is only one of the transactions that mean, that you are not concerned about the causal connections between multiple objects. We do it with respect to one object. Again this similar in complete architecture world we notice that there is some notions of memory consistency models. We have cache

consistency models and memory consistency models. Cache consistency model takes care of only things in particular cache line, whereas, memory consistency models talk about relationship between multiple cache lines, there is some causal connection.

For example, if I do one thing followed something else I need to make sure that just like what we discussed earlier you might have to be made stable in particular order. Similarly architecture also have the similar issues. And there are other issues which for example, other concise models I think I will not discuss this in detail right now, for example, Yahoo has got some other models what they call timeline consistency. What they do is they keep various versions of the file and your application can request a particular version.

So, the ways to figure out which version current version is there we can say that I cannot tolerate anything over than a particular version. And they guarantee certain things about how applicants this versions change. While copies in a lag master record every copy goes through the same sequence of changes, so this kind of models also possible. So, this is a I am not going a too much in detail, but basically there other models also possible. So, this is available in a system called PNUTS which where these kinds of models are (Refer Time: 55:37).

I think in next class, we will start looking at some issues regarding scalability and we start thinking about looks like some systems like Hadoop file systems or Google file systems.