

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

Theoretical Foundations

Lecture – 36

**Theoretical foundations of Distributed Storage systems_Part 5: Basic concepts of
Message Ordering, Ordering Models for Storage Systems**

(Refer Slide Time: 00:50)

Ordering semantics

- none
- FIFO: if a process casts m before m' , no correct process delivers m' before m
- causal: if cast of m precedes m' , no correct process delivers m' before m
 - e precedes f (Lamport) iff
 - a process executes both e and f in that order, or
 - e is the cast of some msg m and f is the delivery of m at some process, or
 - there is an event h such that e precedes h and h precedes f
- total: if at a correct process p , m delivered before m' then m will be delivered before m' at all destinations they have in common

Welcome again to this NPTEL course on storage systems. In the previous class, we were looking at aspects relating to virtual synchrony and we are trying to figure out how to order failure events with respect to other events in the system. Now, I briefly started looking at ordering semantics with respect to messages last time. So, basically there are issues regarding how to order failure events with respect to messages and across message themselves they are two aspects. So, we covered the first part of it in the virtual synchrony model now we will take a look at the ordering of messages among themselves when they going to be delivered. So, I think I have went through a few of them last time and then other no ordering semantics at all or there could be FIFO semantics in which the process either broadcast or multicast can before m prime no correct process delivers m prime before m .

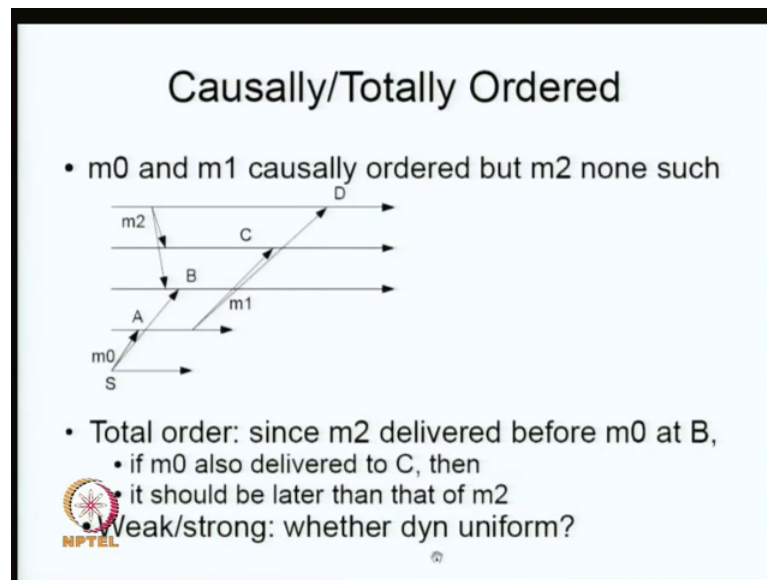
So, basically it is clear that processes is doing both of them m and m prime. Therefore, in every process that survives it should not be the case at m prime goes is delivered before

m. And this will take care of obvious problems that we have seen before I think we mentioned something about in a email system, you will find that a responds to a message comes earlier than the message itself that is kind of stuff you want to avoid. Or in the case of the crickets scores, you essentially guarantee that if it is FIFO you have more or less saying that the scores increase monotonically rather than go upon come down, so this things were avoided with FIFO.

What about for causal if broadcast or a multicast at m precedes m prime no correct process delivers m prime before m. Now, the solution of precedence I think last time we mentioned that there is a way to order events in a distribution system that is due to Lamport. And basically it is a event through it a bit last time basically all its says is that if for example, a process executes e and f in that order then e is before f than e proceeds f either that. Or we are talking about messages in a system, so if e is the broadcast or multicast of some message m and we look at the delivery of m at some other process then we can say that e is before f, e proceeds m because e has been centrally are therefore, it can be delivered. An obvious reason why e has to be before f or there is a transitive closure where we are saying that e is before h and h is before f, therefore, you can say e is before f.

So, basically if it is causal, in case if cast of m precedes m prime then you want to make sure that no correct process delivers m prime before m with respect to total basically here what we are trying to do is we want to order all the processes somehow. So, what choice is if at a correct process p m delivered before m prime when m will be delivered before m prime at all destinations they have in common. So, if m is delivered before m prime at a process p, then it should be at all destinations m should be delivered before m prime these are total ordering it is a very strict and for a sometimes unnecessary illustration, but this is one model.

(Refer Slide Time: 05:04)



So, let us take look at an example. You see this is some messages m 0, there is some message m 1, there is some message m 2. Now, m 0 does a multicast A and B. And you notice that the message m 0 comes to A at this point in time; and then B in turn does a multicast to C and D. Now, in some sense you can say that this event I seen from A it is later than this one. Now, the sync is this may be causally related to A or may not be they can this could be something and come this completely independent, but our distribution system not having this applications semantics or higher level knowledge as long as finding out whether this is actually independent of this. All we can say is that as a process landing at A, we know that this happens later than A.

So, a safe assumption is that this is somehow connected with this, it may not be true in principle in all cases, but it is a safe assumption it is a conservative assumption. So, what you want to ensure is that if you look at m 1, you want ensure that that deliveries of C and D should be causally order with respect to deliveries of A and B. Example here you notice that a followed by this therefore, C should be definitely later than A and D should be definitely later than A. If there is a system it does not guarantee this then they are not causally related. There is something else m 2 which has no connection with m 0 because notice that the only way I can say that m 0 is may be causally related to m 1 sorry m 1 is causally related to m 0 is because there is one common place A where I can see that this one is definitely earlier than this. If there is no such thing then I cannot really figure out what is going on.



For example, this m_2 goes to C and B, and there is no way in which I can say that for example, here I see this is before this here I can say that I cannot really order this with respect to what is happening at A. And C is the event happened here for sending it out I cannot really put m_2 in correspondence with respect to other itself before or after there is m_0 . Now, if I am thinking about total order you will look at B, suppose my timeline is like this m_2 , suppose has been delivered at B before m_0 at B. Suppose I extend this particular message multicast to C also, then if m_0 also deliver to C, then should later than m_2 (Refer Time: 08:35) because in this case it turns out that m_2 was delivered before m_0 . Here you can say that m_2 has come before m_0 at the therefore, if m_0 happens to be also it is a multicast it to C it should be later than that of m_2 because m_2 has come is before m_0 and B. So, in the sense it has to be a global ordering.

Like in all this cases also these ordering can also be let us say divide in two types whether it is weak or strong and it corresponds to whether things are dynamically uniform or not. As I mentioned before in dynamic uniform, you look at those processes which may be involved in it, but they die in between. So, we are saying that we want to guarantee those things with respect to those process also, then it is dynamically uniform. If it only talking about guaranteeing this with respect to only surviving process is then that will be a weaker model or a non dynamically uniform model.

(Refer Slide Time: 10:10)

Logical Clocks

- Associate a counter with each process and msg in system
 - LT_p : logical time for process p (p 's counter)
 - LT_m : logical time of message m ("logical timestamp of m ")
- Maintaining counters for a new event:
 - if $LT_p < LT_m$, process p sets $LT_p = LT_m + 1$
 - if $LT_p \geq LT_m$, process p sets $LT_p = LT_p + 1$
 - for other events, process p sets $LT_p = LT_p + 1$
- For a more accurate clock with static number of processes, vector clock

Now, when we talk about this precedes relationship, there is a way to order events in the system and a semantics Lamport is defined it. And the formal notation comes from the notion of what is called logical clocks. And basically what we do is we associate a counter with each process and message in system. Let us say that at process p there is a logical time p it is a counter correspond to that. And $L T m$ is the logical time of message m suppose. There is a new event, what can do the new event it could be because the process p got a message or it doing something else. Suppose, it got a message then what you do is you look at the message and see its timestamp logical timestamp.

So, because it just received the message and if it is so turns out my current notion of time is smaller than the notion of the time as seen by the message that are just received. Then I tell myself anything that I do in the future has to be later than the message that I got therefore, that has to be my current notion of current time is going to be the most recent event that of this message plus 1 that is what is basically I am updating my notion of clock. Similarly, it so happens that I have done many more other things after a particular message has been received I just have to increment my own time plot by one. Here I put greater than equal to it is just technical thing because simultaneity in a distributed system is essentially impossible, so but in case it happens you can still do this.

So, if it is not because of the message is because any other event I can also increment my clock. So, this is how I keep track of the time events in that system and every event has got a counter sorry every event has a number which basically is a current logical time; and then in every new event, I have to do something about increment in that clock. So, it is basically logical clock. Now, this kind of system works well if you have that is a number of process is that keep on changing with good time is dynamic, the number of process is in system is dynamic. So, it is a very general model, but it turns out that there is a more accurate clock as possible with static number of processes is called a vector clock.

And if you are looking at something like causal models, this vector clocks are quite useful that is say for example, many implementations of causal ordering use vector clocks, because by definition in a group communication system, you are going to be managing membership. Since you are managing membership you exactly know how many processes are there during for that epoc before the view changes. Therefore, we can assume that between failures events definitely there has to be fixed number of

processes. Therefore, you can use a more accurate clock the vector clock when we look at the vector clock.

(Refer Slide Time: 14:00)

Vector Clocks


- A vector clock is a vector of counters, one per process in set
- Vector time for an event e : for each process in vector, how many events occurred at that process causally prior to when e occurred
 - if VT of msg m is $[4,5,6]$, 4 events happened causally prior to sending of m at process p , 5 at q and 6 at r
- VT_p and VT_m : vector times associated with process p and msg m

Given a vector time VT, $VT[p]$: entry in vector corresp to process p

- a count of the number of events that have occurred at p

Updating vector clock:

- Prior to performing any event, process p sets $VT_p[p] = VT_p[p] + 1$
- Upon delivering a msg m , process p sets $VT_p = \max(VT_p, VT_m)$



What is a vector clock is a vector of counters one per process in set. Basically if there is an event e , what are vector time corresponded event, what it is basically is for each process in vector it essentially tells me how many events occur in that process causally prior to e when e occurred. For example, suppose there is a message m and I say it is 4 comma 5 comma 6. What is it mean; that means, that there is the message m is let us say has that it has been sent and basically an event has been received at this process p that means, that four events happened causally prior to sending of m a process p , 5 at q and 6 at r . Essentially we have a fairly a detailed information about what all happened at each place each of the nodes p , q , r .

But we look at logical one it is all gets put into one counter because we do not have that discrimination. So, I can essentially make final distinctions with a vector clock then I can do with a logical clock. So, for example, I can know for example, that my for example, the number of events causally preceding some event at my place has not changed where as if in a logical clock it will be a different number because other events and I cannot configure it out, whereas here I can figure it out. So, suppose I has a notation VT_p and VT_m these are the vector times associated with process p and m with process p and message m .

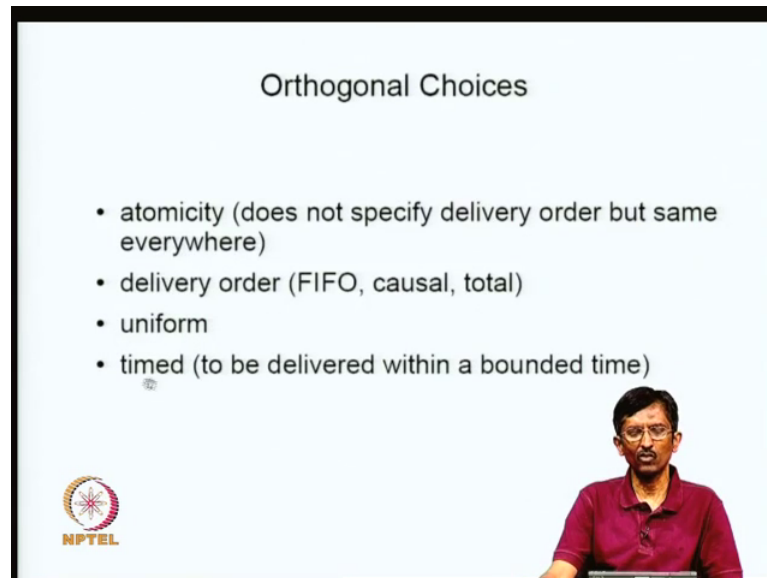
I also have another notation $V T$ of p as I mentioned this one per process in set $V T$ of p means it is a entry in vector corresponding to process p . So, this bracket thing issue the entry corresponding process p whereas, this events are different vector clocks vector time sorry these are different two vector times. So, each of them might will have some vector corresponding to that fixed set of processes.

Now, how do I update a vector clock? Suppose, I am at process p , my vector time is this $V T p$, I am going to do some event right now. I want to label that particular event with some vector time. What do I do, because I am at process p I just increment the corresponding entry for process p , I say $V T$ of p equal to $V T$ of p plus 1. Whereas, if I get a message from somewhere else and basically say $V T$ of p equal to max of $V T$ of p commit because it may be later than where I am right now. So, I am at a particular point I am not doing anything right now and I get the message from somewhere else.

So, I have to assign a particular time for the event that the message has been received by me delivered to me that means, it has to be connected with either the most recent time I have or the most recent time has as reported by the message itself, but what is incoming message. And since this is a vector we have to do the a entry by entry max that is what we have to do because this is our vectors. So, max is basically there are so many entries of fixed set of entries and we have to do max of each one of them. So, this vector clocks give you lot more precision in terms of the exactly the kind of things are going on the system. Again just to remind you basically if you took at a vector time, it basically says for each process in the vector how many events are occurred in that other process causally prior to when e occurred, so that gives you fairly good view of what is going on a system. And you can use that to determine which is before which is after.

So, again if you are designing the causally ordered thing, we have to do this vector clocks if you and then because any way as I mentioned in a group commit system you know how many process where there so therefore, vector clocks are definitely feasible. And then we can figure out what things are let us say causally ordered, and then when you do the delivery you look at the times or the vector clocks and decide whether to do it or not. If it is causally or depending on the vector clocks, you can figure out whether it is which order it is coming and then do the appropriate delivery.

(Refer Slide Time: 19:22)



The slide is titled "Orthogonal Choices" and lists four categories of choices in system messaging systems:

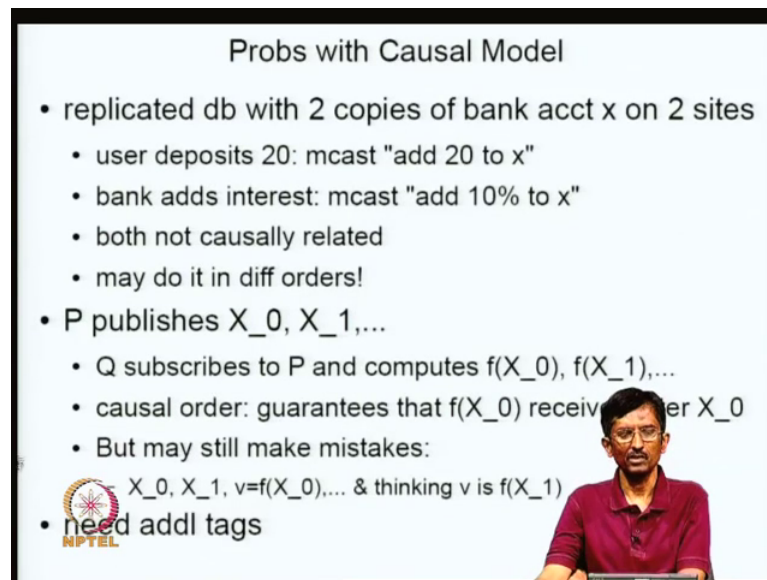
- atomicity (does not specify delivery order but same everywhere)
- delivery order (FIFO, causal, total)
- uniform
- timed (to be delivered within a bounded time)

The slide also features the NPTEL logo in the bottom left corner and a small inset image of a man in a maroon shirt in the bottom right corner.

So, now the important thing to notice is that the lot of orthogonal choices in your design of this kind of system messaging systems. For example, the atomicity a failure atomicity for example, does not specify delivery order. And also we want to ensure that the delivery order for example, could be FIFO causal or total. It can be uniform or not uniform basically you are deciding to say whether some whether you have concerned about processes that fail or not basically I want correct processes or not correct processes, so various possibilities. It can even be timed by as I mentioned earlier that synchronization is very difficult, but it is possible that you can design protocols with time also into an account. Basically if you have some kind of timeouts and then you can decide whether a particular multicast or broad cast has be timed out because not received in particular amount of time. So, various choices are there and different designers have decide play around with various aspects of this.

And each one of them has certain difficulties for example, dynamically uniform is for more difficult, similarly causal requires you to keep set of vector, timestamps that is more difficult than for example, if you do not have any delivery order of specified etcetera. So, this also requires some kind of synchronization, so one to gets a various levels of difficulty in all these models.

(Refer Slide Time: 21:14)



Probs with Causal Model

- replicated db with 2 copies of bank acct x on 2 sites
 - user deposits 20: mcast "add 20 to x"
 - bank adds interest: mcast "add 10% to x"
 - both not causally related
 - may do it in diff orders!
- P publishes X_0, X_1, \dots
 - Q subscribes to P and computes $f(X_0), f(X_1), \dots$
 - causal order: guarantees that $f(X_0)$ receives X_0 before X_1
 - But may still make mistakes:
 $X_0, X_1, v=f(X_0), \dots$ & thinking v is $f(X_1)$
- need addl tags

NPTEL

Now, let me just mention that this causal model actually can be sometimes misleading. So, one has to be careful you have to see exactly what its capabilities are for example, consider a replicated database and we have two copies of bank account on two sites. Now, it is possible that I have multiple operations going on. What is the first one doing first one says deposits 20 units of currency. So, what is equivalent of that I have to multicast add 22 x, but. So, this one user is doing this.

But bank independently add some fixed point time it can add interest every so often and says multicast add ten percent to x. What is that these two things are not causally related the fact that I am depositing 20 it is not connected with bank add in interest. But what is a big problem the problem is depending on when I do this right I can get different results. For example, if I do at 10 percent x before I do this I will get one result, if I do this first followed that 10 percent I can definite do that right. So, this things are not causally related, but if you do it in different order, so you can get different is also. So, what is the understanding that you have to have its not specified by this causal model, because this an application level understanding, the system has not given us enough information to decide what is to be done. So, it can happen any order. So, you cannot guaranteed anything.

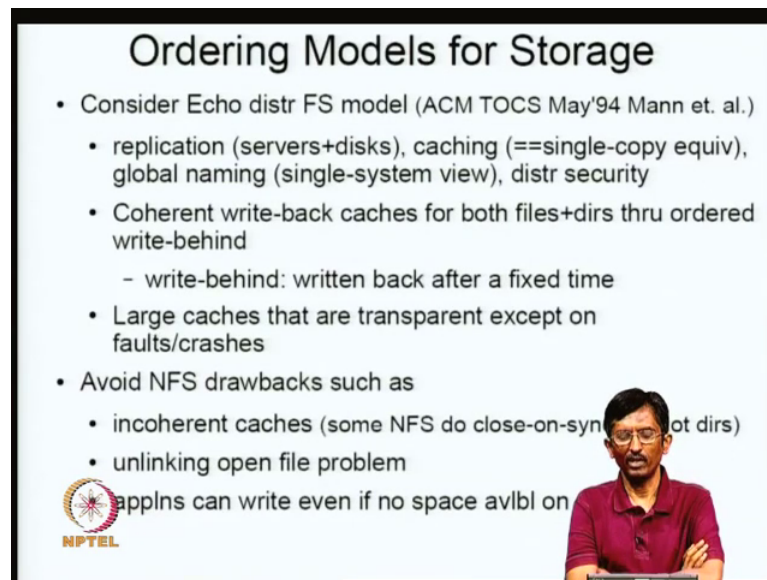
There has been a big controversy in this virtual synchrony models and those people working at application levels as I mentioned this whole year as full of controversies from

with respect to mathematical models with respect to whether this virtual synchrony should be done at which level should be done at the application level, should be done at the system level etcetera. So, depending on all these things we have different results we do at system levels often time we do not have an application level inside. So, you may not be able to do anything you might think it is not causally related. And we go ahead and do it and might get different results depending on which one actually happens.

The other issues similar to this if you take a look at let us say that we have two entities P and Q. P is publishing certain values it is computing, it is basically it is a published subscribed paradigm. So, P is publishing these various values and Q is picking up these values and it is computing some value. Now, because Q is receiving you can actually go with a causal model because once x_0 has been published definitely Q will get it, and because there is a message between P and Q. Therefore, this will be causally connected and therefore, you can guarantee that f of x_0 is basically come receive the after x_0 that you can guarantee. But our problem is that you may not be able to figure out which one is which. See this you may think that x_0, x_1, v of for example, I want to operate and therefore, x_0 , but it may be that I can actually take x_1 and compute and think that it is connected with the first one, whereas actually it is connected with the second one.



So, the thing that is correspondence between the value that has been published which are coming on a stream and what I do with it they might be not causally connected. So, if you want to do that you need to have additional information which says that this is a first value this is a second value etcetera, the causal ordering does not itself give you that. Again this is related to the fact that we do not have application level inside into what is going on and all I can do is I see values and I am doing f of something. So, these protocols have to be carefully constructed, so that this kind of confusions do not occur. And now any problem is our system level we do not have this kind of application or information. So, you really may not be able to do much. So, if I am concerned about those semantics, which are dependent on such things we have to take control at higher levels it can be done at lower levels.

(Refer Slide Time: 26:22)



Ordering Models for Storage

- Consider Echo distr FS model (ACM TOCS May'94 Mann et. al.)
 - replication (servers+disks), caching (==single-copy equiv), global naming (single-system view), distr security
 - Coherent write-back caches for both files+dirs thru ordered write-behind
 - write-behind: written back after a fixed time
 - Large caches that are transparent except on faults/crashes
- Avoid NFS drawbacks such as
 - incoherent caches (some NFS do close-on-syn not dirs)
 - unlinking open file problem
 - applns can write even if no space avlbl on

So, with that I want to start moving to a slightly different topic. So, we are looking at how to order events. And we are talking about how things can fail at the messaging level I want to start thinking about what happens in case I do it at the storage level. We previously looked at NFS model and because of the way does caching, it was not going to be completely consistent. So, now, let us take an example of a different type of file system the echo distributed file system that tries to be giving you good semantics as far as possible close to unique semantics posix semantics. As I mentioned clustering kind of file systems are basically these essentially same as this distributed file systems which give you posix semantics.

Now let us see how this particular design was attempted exactly what we are try to do what kind of ordering that they wanted to put in and how this if you are thinking in terms of is virtual synchrony models. How they could actually be used here and why storage itself act somewhat with the types of issues into the picture. So, we will try to look into some of these things. So, what is echo distributed file system, it has got replication. So, were many servers and we want disks are implicated, servers are replicated it is got a caching as an important part of its design. But it wants to guarantee single copy equivalent. Now, these are different from NFS because NFS has a semi looser notion therefore, it cannot really guarantee this single copy equivalence.

Other thing it has got global naming again this is differs from NFS essentially it has got a single system view. We notice that in NFS depending on very mount the file system you get different names for a same thing, whereas here you want to have a single system view, so that global naming also is an important aspect of this particular system. Again there is an important is that, if I writing applications you look for a particular file in the application and it has been mounted somewhere else then their application will fail. Whereas, if you have a global naming that does not matter if you always be at the same place. So, it is slightly easier thing to work with when you have global naming which is not there in NFS. They also want to look into distributed security. Now, we will not spend much time on this part of it here.

So, what is the one of the most important thing they did was what is called coherent write back caches for both files and directories through ordered write behind. So, again this is where ordering comes in. So, what is the write behind, it is basically something which is written back out of fixed amount of time. I think as I mentioned earlier in many file systems you have various models you have what is called synchronous models where you going write it. The application waits till the write has been committed only when it comes back and says it is done, it will proceeds that is a synchronous model. And it is quite slow because you are actually going to be suffering displacency which is going to be considerable compare to what CPUs and memories can work the speed at with they can work at.

This other model is a asynchronous model where you say something more relaxed model. What do you do is you go to the storage system and say please store it for me, but knowing fully it is going to take a long time, I do not want to wait till the completion. All I am guarantees it that the request has been queued, but I hoping that I am extremely optimized I am just basically saying it will get done sometime that is basically what is the model of asynchronous. I take the table to queue it, but I will not wait for it to complete I just go back and do something else that is the synchronous model.

Then another ones call delayed writes, which is basically the write behind. Delayed writes is basically I do not even queue to the disk I leave it in memory and I hope that someone or whether system or because some other reasons, all the stuff which I have written and made duty in the memory it actually gets written back to memory in some fashion. I am just hoping what other parties to flush it to disk, so this is basically these


also write behind. Now, what is an issue with write behind, the write behind essentially if you it is not ordered then something which got dirtied much later can get flush to disk and then we can suffer a crash and in the previous write which now made into disk may not there in. Sometimes you have there is an inconsistent version of the system of the file in the system.

So, I want to do ordering of if I can do ordering of write behind then I can do something reasonable that is one thing which this particular echo file system attempts to do. So, again this particular system as got a large caches then these things are supposed to be completely transparent except one fault and caches that is even that is a cache problem system except when there is a problem. And as I mentioned earlier this particular design tries to avoid certain drawbacks such as incoherent caches like NFS. NFS also does some close on sync that is whatever file is closely caches to disk, but it does not do the same thing with respect to directories, we also look at this open file problem. If it is deleted, I think we mentioned that NFS has a certain work around for this problem, but this is not complete in all aspects. So, the other issues in NFS which has to be handled and so this particular model tries to do this.

(Refer Slide Time: 33:16)

Echo

- FS data changed only by a write that is logically done at a distinct point in time
 - A fault causes some writes to be discarded
=>undone at a distinct point in time
- On network partition, a client blocks or is returned error on timeout (NFS hard/soft)
- readops (stat, open, rd, ls, lookupn, ...)
- writeops(wr, create, mkdir, rename, fsync, ...)



Let us see what they have done. So, again if you are able to do this they have a particular model basically file system data changed only the write that is logically done at a distinct point in time that means, even if it is done over a period of time, you can logically see it


has been done at a precisely one single point in time. So, it is essentially similar to what we discussed in the case of serializability in databases or when it comes to the closely synchronous model or message delivery we can this also tries to do something similar.

So, from the same reason if a fault occurs, this causes certain writes to be discarded again this undone at a distinct point in time, so that you can reason about it without too much trouble. So, again when there is a network partition we have a block or you are told that there is a time out similar to NFS hard and soft models. So, there is a read operation system and there are write operation system. So, see it is roughly the model of echo.

(Refer Slide Time: 34:45)

Echo

- Ordered & stable writes needed if writes can be discarded at any time
 - write requested by one client and observed by another: write should be stable
 - writes on same obj should be stable in logical order
 - overwrites (length preserving) by one client can be reordered as an "opt"
 - overwrite failure-atomic if only one block modified
 - fsync on dir and files should make them stable
 - forder: constrains ordering of write; returns imm unlike fsync

 forder(f1, f2,...): f1, f2,... logically performed before any ops after rename(/a/f, /b/f) with /b/f existing: modifies a, b, /a/f, /b/f (4 ops)

And we will like to exactly start modeling as facilitating to the ordering of writes. So, if you want to keep the file system consistent, then if you going to drop some writes because as I mentioned because of faults because of faults some rights can be discarded because they are sitting in memory system. And we are saying that it is undone at a distinct point in time and we do not want any inconsistent results. Basically in some sense it should look as of time is also progressing in a forward direction, it is not look as though time has moved backwards for some operations, but it is something done and something has moved forward.

Those kind of inconsistency should not happen so that is why I need to do ordering and ensure that it consistent way of flushing writes to stable storage has to happen; for example, this one of the condition some of the condition that we have to satisfy, write

requested by one client and observed by another. So, basically if that is a case is already the observed by another that means that it has to be stable. The minute I request write to be done and it has been seen by somebody else. Now, I cannot just living it in the memory and is it there is power failure I can just drop it because that has been observed by somebody else observed, because this party who is not me has actually seen it before that means, I should definitely be stable, so that if somebody else looks for its also available.

So, this is one critical thing. Write on same subject should be stable in logical order. So, if it is I am writing to the same object they should also become stable in same order, again this also is related to what cache also requires. Cache also requires that if I send multiple writes in some order they should get committed in the disk in the same order that also easier. Now, only thing that they try to relax is respect to overwrites.

What are overwrites, basically I am not increasing size of the file, so that the file. For example, take megabytes and I am always writing inside that 8 megabytes I can I am not really extending the 8 megabytes. So, if there are multiple writes somewhat for example, 1 megabytes, somewhat 4 megabytes, somewhat 5 per kilobyte or set of some multiple blocks, then it can be reorder as an optimization.

So, this is one thing they are willing to relax. So, in this particular file system, they decided that this is not something critical. There is in being, but if you do not want to relax taking other quantizes across multiple files is because you might use of files in a way to know when to order events in the system. Whereas, doing it on the same file is often times not the common idiom it is not done that much that is why this not they have decided that this is that can be relaxed.

And that is even why they have this particular motion that over write is basically failure atomic, it is only one block is modified. Again why this is the case because notice that if you want to do overwrites, it is basically a read modified write you have to read it update it and then flush it back to disk. If you are going to have this reordering as an optimization, then you cannot really guarantee failure transitivity which is not possible that is why these two things are connected. Now, the other thing that they have in mind is some ordering operations fsync is quite well known it basically is a command that can be given by application level to synchronize the file contents which are sitting in memory it

can be flush to disk that is `f sync`. So, it is possible to do it on directories and files, so that when it become stable means it is sitting on persistence storage like disk.

In addition to this they are something called `f order`, because notice that `fsync` is only on particular directory or file at a time, but suppose I am interested in constraining order of multiple files at the multiple directories. I want to specify the order in which some particular operation takes place on certain files then `f order` can be used, and this is something which is not there in `posix`. `Fsync` is there in `posix`, but further is something which is there. So, that you can essentially satisfy this conditions various design aspects about coherent write back cache etcetera for this things you need `f order`. So, this is something is a new thing it is not part of a `posix`.

So, other thing that is different what `infsync` and `f order` is `fsync` basically you wait till it is committed to disk; that means, it is a synchronized operation. And only reason why it is called `fsync` is because it is `f synchronous`. We are sure that at the end of it we have done it. Whereas, `f order` it constrains ordering of write, but it returns immediately unlike `fsync`. Basically this is yes telling you what order should be where intimating to the file system that please take this, please flush in this order it is not a we are not waiting for it to do it, we are just telling it whatever you do keep track of this. And in the future if I am going to flush it please make sure this happens and it has to be in spite of whatever crashes whatever is happened is that of follow this order that is all I am telling you. I am instructing the files include.

For example, suppose I say `f order f 1, f 2` etcetera then basically this `f 1 f 2` etcetera, etcetera they are logically performed before any operations after these have to be done before any operation that come after it. Now, reason why this important is that consider for example, a rename operation. Rename is a fairly complex operation for a file system angle, because what are they doing in rename, this is the source, this is the target and basically what we are saying is that I am going to graft this tree at this place that means, it is a whole trees getting move from one place to another place. That means, we are going to modify the directory `a`, directory `b`, `a slash f b slash f`. As a basic these are all actually they are more than that, but and simply state we can say it is equivalent before operations and if you are operating at four different places on the disk, we have to ensure some model by which in spite of failures, it is being something reasonable.

Semantics is slightly complicated either you can do the following. We can say that I keep track of the way it is happening in case a crash when I come back up again I remember where exactly I was and I redo the rest of the thing that I did not complete that is one model which is typically followed by file systems. Or I can say that I do something and then when a machine crashes I undo whatever I done so far. Now, since file system usually do not have extremely long running transactions, it is simpler to go with redo models.

So, for redoing I need to have some information about the way I have to flush this into disk and that is what order can be use as a way to tell it which way it has to be done. So, and this can be arbitrarily complicated because this tree etcetera can be arbitrary complex. So, the standard way it can be done is for transactional models and most general in file systems do this kind of operations using some log, and idea is instead of using a log can I do order. Order also can constrains ordering something has to be flushed. And some of this things are also used in some commercial file systems similar to order. For example, if you look at the b s d file system, flush file system they use equivalent of order. There is a particular scheme in that system by which all the buffers which are dirty we keep track of the dependencies and then you when you flushing it you look at it dependencies and then you actually flush them in that particular order.

So, in a sense b s b file system has some notion of order, but it is it is not available at the application level because you notice that fsync is a available application level and so is order. Whereas, what b s b is doing is done it in the common level it is not really it is not available at the user level as at to constrains that means, that they have to do some specific things on top of this. So, it is basically slightly less elegant solution than having something is order.

(Refer Slide Time: 45:06)

Echo Model

- Define 2 relations: \rightarrow (data dep) and \Rightarrow (partial order for stable writes)
 - \Rightarrow a subset of \rightarrow
 - both \rightarrow & \Rightarrow transitive
- $o1 \rightarrow o2$ if $o1$ is a write, $o1$ & $o2$ have an operand in common, $o1$ performed logically before $o2$, $o1$ not discarded when $o2$ performed
- $o1 \Rightarrow o2$ if $o1 \rightarrow o2$ and $o1$ & $o2$ writes but not overwrites
- if $o1 \Rightarrow o2$ and $o1$ discarded implies $o2$ discarded
- if $o1 \rightarrow o2$ and $o1$ & $o2$ on diff clients, $o1$ stable when $o2$ performed
- if $o1 \Rightarrow o2$ and $o2$ stable implies $o1$ stable
- if $fsync(f)$ successful, f is stable

So, let us look at further details. So, in this echo model, you need to have two things one is data dependency and one is the ordering in which stable writes are going. Again what is a stable write a stable write is one which is persistent in disk, this persistent it means it cannot be we cannot go back it has been once it has been equivalent of committing to disk. And both these things are required because basically what is happening is that there is something in memory, something in disk and you want to carry the contents of the memory to disk in a particular sensible order, so that is why I am keeping track of data dependencies. And also keeping track of how things have been ordered in the disk. It is a partial order because sometime there is no ordering required it can be go in any order, but sometimes are require a particular order that is why there are two relationship and then let us see exactly what they are doing.

So, it turns out this partial order for stable writes is a subset of data dependencies it has to be because if it is not dependent then you can do it in any order. So, if it has to be order in particular way, it is because there is a reason why several things are going in a particular order that is why you are going to have the stable writes the order in one way and data dependencies are going to be a superset of this particular set of things that has to be done. And these two things are transitive.

And just look at some of the conditions they have imposed. If $o1$ their dependent $o2$, so what is it mean it means that $o2$ is dependent on $o1$. So, $o1$ first and $o2$ is later. $o1$ is a

write and o 1 o 2 have an operand in common, o 1 performed logical before o 2, o 1 not discarded when o 2 performed. That means, that you take the distribution system and if somebody says o 2 has been performed then o 1 cannot be discarded that is even if it is sitting in memory whatever it is you have to some of flush in some particular order in a such a way that o 1 in no circumstances will ever be discarded, the minute to say o 2 is performed. Because o 1 is prior to o 2 and there is a data dependency is there.

If o 1 there is a partial order with respect to o 1 and o 2, what is this saying if this is a case only o 1 is or let us say o 2 depends on o 1, o 1 and o 2 are writes, but not overwrites. So, what are we saying here, we are saying that if you are going to do something because these two are writes, I am trying to ensure that if o 1 goes to disk then o 2 could be discarded, we are not really saying o 2 should be the fact that o 1 has gone does not mean that o 2 has to be also be (Refer Time: 49:04) does not matter. This one already saying is that there is a order in which these two have to become stable that is all it says. They already have a connection with respect to the dependence, but I am also telling it how it should happen in the case I flush to disk.

What about this if o 1 followed by this arrow o 2, o 1 discarded implies o 2 discarded. So, this is opposite of this, basically in case what did I saying this is basically o 1 should be if in case I am going to make things stable then if it is turns out that o 1 has been discarded that means, o 2 should also be discarded, this another condition. Now, if o 2 is dependent on o 1 and o 1 and o 2 on different clients, o 1 stable when o 2 performed. So, if operation o 2 is done on a different client, I also have to make sure that o 1 is stable, so that other parties also can notice it. So, if it is not stable then I cannot guarantee that o 2 is performed that o 1 can actually crash that write to operation cut the node on which o 1 was done can crash and then there is no way to do that is what o 1 has to be stable .

Now, if o 1 in terms of ordering for stable writes if o 1 is has to be made stable before o 2 then if o 2 is stable that means, o 1 has to be stable and if fsync of f successful, f is stable now. So, you have various rules of this kind which is basically tell you how you should structure your flushing of disks. There is also as I mentioned earlier is also forder also is there. And this also will tell you if it is logically performed that means, it has to be made stable in which order has to be done. Now, you can use both forder and fsync and fsync will be heavy handed approach, if your file system is able to use forder if it has got that

motion then the application level ordering through `forder` can be used while the file system to decide which order should flush it and then while keeping all this in mind.

So, the basic condition is a following is that if we have a partial order for stable write you have to ensure that it happens in that particular order. The data dependency is basically just tells you what is let us say what is dependency of one operation or other it does not say anything about what will happen on failures, it does not say anything. Whereas, this one says what should be done with respect to the flushing with other failures, so that is why we need these two notions. And what we are doing here in all this when we describing all this things is to take care of the situation where you will only have the writes in the way this particular order has been specified. We want to make sure that they all flush to this in this particular order and using this you can come up with models for things like the following.

For example, when we talk of rename etcetera we can actually sit and workout like exactly what the application or the let us say the application high level application has to specify and that can be used to ensure that this really happens. Unfortunately this `forder` is not really part of the current API and posix. So, what you have to do is to do `fsync` has able to order things. So, it is turns out to be for more let us say costing. In future when we have things like flush storage and other kinds of flush storage it probably I think that will be lot more concurrent activity in the storage system, I think at that time probably people will probably start designing API s which correspond to think like `forder` and that might actually make lot more parallelism possible.

Currently the reason why it is not done is the number of disks is quite small typically. And the need for parallelism across the multiple disk is not that very high it is there in big databases and some other systems. But in most systems the number of disk are rather small, therefore we do not have this particular need to figure it out how to have some partial order, so that many things can be flush to disk can relax orders. And only those syncs that need to be order in the particular they can be specified and then flush to in that particular order. So, I think with storage class memories are etcetera what will happen is that this lot more let us say subdivisions in that storage system which can be run in parallel. And then I think all these things were also I think I suspect that this part of the storage system will get essentially new APIs, so that this can be done at effective.

I think I will continue next class based on this particular model. I will show examples of how rename etcetera can be done with this particular set of operations.