Storage Systems

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

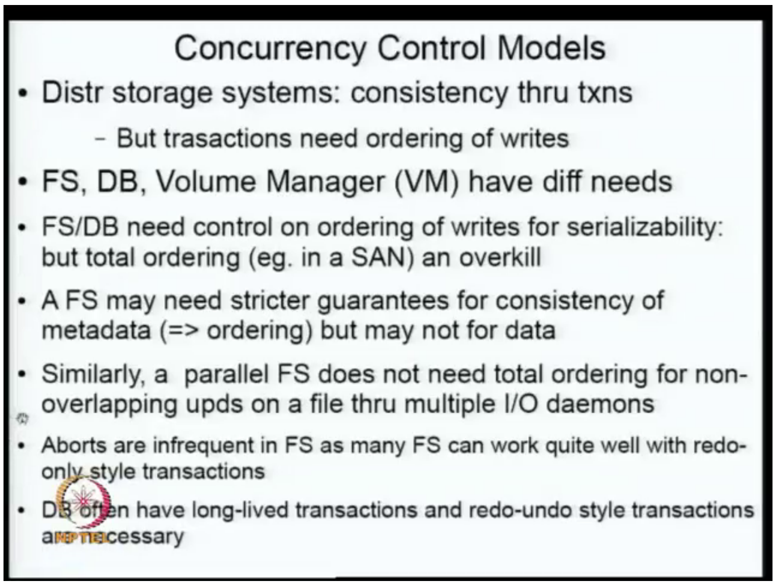
Theoretical Foundations
Lecture – 34
Theoretical foundations of Distributed Storage systems _ Part 3: Group Communication problem

Welcome again to the NPTEL course on storage systems. In the previous class, we were looking at some aspects relating to consistency. First I looked at the atomic commitment problem, then we looked at the consensus problem and then as an example of consensus problem we looked at Paxos.

So, today I want to go a bit in detail with respect to another type of consensus problem, the problem of how to ensure that messages are received across multiple nodes in a what you might to call it consistent manner that is if I send if a node one sends messages sends a message to 2 nodes; a, b, c and then it sends a message 2 to same nodes a, b, c, it is required in some cases that all the nodes a, b, c should receive the first message before the second message and if it is you cannot guarantee this there are some interesting promise that develop.

So, we look at this particular problem this again some kind of consensus problem, but respect to message ordering and these are important in storage systems.

(Refer Slide Time: 01:45)



Concurrency Control Models

- Distr storage systems: consistency thru txns
  - But trasactions need ordering of writes
- FS, DB, Volume Manager (VM) have diff needs
- FS/DB need control on ordering of writes for serializability: but total ordering (eg. in a SAN) an overkill
- A FS may need stricter guarantees for consistency of metadata (=> ordering) but may not for data
- Similarly, a parallel FS does not need total ordering for non-overlapping upds on a file thru multiple I/O daemons
- Aborts are infrequent in FS as many FS can work quite well with redo-only style transactions
- DB often have long-lived transactions and redo-undo style transactions are necessary

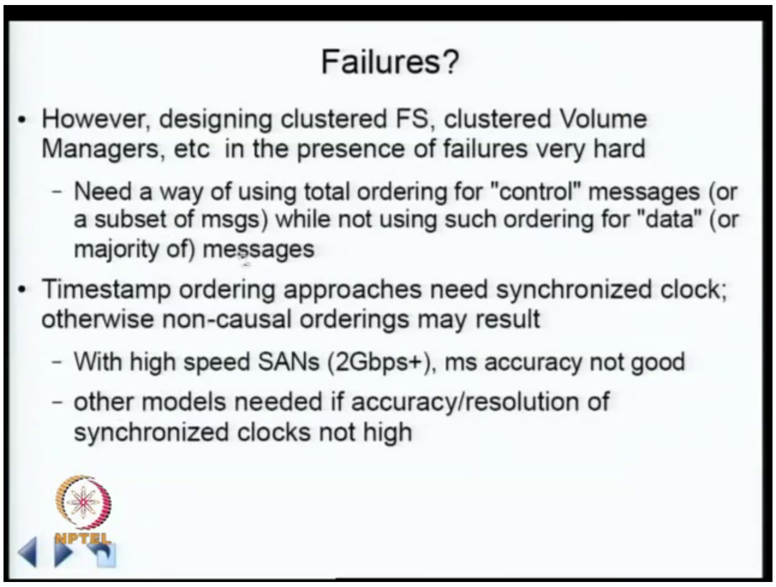
Because storage systems often update this state based on the messages, I get especially distributed file system and distributed databases or distributed volume managers. So, basically I need some kind of a; then which I can control the order of messaging and example, if you look at distributed storage systems, I have consistency files through transactions, but transactions need ordering of writes. So, how this writes are ordered as critical, but what is interesting is that most storage system that as file system databases and volume manager have different needs; for example, for serializability files storage databases need control on ordering of writes.

But if you order it too much it is an overkill because as we noticed earlier this; this systems are very slow storage systems are quite slow and you want to exploit abilities of multiple things going at the same time as possible and so, if you can relax the ordering its better if you force them to go in a particular order all the time, it is too restrictive and many interesting optimization various specific optimizations like; for example, in the case of disk right the elevator of algorithm those kind of things become invisible. So, you do not want too much ordering, but there are sometimes you do require order. So, basically you need to have some good mixture of this 2 mix.

So, again a file system may need stricter guarantees for consistency of metadata, but may not be for data suppose you consider parallel file system in a parallel file system, it may be that a single file is being updated by multiple node at the same time each node having a partial due of the file and how this updates go to the file may not be the exact order may not be that critical in many situations, it will be critical in some situation, but not in our situation. So, what we need is some way of control in this order, how this messages go that the reasons also why this important if you do concurrency control, it turns out in a file system usually you do not have that many aborts because they are whatever transactions we do in file systems; typically are sort of short duration and so, it turns out that the many complexities.

But that is possibility; let me just say that file systems can work quite well with redo only transactions that databases have often long lived transactions you need redo undo style transactions. So, basically your messaging models and concurrency models they have to have certain flexibility and one side spectral does not work here. So, you need to have certain let us say flexibility here.

(Refer Slide Time: 04:50)



Failures?

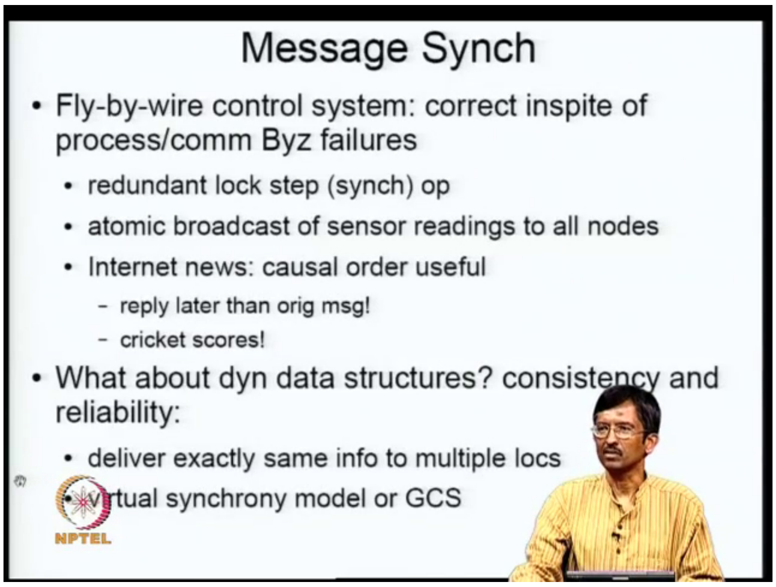
- However, designing clustered FS, clustered Volume Managers, etc in the presence of failures very hard
  - Need a way of using total ordering for "control" messages (or a subset of msgs) while not using such ordering for "data" (or majority of) messages
- Timestamp ordering approaches need synchronized clock; otherwise non-causal orderings may result
  - With high speed SANs (2Gbps+), ms accuracy not good
  - other models needed if accuracy/resolution of synchronized clocks not high

But this makes it even more difficult in place in the presence of failures basically if you have failures what happens is that one note might be aware that a particular node has failed some other node has failed, but other nodes also do not have the same model, then they can be some interesting problematic situation developing so, many times, we can distinguish between control messages and data messages. So, it may be that you need total ordering for control messages whereas, you might need some ordering for data messages for example, certain transactions you might need it.

But for some other situation like parallel file systems may not need it one standard way of ordering messages is to timestamp, but the problem timestamps is that require a synchronized clock. Now it turns out this synchronized clock is not going to be easy to do because the accuracy typically not very good typically; you get only about millisecond accuracy or volts you do not get anything better; it could be in terms of velocity and if we are talking about high speed storage systems in this millisecond accur[acy] certainly not useful. So, you need other models if you need if accuracy and resolution of this clocks are not high you need something else. So, our basic problem is that distributed systems communicate through messages.

And sometimes ordering of the messages is critical and especially this critical for storage systems or other kinds of systems which have which are trying to be resilient to failures and the way they handle failures is by having replicas and so, if some update is done we have to send it to the updates to multiple site at the same time and you have to ensure that once that updates that is logically occurs some other set of updates; they should also happen on this receiving set in the same order if the get in some different order the logic can get completely mixed up that is very critical; that is what you trying to understand.

(Refer Slide Time: 07:24)



Message Synch

- Fly-by-wire control system: correct inspite of process/comm Byz failures
  - redundant lock step (synch) op
  - atomic broadcast of sensor readings to all nodes
  - Internet news: causal order useful
    - reply later than orig msg!
    - cricket scores!
- What about dyn data structures? consistency and reliability:
  - deliver exactly same info to multiple locs
  - virtual synchrony model or GCS

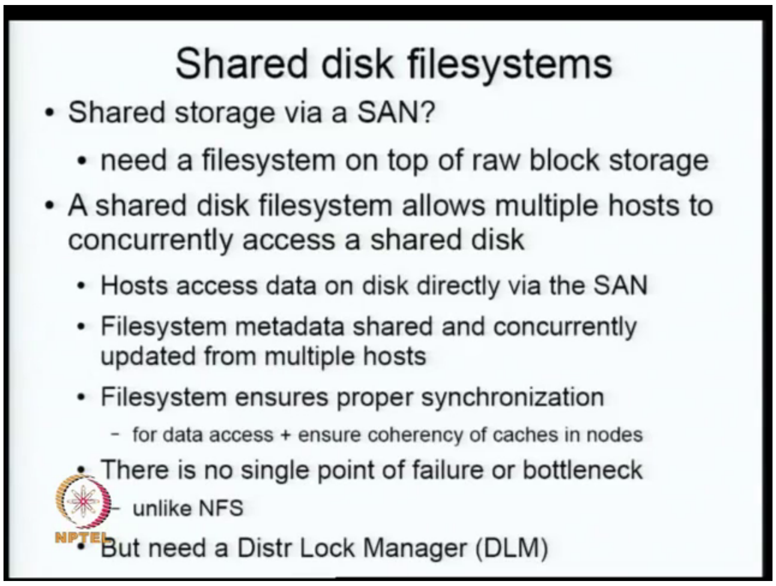
It is also useful in some other context as I mentioned earlier in the previous class real consistence have another desperate need they might need to have message synchronization.

Which is correct in spite of process or communication and failures including byzantine failures you might need something like atomic broadcast of sensor readings to all nodes example consider a aero plane where there are multiple sensor readings and you have to ensure that all those you are doing some computational; all those readings and they have to be essentially of the same approximate time instant, it can be 2 different time instants. So, you need some kind of atomic broadcast and is also useful in other situations; for example, we already considered one in internet for example, if you are having mail systems if you get a reply later than the original message; it is a problem it actually happens in news net if you are talking about some news group it does happen basically.

Because the reply comes earlier than the initial courier whatever it will happens same thing with cricket scores also this things all happen. So, in many situations is very critical, but in some other situations that is just odd or not very pleasant one of those things. So, I also need to figure out how to do this a synchronization storage systems depend on message synchronization for consistency of data if that is not guaranteed anything can happen. So, basically what we are looking for is and want the same information to be delivered into multiple locations in the order that application decide very critical and this is what is often called group communication systems or there is a particular specific model called virtual synchrony model; we will look in to some of these things; I hope I motivated why you need this message ordering as a critical component on distribute systems and basically you need some sense an equivalent of consensus problem.

Because here we all have to agree that that everybody agree that they receive the particular message followed by some other message. So, all this can be seem to be some instance of a consensus problems essentially this things are in principle difficult or impossible N theory, but we can almost always try to do something in spite of the difficult in theory.

(Refer Slide Time: 10:03)


Shared disk filesystems

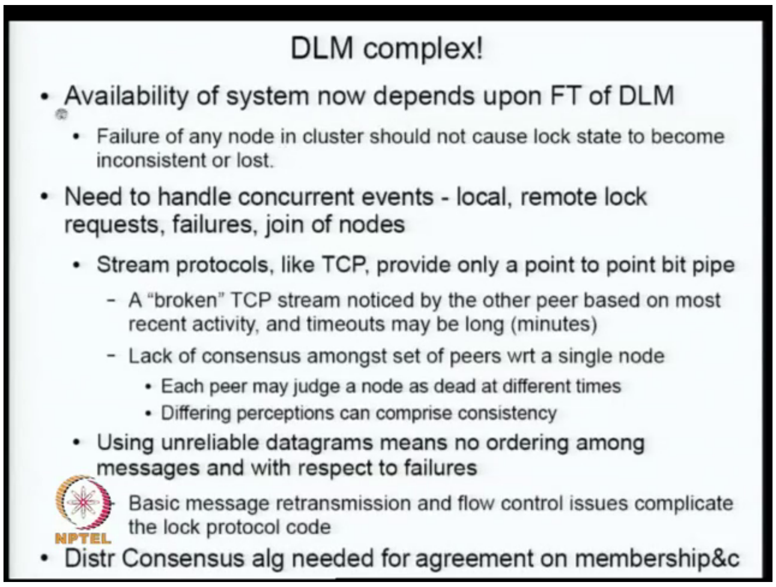
- Shared storage via a SAN?
  - need a filesystem on top of raw block storage
- A shared disk filesystem allows multiple hosts to concurrently access a shared disk
  - Hosts access data on disk directly via the SAN
  - Filesystem metadata shared and concurrently updated from multiple hosts
  - Filesystem ensures proper synchronization
    - for data access + ensure coherency of caches in nodes
  - There is no single point of failure or bottleneck
    - unlike NFS
  - But need a Distr Lock Manager (DLM)

Again let me motivate one more example suppose you want to have shared disk file systems. So, basically what is it mean it means that you have multiple disks which are shared by multiple operating systems and or it could even be the same operating system. So, let us take an another simple case that is multiple disk are there and there is a single file system which is running are all this multiple disk and it is in a storage a network.

So, in essentially what we need to do is the storage a network gives you a block storage and I have to put a file system on it. So, basically all the hosts access data on disk directly via the san the file system metadata shared and concurrently updated from multiple hosts. So, there is basically the reason the reason why want to do is for scalability and throughput and it is the responsibility of the file system to ensure proper synchronization for both data access as well as ensure coherency of cache in the nodes because typically in this system you have cache in and there is cache in you may have to ensure that who were as got a dirty copy of the cache right dirty copy of the data it might have to be revoked and the synch will be sent saying that who were has got the dirty flush out to disk or somebody wants to write to something and some other parties have cached in the read more.

And they all have to flush it and somebody sends the message saying who were got a read only copy please drop just throw it out. So, that somebody else can be given the chance to write to it. So, you need to do a lot of messaging. So, that there is some proper synchronization to both data access as well as for coherency of caches now if this is a case typically the reason why want to go for this kind of models is that you do not want single point of failure or bottleneck and that is why we need what is called a distributed lock manager. So, if you plan to do shared disk file systems you need to do distribute lock manager we also notice NFS as a DLM also that is up usually at the application level.

(Refer Slide Time: 12:41)



DLM complex!

- Availability of system now depends upon FT of DLM
  - Failure of any node in cluster should not cause lock state to become inconsistent or lost.
- Need to handle concurrent events - local, remote lock requests, failures, join of nodes
  - Stream protocols, like TCP, provide only a point to point bit pipe
    - A "broken" TCP stream noticed by the other peer based on most recent activity, and timeouts may be long (minutes)
    - Lack of consensus amongst set of peers wrt a single node
      - Each peer may judge a node as dead at different times
      - Differing perceptions can comprise consistency
  - Using unreliable datagrams means no ordering among messages and with respect to failures
    - Basic message retransmission and flow control issues complicate the lock protocol code
- Distr Consensus alg needed for agreement on membership&c

But it turns out the DLM is a very complex entity and it is nontrivial for various reasons for theoretical reasons as well as the practical reasons first of all you should mention that if there is a DLM; the DLM.

Now, is the only entity that stands in the availability the DLM is everything is now being let us say now everything is seems to be now depends on the fall trends of the DLM. So, failure of any node in cluster should not cause lock state to become inconsistent or lost this critical, but what is our problem why is this so difficult; it is possible that the other lots of interesting concurrent events that can happen the local events there are remote lock requests failures new guys can join, etcetera that is people can exit and enter a system. Now let us think back about why there are some complexities in some of this cases first of all this distribution system will get to know about the system through messaging only that is only we can concurrent anything. So, either we can use TCP kind of protocols or the unreliable datagram protocols like UDP. So, first we used TCP; now

TCP is an excellent protocol it provides a stream model and what it gives you is a way in which if there are any losses it is retransmitted.

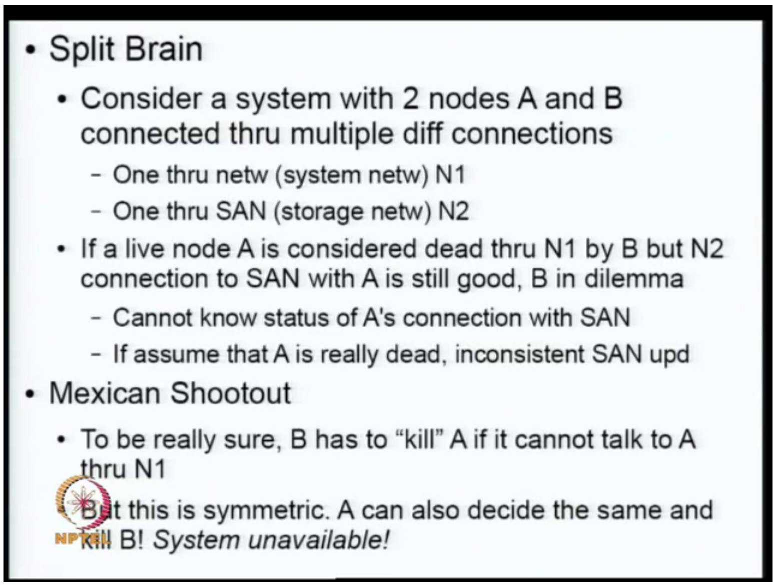
So, that the recipients gets the bites in bites in a in a sequence as. So, intended and that is why it is called a reliable transmission protocol is reliable in the sense that the recipients get a bits in the same order it was sent by the sender. So, the kind of failures it handles or transmission losses not new not no failure that is important to remember it is reliable with respect to losses during transmission that is intermediate networks they drop or the routers that drop the packets for that they have a method of retransmission to handle it, but this particular protocol has not been designed its failure of nodes in a first place properly what do you mean by that suppose, I am sending some piece of data and then my node dies now there is something called keep alive in the received protocol and then these are sent when there is nothing else being sent on the nodes.

So, that to keep the connection alive it is called keep alive and a keep alive timeout is about in terms of minutes that is if I die there is no keep alive for sometime the other node thinks that everything is fine even if I am dead and gone. So, basically a TCP that is broken the stream that is broken is noticed by the other peer based on most recent activity when it does sends messages I meant the recent most recent keep alive come. So, because of this if I am talking to 5 different parties depending on when I talk to those 5 guys each guy will have a different notion when I died; it is not a consistent notion first example I talk to party a first and then I immediately die then a thinks that I am alive for a longer period an somebody who talk much earlier because for him he has the keep alive etcetera will come much later sorry will keep will come much earlier sorry.

So, there is some kind of a lack of consensus amongst set of peers distributed to similar whether it is alive or dead. So, now, if I have different notions of alive and let us say non aliveness they can be some consistency problems I will come to that in a minute by that is a case. So, so stream protocols are not a solution it is also turns out unreliable datagrams are also no solution first of all, they do not give any ordering of any kind at least TCP I think give you a reliable stream it is not able to handle failures.

But it give you reliable set of bits the same ordering which was sent by the sender here it is basically there is no ordering among messages because it is not a stream protocol its basically message oriented so, but this no ordering amongst messages and worse this sent have any flow control etcetera and because it does not have any of this vote reasonable features the application is going to have lot of trouble to work around all this its non trivial. So, basically I need some higher level support and one of thing that we can do is do might need a distributed consensus algorithm for agreement and membership on many other kinds of things. So, again I will explain this further lets go to one example.

(Refer Slide Time: 18:38)



- Split Brain
  - Consider a system with 2 nodes A and B connected thru multiple diff connections
    - One thru netw (system netw) N1
    - One thru SAN (storage netw) N2
  - If a live node A is considered dead thru N1 by B but N2 connection to SAN with A is still good, B in dilemma
    - Cannot know status of A's connection with SAN
    - If assume that A is really dead, inconsistent SAN upd
- Mexican Shootout
  - To be really sure, B has to "kill" A if it cannot talk to A thru N1
  - But this is symmetric. A can also decide the same and kill B! *System unavailable!*

I mention that a TCP; NIP you might have different notions of when a no disk live or dead.

So, I will you know try this with the slightly different example it comes out there are some 2 classic problems called a split brain problem and the Mexican shootout problem what is split brain problem; suppose I have a system with 2 nodes; A and B and that connected through multiple different connections this is a called a natural thing because I might be connected A and B might be connected through the system network that is a gigabit Ethernet; whatever it is I might have gigabit Ethernet running between A and B; therefore, I am connected to that typically the system network I might also be connected through san to a shared storage system that is A and B are writing through san to the shared disk system that is why A and B can; A can put some stuff through san and B can raise from at san they are connected in somewhere they are not connect directly.

But they connected through A san whereas, this N 1 say direct connection A sends message to B directly and B sends directly its going through the regular network suppose a live node A is considered dead through N 1 by B, but N 2 connection to san with A is still good that is B is finding that A is unresponsive again there are lot of certainties here also because it may be that A is temporarily heavily loaded it is just not able to respond. So, by the FLP result basically we were talking about the fact that in a distributed a synchronized system; you cannot agree on if this nodes consensus is possible you know this one faulty node basically because you cannot distinguish between in arbitrary slow node and a dead node this is node to dead. So, if B is for some reason other thing is that a is dead.

But it turns out a still able to talk to san. So, B is in dilemma; it does not know what really situation it cannot figure out a s connection with san of course, there are some protocols varies also with something called recursive reservation, but let us say; we are not doing it, right now because even if you have discuss reservation I will find some other failure mode in which I can show this example that is always a better; I can always reconstruct mythological case assuming that there is no recursive reservation kind of things B cannot find out whether a is actually able to talk to san now it cannot keep on waiting; for example, I am interested in a highly available system if it is ensure it cannot keep on waiting it keeps on waiting; that means, that it has to update something it has to keep updating till it is get a sign of live from a through network N 1; that means, all my system is sort of not available.

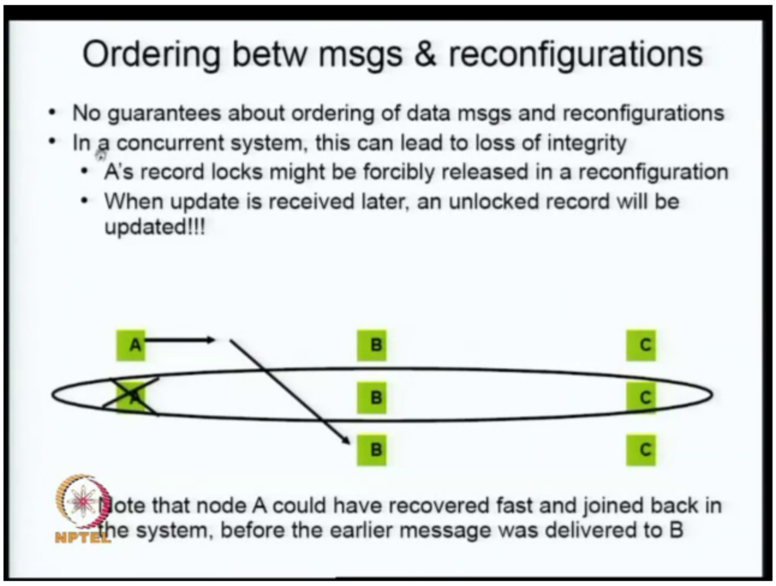
So, if I do not want to get into that unavailability situation, but I still cannot figure out whether a is able to talk to san the only thing is that I can do is I can be optimistic and say that yes, I am not able to talk to a most probably, he is dead, if I do that I am getting to system problematic thing because where is at the case, it may be that a has taken some locks on the san to prevent inconsistent update, but since I cannot figure out whether a is live or dead I cannot wait for him to drop the lock. So, since I cannot wait for him to drop the lock I cannot take a lock I cannot take a lock therefore, what I should do is just go ahead and update without thinking that a is alive, but that can mean that a could also be talking to through san to the same place; that means, there could be inconsistent san update. So, whatever B does is going to be a problem, it does not either; it is unavailable or inconsistency results.

Now, what is one solution out of this problem there is one solution colorfully called as Mexican shootout what is this if B is in dilemma one way to do is to be the bad person says I cannot figure out that in A is alive or dead I will actually indeed make a dead; I will shoot him out because I have update something I cannot figure out if a is never dead one way to do is to have some other mechanism by which I take the power of this system; I just there is some way to yang the power cable from a and this thing is are available, if you talk to any highly available system design they have various things called washed out dilemmas there are various news on which you can remove power for some other node. So, B can decide to be the person to actually kill a to make completely show that he can go on access the san without any competing accessible with that that can make the access inconsistent.

So, B can proceed to kill a because it is not able to talk a through N 1, but our problem is that if the inability to come to talk to a though N 1 by B; it could be the symmetric fault; that means, that if B is not able to talk to A; A also not able to talk to B and A also can decide the same thing exactly the same thing and therefore, he can also decide to yang the power cable from B. So, now, both the systems are down and it is not available now this is the if A designing to highly available storage systems is a problem. So, you can see that under underlining all this stuff is results like a FLP, etcetera; these are all thing that are actually underlying this systems. So, again basically what is a problem if there is a fault you are not have you do not have a consistency view of the where is the faults it can be because of multiple reasons, it could be because of network problems and that often create lot of confusion.

Actually there is even more a serious problem if you remember I talked in the previous class about result where we said that in a camel model c a m e l model when there are crashes a synchronous messages can be lost and it is what is N stand for c a m l Lossy; I am sorry, I forgotten N stands for in that particular class I mentioned that if you are not careful in this particular model it is this possible that you can get arbitrary messages on your on a network which is not connected with any previous previously function existence I will explain this later in an in the next class further mode. So, it turns out that essentially any storage system has to worry about getting all kinds of problematic massages that could be setting there; there has to be somehow made sense of and whatever does not make sense; that is to be dropped on the floor.

(Refer Slide Time: 26:41)



Ordering betw msgs & reconfigurations

- No guarantees about ordering of data msgs and reconfigurations
- In a concurrent system, this can lead to loss of integrity
  - A's record locks might be forcibly released in a reconfiguration
  - When update is received later, an unlocked record will be updated!!!

Note that node A could have recovered fast and joined back in the system, before the earlier message was delivered to B

So, now let us look at in a slightly simpler setting.

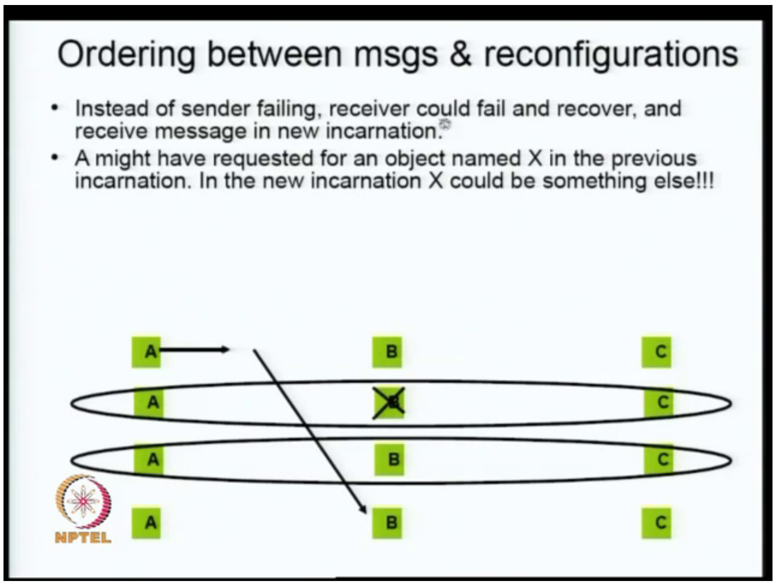
So, again I am talking about ordering between messages in the context of reconfigurations what exactly reconfiguration there is a systems A, B, C and then a dies and then system gets reconfigured. So, that B and C only work that is a reconfiguration and then later a can come up and it can participate in this system again. Now suppose I do not have any guarantees about ordering of data messages and reconfigurations. So, in a concurrent system this can lead to loss of integrity what is that situation similar to the previous situation right in the split brain you can have this insulation here example, let us say that a has taken a lock and it has sent a message to B or before this message may leads relate to B A died and then after the death of a reconfiguration took place and the message was waiting here wherever finally, made is leads to B, let us say the logic, it is written in such a way that whenever the message of this kind comes here its assumes that a has taken a lock.

Therefore it is safe for B to act on it look as people know what A is doing you know the code that particular system therefore, the definition a could not have sent it without taking the lock. So, there B makes a assumption in the context of reconfiguration, it could be better because it might be the case that a might have been a s lock might has been forcibly released in reconfiguration and the message comes in and B if you does not have a clear idea about how the state of the system if it acts upon this message, it can updated without a lock that a should now would have normally taken. So, it is possible that this particular update without a protecting block can be causing system consistency to go away here also we can think about another situation where a dies at any one come back up again and even that also happen.

And so, the various situation that you have to handle if somebody writing the Bs code is wont handle this kind of situation reconfigurations then they might have to think of all this cases and if you really want to be correct in spite of all this problem the code of B can become are quite complicated and sometimes were also happens is that if some messages have been sent to a and this messages can be diverted to B because a died now without any additional high level support for this kind of situations then B might have to think the following these messages are actually for a, but I am handling them these measures actually for me and I am handling myself. So, it has to keep on in its code it has to think about all these things. So, it might say for example, this message came to a a did partly some part of the processing it sends a message.

And now I have to handle all those things I have to remember exactly what state a had done it and died those kind of complexity is B might have to think about. So, this kind of code is completely in practical it can be return, but it will be totally impossible to reason about and to get it right some was impossible it is done people have done it, but it is extremely difficult.

(Refer Slide Time: 30:39)



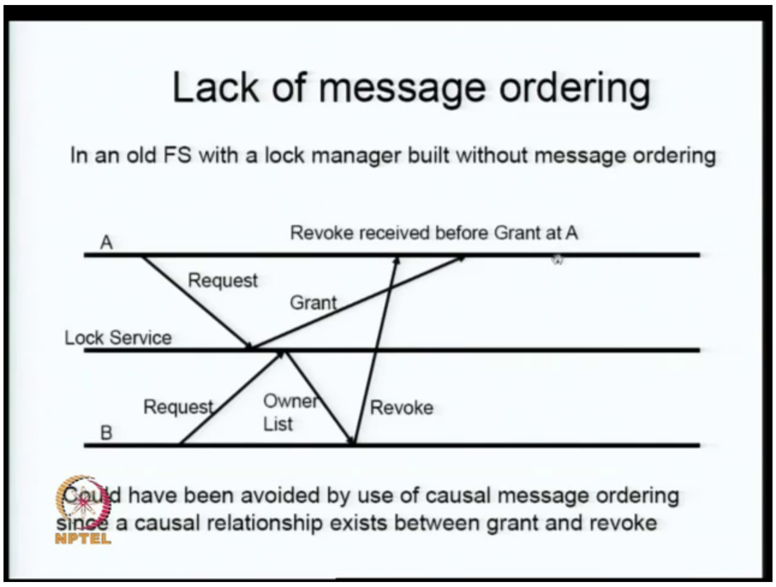
Ordering between msgs & reconfigurations

- Instead of sender failing, receiver could fail and recover, and receive message in new incarnation.
- A might have requested for an object named X in the previous incarnation. In the new incarnation X could be something else!!!

There is also another possibility B in the previous one we talked about the sender failing here we can talk about. So, in the as a receiver can fail. So, since the sender failing receiver could fail and recover and receive message in new incarnation that is A sends something this B died again it rebooted again the messenger send came to B and A might be saying do something about X now this X could be quite different in the new incarnation definitely possible. So, it has to again keep track of history about what happened in some sense you need to know everything that happened before.

So, as you do not have this information it is possible for you to do some wrong thing.

(Refer Slide Time: 31:34)



Lack of message ordering

In an old FS with a lock manager built without message ordering

Could have been avoided by use of causal message ordering since a causal relationship exists between grant and revoke

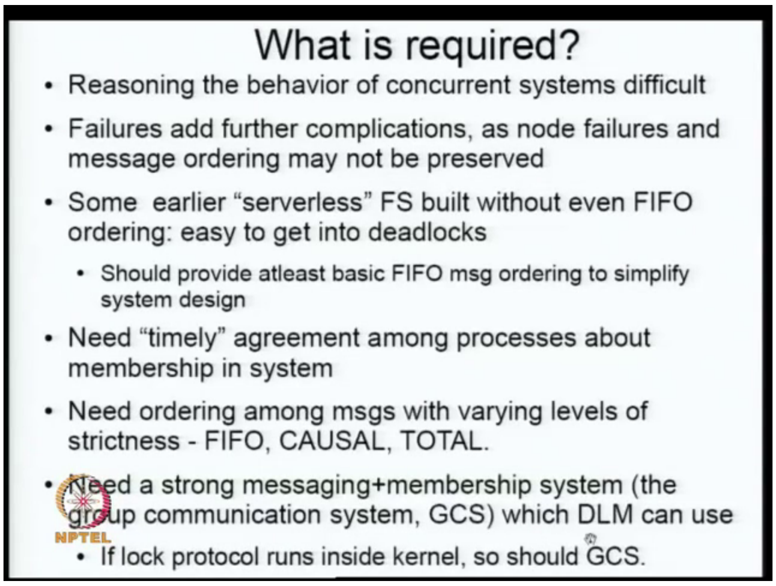
There is also issue that call for if you do not have a message ordering like the following example in some old file system with a lock manager built without this message ordering guarantees you can come across this kind of problems; what is the situation here A is 1 node, there is a lock service there is nodes A and B, there is a lock service; now A request the lock, but the grant of this lock takes a long time sometime; let the B request, it and the lock service what it does is it gives a list of the way this particular file system was constructed it basically says I will tell you who were all owing it you go on talk to them and get a lock from here that is what this owner list.

Basically what does happened is that B has sent a request and it gives you set of owner because basically it has granted it here at this point therefore, it knows who will I have got it. So, this guy now gives saying that I had given it to some party a and probably some other situation some other parties I am going to give that list here now B gets a list of all the guys who have got the lock. So, B now write to a saying please revoke that I want to get hold of that data object please drop the lock you have, but suppose by some aspect of a network this revoke message goes earlier than this one now the logic of A; the program that has been written should be give it to handle this kind of situations. So, basically what we are talking about is it is possible that I can just drop this particular message that may be in this particular situation.

But because other situations where you cannot just say that drop that thing just because saying that logically does not makes sense it may be that really required because a could have been substituting for CD and whatever reasons what we can we come at completely showed of it. So, it turns out that this revoke message is unexpected is just thing often time is to drop it on the floor, but without any guarantee that dropping on the floor is let us say safe you will have to your logic has to look at the whole thing before you decide what to do it is not a again the logic consistency will be complicated. So, here you notice there is something what is called the causal relationship suppose is what is called causal message ordering basically only after grant can get a revoke you cannot.

So, if you are able to use causal message ordering then you can probably reason about it and then decide what is the right thing to do or you figure out some way in which there is what is called causal message ordering you ensure somehow that the revoke actually comes after this grant if you are able to do that then the logic can be simpler that is somebody else is taking care of this message ordering.

(Refer Slide Time: 35:12)



What is required?

- Reasoning the behavior of concurrent systems difficult
- Failures add further complications, as node failures and message ordering may not be preserved
- Some earlier "serverless" FS built without even FIFO ordering: easy to get into deadlocks
  - Should provide atleast basic FIFO msg ordering to simplify system design
- Need "timely" agreement among processes about membership in system
- Need ordering among msgs with varying levels of strictness - FIFO, CAUSAL, TOTAL.
- Need a strong messaging+membership system (the group communication system, GCS) which DLM can use
  - If lock protocol runs inside kernel, so should GCS.

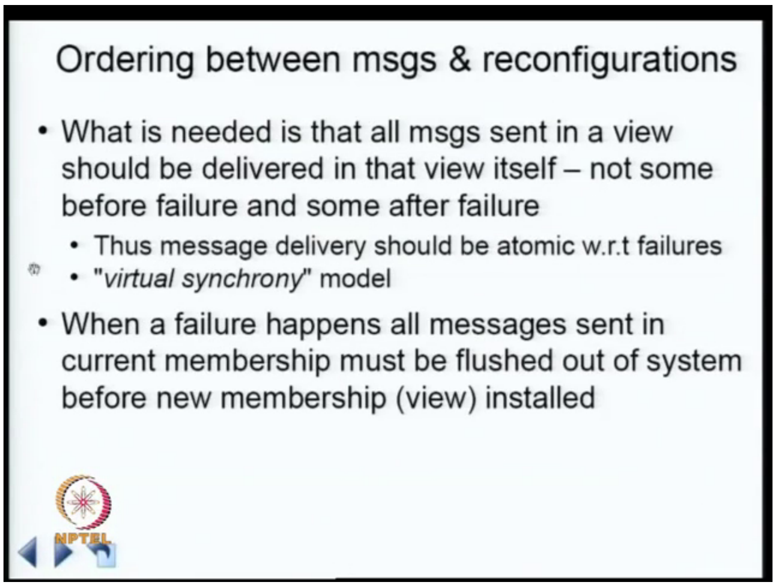
So, what is required one thing to notice is that concurrent systems very difficult to reason about now if you had fill is to it; it becomes that much more complicated and they have been many just like a previous file system and other file system also people were designed without FIFO ordering.

And this get into deadlocks. So, basically it turns out if you really want to have reasonable code at least you should have basic FIFO message ordering other thing that we need is timely agreement among processes is about membership in system now this 2 things are connected one is message ordering other one is who all exist because when I am sending message to replicas I need to know who is there and everything is done through messages; that means, that this 2 problems there is some tight connection and I need to figure out how to do it. So, I need ordering among messages with varying levels of strictness I might have things called FIFO first in first out or causal I will come to that soon basically this is an example of the causal a causal message ordering.

Basically you can say that this things are once you have granted it then only can you revoke it therefore, this has to be somehow met to come like this there is some relationship between the messages. So, it is causal relationship there is also a total ordering basically wait for all the messages to go on other side all the replicas completely make sure that the some that I distributed consensus that is done across all the nodes that I getting the replica messages and once it is completely, then you send the next message it is an example of total ordering. So, essentially we need a strong messaging and membership system which is often called as group communication system and if you have this kind of system, then a distributed lock manager can use it, it can be its makes that life of DLM quite simpler because finally, we have seen looked at it right DLM is going to be your only thing that is stands in between causal order.

Everything is now for your whether how messages are how the data is getting flushed or how the things are getting updated all those things now depend on the DLM. So, you have to ensure that this particular thing is works on it and for this 2 work; we need to have some way of ensuring that when we have replicas and you are sending a broadcasting messages reliable broadcast I need to have single.

(Refer Slide Time: 38:13)



Ordering between msgs & reconfigurations

- What is needed is that all msgs sent in a view should be delivered in that view itself – not some before failure and some after failure
  - Thus message delivery should be atomic w.r.t failures
  - "*virtual synchrony*" model
- When a failure happens all messages sent in current membership must be flushed out of system before new membership (view) installed

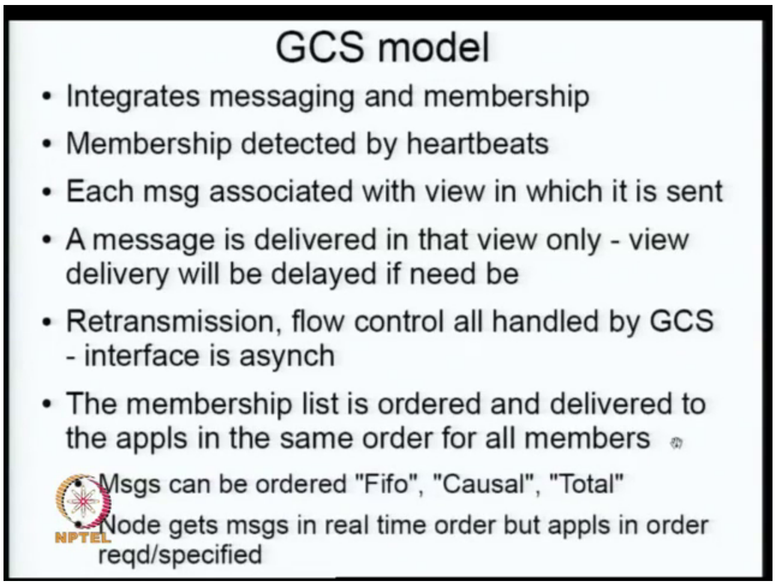
So, one of the models that people have come up with is a following what is here is that all messages sent in a view should be delivered in the view itself what I mean by view is that a view is a particular state of the system with respect to who is live who is dead that is this is with respect to failures every time there is a failure there is a new view. So, the system basically can think of it as a set of views each you bring a coherent set of nodes working together when a failure occurs then you get a new view.

So, what is needed is that all messages sent in a view for example, I am in a particular view that of bunch of nodes talking together; if I send the message in that form preferably it should be received in the same view it becomes complicated in the case I send it in one view and it received in a different view; that means, that one message has gone from one view to the other view and the recipients has to figure out what to do with that particular thing even that some node as some nodes some failure taken place it has to some failure process before it can decide what to do. So, basically it is the program become that much simpler to write in case message delivery is atomic with respect to failures this often call the virtual synchrony model again we will go through some of this in some detail.

So, again I just want to do clarify what we need is basically the application should not worry about failures impossible if it gets a message you should have seen that it has come in the same view therefore, it has consistent notion of who is there who is not there and all the parties who are in that particular view or on the same view. So, there is no question of any confusion, if in case the message sent in one view and is going to be given in the other view then the multiple possibilities one thing I can do is I can prevent that message to be given to the new view I can just drop it on the floor that is one possibility or I can do other aspect; suppose I cannot drop the message on the floor is an important message I will essentially stop the whole system in some way prevent anybody else from knowing about the new state of the system.

I give the message and then only tell everybody that I some particular thing has failed I can do either of these 2 things that is basically what it do is the message is received by the node fine with application does not have to know application is stored only the application is given the message then only it is told that some failure taking place. So, when a failure happens all the messages sent in current membership must be flushed out of system before new membership view installed this is the dropping the all the messages before a new. So, these one thing we can do also. So, there is a multiple ways to handle it virtual synchrony is one model is basically tries to ensure that message sent in a view which delivered in that view itself that is what that is what virtual synchrony it happens in the same view.

(Refer Slide Time: 42:04)



GCS model

- Integrates messaging and membership
- Membership detected by heartbeats
- Each msg associated with view in which it is sent
- A message is delivered in that view only - view delivery will be delayed if need be
- Retransmission, flow control all handled by GCS - interface is asynch
- The membership list is ordered and delivered to the appls in the same order for all members
  - Msgs can be ordered "Fifo", "Causal", "Total"
  - Node gets msgs in real time order but appls in order reqd/specified

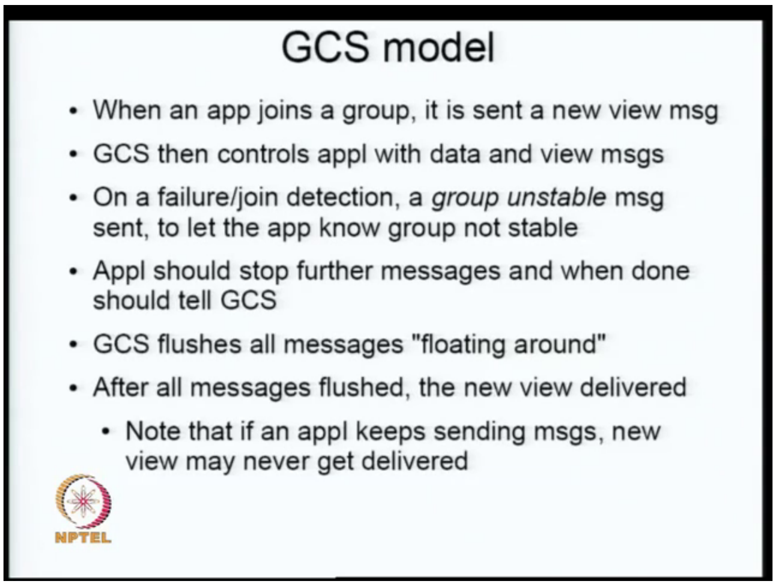
So, the GCS; the group committed system it integrates messaging and membership.

How is a membership detected by what is called heartbeats what is heartbeat roughly node sends a heartbeat messages to every other node and I get information saying that and that other party also sends back they reply that is how you know the other parties alive because there is no other way to know it because everything communicates through messages it can be through electrons or protons, but everything has to go through messages and you have detected only by some messages which are called heartbeats each message is associated with a view in which it is sent the message is delivered in that view only view delivery will be delayed if need be.

Because; that means that if a new view has to be notified to all the nodes; it will be delayed till the message is delivered in that in the previous view. So, that the application does not have to think about that it is seen some unusual new situation has developed it is thinking as if because a old situation. So, other issues like flow control, etcetera also handled by GCS. So, the membership list is ordered and delivered to the applications in the same order for all the members so; that means, that every member has a same idea about who is live and who is dead that is why we not have this kind of spilt brain kind of problems; that means, that someone is magic is doing something unusual it is doing some kind of a consensus which is said it is theoretically impossible, but it is doing something reasonable.

So, messages can be ordered in some multiple ways FIFO causal or total see the most important thing to remember is that node gets messages in real time order, but applications in order required are specified. So, it can see the distinction; distinction is that message are being sent as usual the nodes gets it, but the application is not told of it. So, it is buffered in a curve something like that buffer somewhere typically you can come here and then the application are notified after the new view has been installed or before the new views installed depending on that what you want.

(Refer Slide Time: 44:36)



GCS model

- When an app joins a group, it is sent a new view msg
- GCS then controls appl with data and view msgs
- On a failure/join detection, a *group unstable* msg sent, to let the app know group not stable
- Appl should stop further messages and when done should tell GCS
- GCS flushes all messages "floating around"
- After all messages flushed, the new view delivered
  - Note that if an appl keeps sending msgs, new view may never get delivered

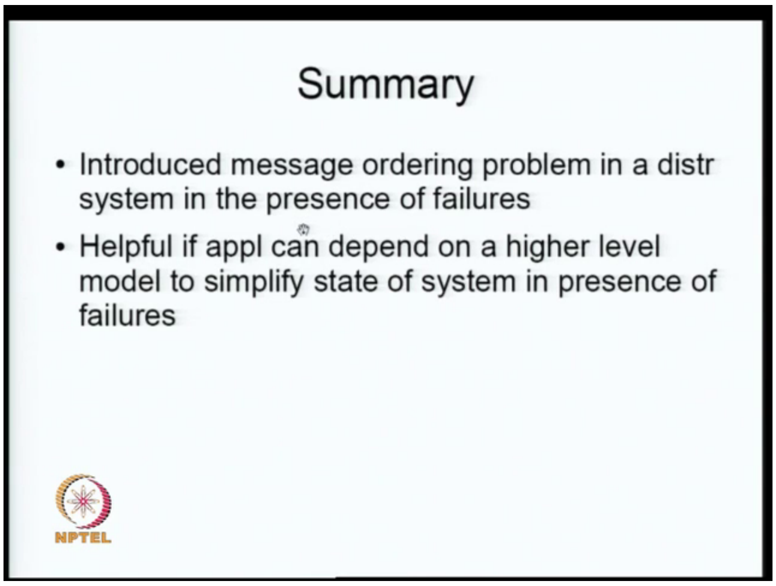
So, in this particular model in application joins a group; it is sent a new view message when as soon as anybody joins a group it is sent a new view it says who all the parties were there and then GCS after from that on onwards in from that time onwards control application with data and view messages it keeps saying everything is fine.

So, far keep on exchanging whatever data you want to exchange then suddenly one failure occurs it is noticed; some particular view you were going; how it is done; it is noticed and then GCS will essentially flush all the messages that has still extent, then it will send a new view message to everybody, then the new then all the nodes will update their nodes idea about what is a current state of the system again, they will proceed further. So, on a failure or join detection a group unstable message sent to let the application know group is not stable. So, what is why is it done. So, that application to stop sending further messages and only when it is done you should have GCS basically something are critical.

I want to do it actually in the same view because applications also has some notion of what is safe to do it has been told that something is happened, but it does not want to react it to immediately because that some urgently is to do if it is figured out all the absolutely critical it has to be done then it goes. Now I am ready to change my view of what is the situation is. So, then GCS flushes all messages floating around I think that lot of details we have to go through it, but I think for the time being we just say that after all messages have flushed the new view is delivered notice that that has to be some understanding between GCS and application with application keeps on sending messages then new view may never get delivered basically because it is a cooperative model where if there is a failure all have been have to cooperate to install a new view.

If a particular node does not; what is say cooperate right and keep sending messages then GCS has to keep waiting because it might think that all this messages critical it has to be somehow sent in that particular view itself because that is a process in that view itself otherwise there is going to be inconsistency is why it has its just some keeps on waiting till that is stop.

(Refer Slide Time: 47:25)



Summary

- Introduced message ordering problem in a distr system in the presence of failures
- Helpful if appl can depend on a higher level model to simplify state of system in presence of failures

So, basically I would like to mention and summary that this is a basically we have a message ordering problem and the reason why it is so complicated because a failures actually make it difficult to figure out what is going on. So, what I need to do is the following I need to somehow get notification of the failures once again we have to failure I know how to figure out what are all the messages are set in the previous view.

And somehow decide what do with that this we will discuss in detail in the next class, I just want to mention that this is a critical part of the problem because it depending on the messages we can do various things you can even delete while file system or something unusual. So, and for that reason you may have to think about how to ensure that if you act on a message it is for the right reasons and sometimes the right reasons are meaningful if you look at the ordering nature has come. So, all basic idea is that suppose an application can depend on from higher level model and what is that high level model we talked about group communication systems or the virtual synchronized model if that is there then what will happen is that every party in that system has a consistent view of who are in the group and therefore, any decision any logic that is written can be consistent this incorporates.

Whether it is what is a cache consistency whether it is a question of updating a san various kinds of things for all this things I really need to know exactly who is there hosting on there if this basic thing is not available it becomes complicated now hence that been various models for group communication systems with different notions and different capabilities and for example, a causal model is slightly more difficult than FIFO model because I have to track dependencies across the various event of the system and a total model is a easy model for an application to write, but it is to restrictive because every message essentially has to be delivered to all the replicas only then you can think of sending the next one in a sense, it is serializes everything and this can be a low performance system.

So, ideal thing would be to have somewhere in between this extremes it have a causal system it turns out to be a reasonably good compromise. So, the systems which employ this kind of models I think we will next class, we look at how this consistency this message consistency models are actual engineered.

Thank you.