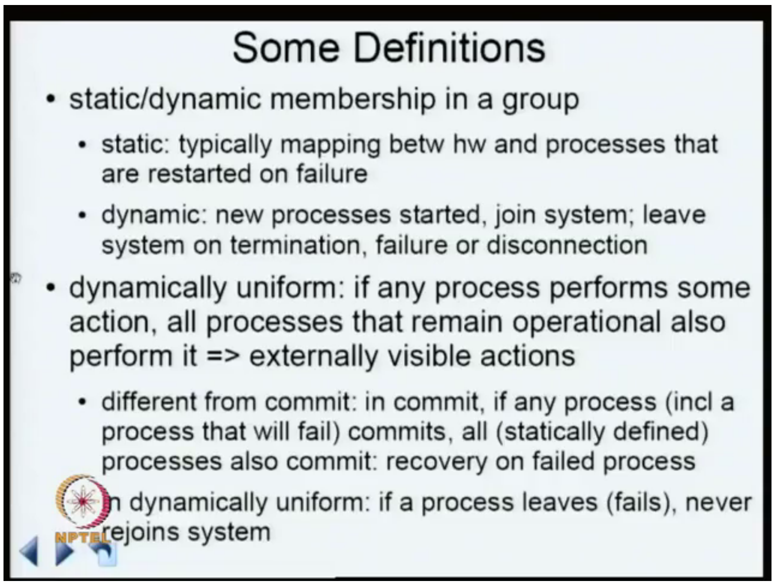
Storage Systems
Dr. K. Gopinath
Department of Computer Science and Engineering
Indian Institute of Technology, Bangalore

Theoretical Foundations
Lecture – 35
Theoretical foundations of Distributed Storage systems_Part 4: Models for message ordering in the presence of failures

Welcome again to the NPTL course on storage systems. In the previous class, we were looking at how message ordering, it should not done well can create complications, and today we will look at this in more detail. As I first mention that this is a slightly tricky subject. And there has been a lot of lies lot of research was done in the last 20 years or so and there has been lot of lot of let us say, controversy on some aspects also. Example the people would build a systems, they might have a certain model of how thing should work. And then if people who are more trained at the mathematical side of things they come and look at it they might have different prospective on what is going on. There has been interesting exchange of ideas or even controversy among these 2 communities, and this area is one such interesting area. It is so, there is lot of certainty and depth in this area.

So, I will try to mention some aspect of it in this particular talk today.

(Refer Slide Time: 01:31)



Some Definitions

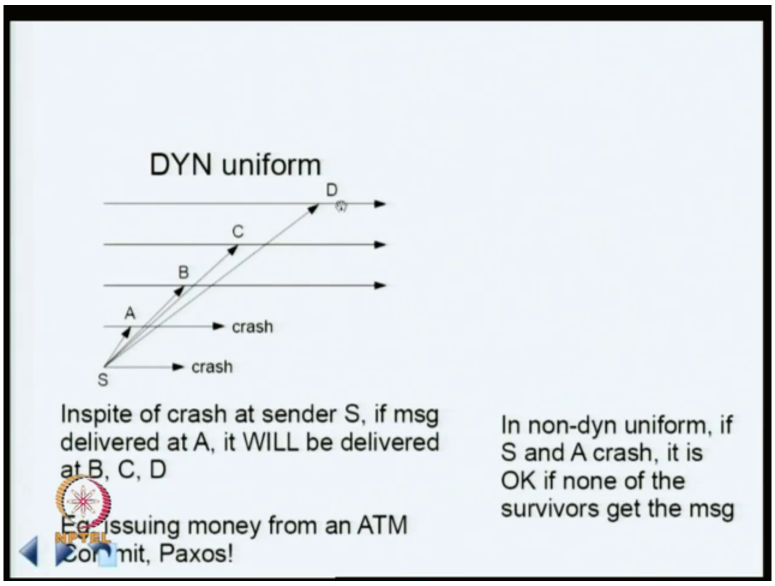
- static/dynamic membership in a group
  - static: typically mapping betw hw and processes that are restarted on failure
  - dynamic: new processes started, join system; leave system on termination, failure or disconnection
- dynamically uniform: if any process performs some action, all processes that remain operational also perform it => externally visible actions
  - different from commit: in commit, if any process (incl a process that will fail) commits, all (statically defined) processes also commit: recovery on failed process
  - in dynamically uniform: if a process leaves (fails), never rejoins system

So, let us look at again the whole issues about message ordering in the context of failures and how if you can somehow make the job of the programmer easier, that is a whole idea. So, let us first make some definitions. First of all, we can have a these are a notion of a static and dynamic membership in a group. What is static; this typically mapping between some hardware or some entity and processes. Now we started on failure. So, the typical model is that it is restarted an failure. A dynamic situation, you start a new process it joins a system, and then it releases system on termination failure disconnection never comes back again. Again, it is somebody else comes into as a new processes it is not the old thing that was restarted.

So, in this case it is always somewhat static, it is something dies it comes back again with it is same ID in some sense. Here the ID's are keeping on changing if somebody comes and goes. So, that is one part of one definition. That is how something called dynamically uniform which has such are more settled concept. Let me explain what this, is if any process perform some action all process is that remain operational also perform it. What is that mean? I could do something and then if I end up doing that then whoever is in that group should also do it irrespective whether I live or die. So, somebody says very critical that is happens for example, in the commit kind of protocol, if one of them is committed everybody also should also commit. Again, just take a slight different example suppose I have a parallel file system. And different processes are doing some computations on different portions of the file.

It is basically parallel file system, the file is a very long one let us say it is about 10 gigabytes, there are 24 processes or 25 whatever 100 processes working on that one 10 gigabyte file. Each one has got a certain domain in of is operating if one of them updates one part of it and dies. It is important that other processes is irrespective whatever happened they also should update it otherwise, what will happen what will happen is that you will have some updates from this guy, but no updates for that guys. So, it is some kind of a mixture between some updates made by this party and no updates in other parties. So, those things are somewhat problematic.

(Refer Slide Time: 04:21)



DYN uniform

Inspite of crash at sender S, if msg delivered at A, it WILL be delivered at B, C, D

Eg issuing money from an ATM
Commit, Paxos!

In non-dyn uniform, if S and A crash, it is OK if none of the survivors get the msg

So, what is in dynamically uniform again I will show it to by an example by diagram. Suppose you have a; some sender S it is doing a multi-cast. So, it is going to A B C D.

Now, S sends a multi-cast and it dies. Similarly, A gets it, it does whatever has been done and dies. And this B C D they are survivor. Our interest is that if a gets something and does something because of receiving that message from S. It is very important that B C D also actually act on it. This what they call dynamically uniform irrespective of what failures the minute a message is delivered to one party it is absolutely critical that everybody else also. Sees even if the over has got it die before other party get the message. So, again in spite of crash at sender S, if message delivered at A, it will be delivered at B C D. So, for example, that is my commit protocols or paxos, etcetera, they have this model if somebody has committed somewhere does not matter if that parties not living you also have to commit. Similarly, in paxos if somebody has in the consensus kind of algorithm right.

If somebody has accepted this, and there is a majority, and there has been accepted as the; for that particular round as the agreed upon value everybody also should do it irrespective of who lives and who dies. So, this is a critical thing in many situation, but the other situations is what is called non-dynamically uniform. If I send a crash and we can even assume that there is some kind of a partition. I send some stuff in this guy there were some partition that came along therefore, this messages somewhat never made it to remain in this places. So, this application here never receive that message. And it is most kind of situations to not in the survivor not get the message and nothing can be matters. For example, if you are air traffic control system somebody is giving updates of the current aircraft that is out there.

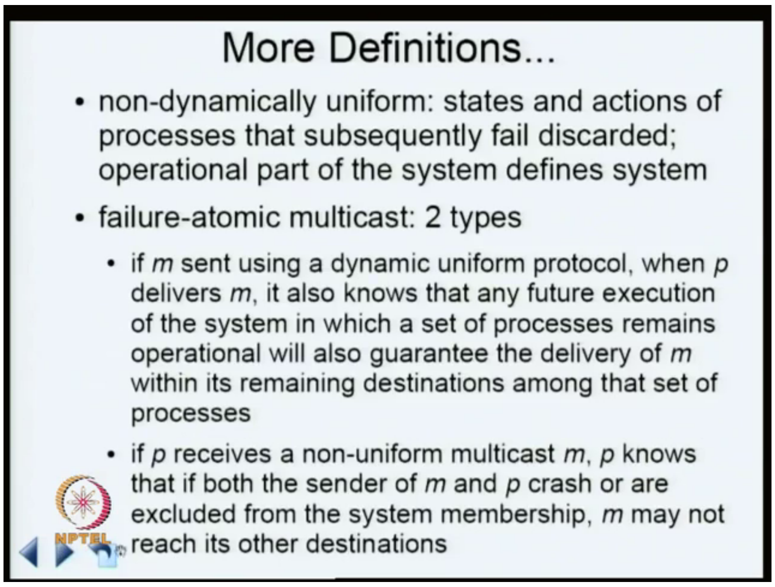
This could be displace station A B C D, the fact that with a new updated view of what air craft is being seen is not aware with B C D. It is not it is not really seriousness issue. Whereas, if I am issuing money from an ATM for example. And it has debited the money from that from the database or whatever, but that ATM itself has a mechanical mall function right. It does not give the money that is a bad situation, right. Basically, if in case it has committed there irrespective, what happens? Somebody has to know that is ATM mall function then it will actually keep track of update and again try to deliver it money to you somehow. If you does it job of debiting it from the database your account, but does not take care to ensure that you get your money. Then something is wrong because you can see all this is are distributed this place and time, something is happening database somewhere else something is happening at the ATM mechanical contraption that is giving you the money.

And if it is a dynamically uniform system then if something has been happen something is happened in the database it will happen also at the mechanical contraction of ATM where you are using it. It cannot just assume that the database failed, and none of this fellows were none of the ATMs were or the ATM was informed about it you can keep quiet, that does not work it is not a sensible thing. So, that is where so, this are dynamically uniform system which is a slightly more complicated system. Basically, because there are external visible actions, that is critical. So, there are 2 or 3 issues here this is slightly different from the commit also, I mention that dynamically uniform. And commit are similar, but there is some difference there also. So, if you look at this particular diagram, if some of these guys die; it is that this guy do not see it whereas, in commit protocol.

Because you are assuming that there is going to be restarted, they have to be it is important that the message actually goes to them also. So, basically in a dynamically uniform system what is guarantees that if party does something, right? The all processes that remain operational, that is important. That is a difference between commit, and dynamically uniform it is slightly some there is some settled here. That we should remember basically what is means is that one processes has done something who are remains operational from that time onwards it should also do it. But is in a commit protocol all of them have to do it irrespective of failures, because all of them have to commit. So, if some other fail they have to restart again they have to set the system, right and then do it that is a difference there.

So, in a sense makes a dynamic uniform is somewhere between the model of commit, and the one which is the previous model the different from the dynamic one and the one which is the commit one there is some differences. So, that is one. So, in a non-uniformly dynamic system, it is states and actions of processes that subsequently. Fail discarded operational part of the system defines the system.

(Refer Slide Time: 10:24)


More Definitions...

- non-dynamically uniform: states and actions of processes that subsequently fail discarded; operational part of the system defines system
- failure-atomic multicast: 2 types
  - if *m* sent using a dynamic uniform protocol, when *p* delivers *m*, it also knows that any future execution of the system in which a set of processes remains operational will also guarantee the delivery of *m* within its remaining destinations among that set of processes
  - if *p* receives a non-uniform multicast *m*, *p* knows that if both the sender of *m* and *p* crash or are excluded from the system membership, *m* may not reach its other destinations

So, this is a like a more easier see just imagine what has to be done in the case of dynamically uniform. What you have to do is if any time a message goes to for let us say this are nodes, there is some nodes 1 2 3 4, the processes A B C D so, message goes to node 1.

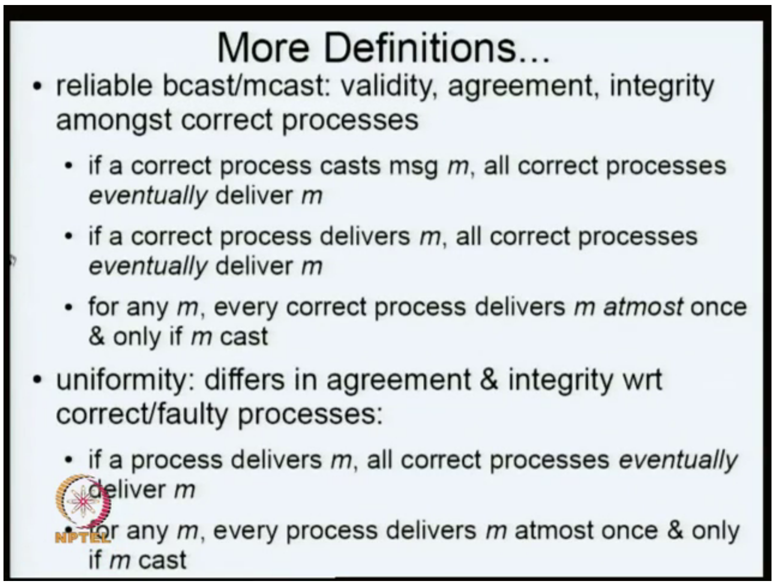
Now, what you have to do for dynamically uniform is you have to ensure that these messages are received that 2 3 4, they are buffered. They do some kind of a protocol saying that all of us received a message, whoever is surviving. And once that is done then a will then at this point you deliver it. And B C D having got those things earlier on, they will also have saved it persistently. So, in case he has actually it has been delivered to a the processes A node 1. Then it is possible for us to make sure that node 2 also delivers it to processes B and node 3 gives to C, and all those who actually survive after this was delivered at this point; that means, there has to be lot of activity that has happened before you can deliver the message to a the processes A B C D.

That means first you have to send the messages to all the nodes, all of them agree that they have received a message all of them agree that, they can store it in a stable storage or some kind, right. And then once you one of them has delivered it other parties also can assure that it can be done. Otherwise it is not possible. So, in an non-dynamically uniform system you do not have to do all this things. Because what it is says whatever is remaining it is because we can do delivering for whatever parties system that remains. So, similar way if saying there are 2 types of what is called failure atomic multi-cast. One is a dynamically uniform, the other one is non-uniform. What is in the case of dynamic uniform? If it is a case that a suppose, you send the message m and if p delivers this message m; that means, it is guaranteed.

Because it is dynamically uniform that is any future execution, because there is some future after this, right. Because once p has delivered to m, there is something else going to come after that it is guaranteed that minute has delivered m it knows that in the future execution in which a set of process remain operational. There is some set of process that remain operational. It is a guarantee that that delivery of m among that set of process also is guaranteed. So, that is basically dynamically uniform protocols whereas, in a non-uniform protocol, if p knows that if both the sender of m and p crash, m may not reach it is other destinations. The fact that it has been delivered in p, but a p as well as the sender both crash.

It’s perfectly that the message m does not go to other guys. Whereas, here important thing is that the minute 1 plus in one at one node one message m has been delivered, then it is some of the system has to do something whatever it has to be done, to ensure that it goes to all the other remaining operational processes. That is difference there is some situations where this important, or some situation that is important. So, typically if there is some kind of a persistence state that has to be there, and has got extremely visible actions this is important, if there are no seriously what important extremely visible actions this may be. So, there are a few more definitions. There is a notion what is called reliable broadcast or m cast, sometime in some places we call it a cast instead of referring to broadcast or m cast, that is multi-cast or broadcast we referring it as in general as a cast, that is why we calling cast here.

(Refer Slide Time: 15:11)



More Definitions...

- reliable bcast/mcast: validity, agreement, integrity amongst correct processes
  - if a correct process casts msg *m*, all correct processes *eventually* deliver *m*
  - if a correct process delivers *m*, all correct processes *eventually* deliver *m*
  - for any *m*, every correct process delivers *m atmost* once & only if *m* cast
- uniformity: differs in agreement & integrity wrt correct/faulty processes:
  - if a process delivers *m*, all correct processes *eventually* deliver *m*
  - for any *m*, every process delivers *m* atmost once & only if *m* cast

Now what is we have to decide what is a correct, what is a correct process a correct process one which has not failed; that means, it is all we are now talking taking care of we are talking about what is called fails stop processes; that means, when they fail they stop they do not lie they do not do byzantine index. Say what is a correct processes? The correct processes one does not has a not failed. So, if I just say a process, it may mean it is possible that during whatever we are doing this way. Whereas, if I say it is a correct process it means that whatever I am talking about it does not fail during that time. So, what about I am saying here what we are saying is it is a reliable broadcast m cast.

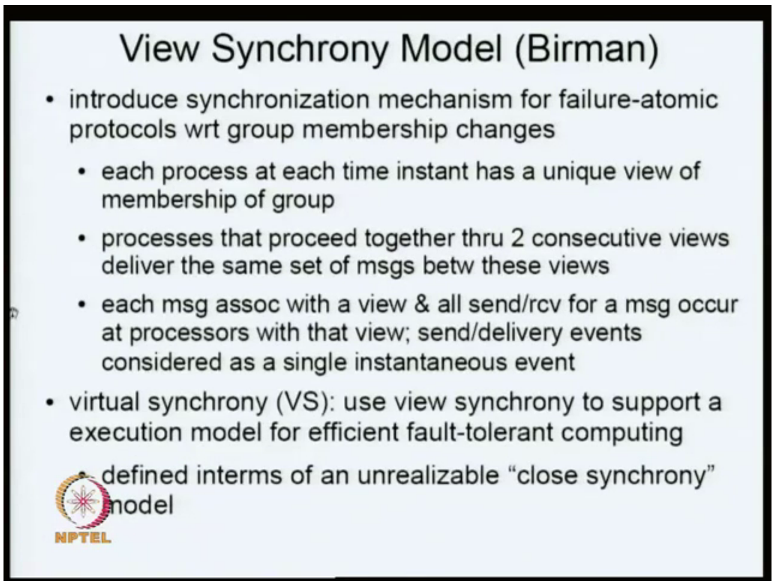
We are only saying about correct processes. That is, they which do not fail during this particular operation. The correct process casts message m, all correct process is eventually deliver m; that means, that none of this things the guy who has sent it or the guys who are received none of them failed during this particular process, during this particular cast if a correct process delivers m all correct processes eventually delivered. This is the guy who started it the guy who has one of them who has delivered. He is also a recipients. With one process basic p receives other process also received, for any message every correct process delivers a at most once. And only if m is cast ok again this at most once also not easy I think you remember that in NFS you cannot really do this.

That is why you need to have some logging going on there also this is a non-trivial, because in asymmetric if you have there is a message loss, asymmetric and it is possible that the act being go through, but the message went through. So, you can actually get more than one copy of the message; that means, at somebody on the receiving side, should keep track of it and drop the second one. So, this also a non-trivial. So, this is a case, with we are only discussing about correct processes. Just like in the case of dynamically uniform situation, you can also have a uniform model corresponding to this cast. So, this is differs in with respect to correct or faulty processes. So, they are talking now about faulty processes, again if you remember our previous discussion, commit protocols there is a consensus protocol, right that differ with respect to failures.

Commit protocol is irrespective of failures, you really have to ensure that those parties that are failed actually reboot again and do something. Whereas, in consensus protocols, we do not care about those processes that fail. We are only talking about correct processes there same thing here also. So, that is what this was the case whereas, in the case of uniformity you are saying you can see the difference here you say a processes delivers m. Whereas, here we say if a correct process; that means, that this process can deliver m and die. Whereas, in this case because the minute as I said the correct process delivers sends; that means, the process list through this particular broadcast or m cast, whereas, here the process can deliver m and then die.

So, if a one party succeeds in delivering it all correct process eventually deliver m. There is always guys who survive, right for after this guys delivered they actually deliver m. So, there is a small difference between this 2 things, and it turns out that there are if you are modeling all this things mathematically all this matter seriously. That is where there is gone a lot of controversy in some of this issues. Again, I will illustrate some other delivers.

(Refer Slide Time: 19:45)



View Synchrony Model (Birman)

- introduce synchronization mechanism for failure-atomic protocols wrt group membership changes
  - each process at each time instant has a unique view of membership of group
  - processes that proceed together thru 2 consecutive views deliver the same set of msgs betw these views
  - each msg assoc with a view & all send/rcv for a msg occur at processors with that view; send/delivery events considered as a single instantaneous event
- virtual synchrony (VS): use view synchrony to support a execution model for efficient fault-tolerant computing
  - defined interms of an unrealizable "close synchrony" model

So, given this kind of situations, we need a model by which you can program our systems. We will last time we will looked at some issues which creates complications, and they are trying to program this kind of systems. So, there is one model which is called a view synchrony model, I introduced it to you briefly last time and Ken Birman from Cornell University and his students they developed it first.

And there are other models at various other research groups, which come up with I am going to follow most of the Birman kind of model. So, in this particular talk in this particular lecture. So, what exactly are we doing in this case? Here the idea is that somehow the messages are sent, you have to ensure that the messages are sent and received in the same view. What is a view? View is some notion of what all the processes that are (Refer Time: 20:56) and any time a failure occurs. You are going to give that as separate event which ensures, that previous messages all are delivered by that time, and then this failure event occurs and then again messages are sent.

So, in the sense failure events somehow separate out message sent previously and after some kind of barrier. If you are able to do that then, programming of this kind of system becomes slightly easier. Whole idea is failure remains are some kind of barriers. Now it in principle it is not possible. Because while a message going on you can also here, right in messages in transit. So, what you have to do is to find the way in which you somehow delay the delivery of messages in such a way that you get this particular prospective.

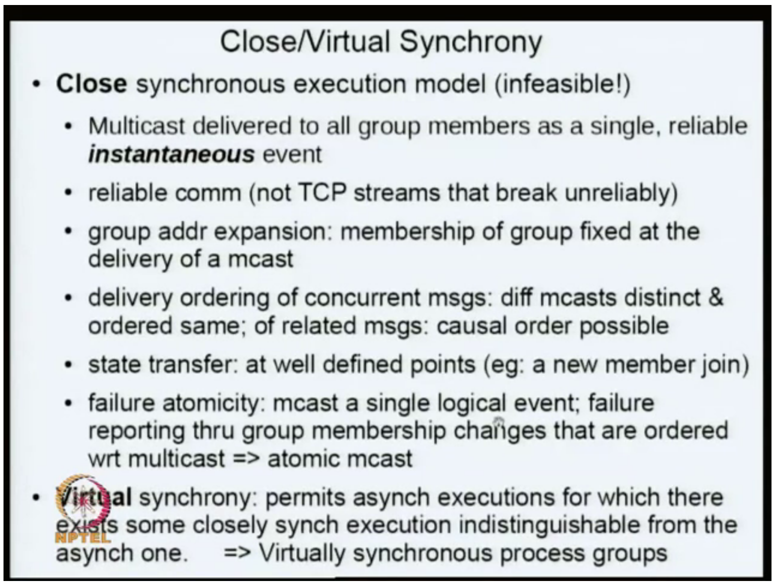
Real lie real systems are different, idea is to somehow move around things a bit. So, that is still have the notion by which messages sent in a particular view it is delivered in that view and then the failure occurs, and then messages have again start taking place it is that basically what it is.

So, what is the requirement? Each process at each time instant have a unique view of membership of a group; that means, when a failure occurs atomically somehow everybody knows that that fail party is no longer part of the group. This is sort of instantaneous of course, there is nothing instantaneous it has to manage somehow. So, that this kind of views possible, processes is proceed together to consecutive views. Deliver the same set of messages between these views again what is a view again I mentioned already, what it means it means a coherent idea of all the processes that are of a running for example, I know that for processor of a running they are p q r s, then lot of them dies then let us say r dies, when there is p q and s after this failure event everybody else in p q s knows that there are only 3 processes consecutive views.

And if processes that proceed through 2 consecutive views deliver the same set of messages between this views. And there is some ordering between this also, we will talk about that later. That also has to be possibly be followed. What is critical about this view synchrony model is each message associated with a view. So, each messages associated with a view. Every messages also as a view, and all send and receive for a message occur at processor with that view, only in that view only somehow you have fake it somehow. So, that happens very critical send and deliver events, considered as a single instantaneous event somehow you have to find a way how making (Refer Time: 23:52) we will come to this one basically, these are all abstract or impossible views to for it to happen, but we will go into take this try to come with a similar actual realistic model.

So, essentially what we are talking about is this part. All send and receive for a message occur at processor with that view. Virtual synchrony is the actual model, and this basically used uses this view synchronic kind of idea to support execution model so that it is easy to program. And this is defined in terms of a unrealizable close synchrony model which I am going to talk about next.

(Refer Slide Time: 24:37)



Close/Virtual Synchrony

- **Close** synchronous execution model (infeasible!)
  - Multicast delivered to all group members as a single, reliable ***instantaneous*** event
  - reliable comm (not TCP streams that break unreliably)
  - group addr expansion: membership of group fixed at the delivery of a mcast
  - delivery ordering of concurrent msgs: diff mcasts distinct & ordered same; of related msgs: causal order possible
  - state transfer: at well defined points (eg: a new member join)
  - failure atomicity: mcast a single logical event; failure reporting thru group membership changes that are ordered wrt multicast => atomic mcast
- **Virtual** synchrony: permits asynch executions for which there exists some closely synch execution indistinguishable from the asynch one. => Virtually synchronous process groups

What is this close synchrony model, and how is that used to define virtual synchrony, that is what we are going to look in to. What is a close synchrony execution of model? First, I should mention again repeat one more time this infeasible. Multicast delivered to all group members as a single reliable instantaneous event which of course, makes it impossible. That is reliable communication it is not like TCP streams that can break unreliably at different times as seen by different people. So, it is not like this way it is unreliable communication.

So, if the group expands membership of group fixed at the delivery of a m cast; that is, suppose I am trying to send the message where multicast the before I just about before as I send it I have a membership of group. I know exact who is; who all are part of the group delivery ordering of concurrent messages what are concurrent messages. There is no way in which you can ordering the messages there is there is only at the most of partial order amongst messages concurrent messages that there is no way to order them different multicast distinct and ordered same. So, it is there are 2 different multicast which are distinct, they are ordered in the same way of across all of them all the process. That is here with this is a delivering ordering related messages suppose there is some relation between messages.

That is, one is R there is reason why they are related one happens first it does something because of that because of this causal connected with the second one, right there is something else happens. So, this is a causal kind related messages then I have to ensure that the messages that are received that same causal order is possible. So, again we will go into this a greater detail later, but this is we are interested in a particular abstract model where either the messages are concurrent and therefore, they have more relations amongst themselves, but whatever the messages are there they will be received by all the parties in the same way. If they are related then I also have to make sure that, they are received in the same causal order the state transfer happens at well-defined points. For example, a new member comes in it is a new the group goes through a new state failure atomicity.

What we are going to say is that the m cast is a single logical event. Similarly, failure reporting through group membership changes that ordered with respect to multicast. Again, any failure that happens is some kind of a atomic event. So, we want to ensure that all this m cast or failure events they are seen as atomic events. So, that you can decide how to order them. Now this things definitely some other very difficult to achieve, but if you have something this kind it makes programming easier.

So, what is virtual synchrony it is basically a same as this, but you can permit a synchrony executions which has to be same as something similar to this, but and it has to be indistinguishable. I think it is similar to the way that we are talking about serializability on transactions, what are we talking about serializability transactions, they are actually concurrent, but what we say is to sets a transaction serializable in case there is this that 2 exist even if it is execute concurrently they have the same effect as some order some arbitrary ordering.

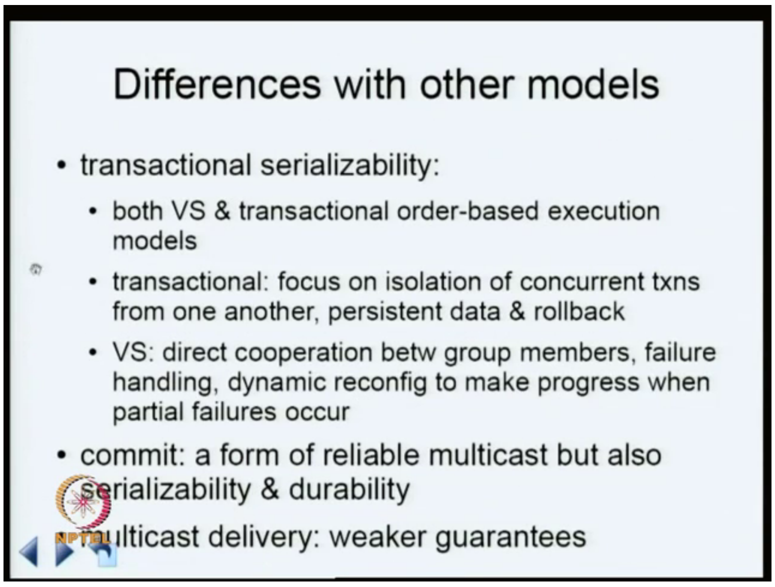
Which that 2 transactions are executed in serial manner, right, similar to this also we say a virtual synchrony is something which is look at a closely synchrony execution models of the various events are taking place. You are saying that a virtual synchrony, it is a virtual synchrony in case you can show that whatever way you reorder things. It is corresponding to one some close synchronous execution any one does not matter which one. So, again just to recapture rate if you look at serializability of transactions what are they doing there actually the transactions are concurrent. So, that actually reading and writing data possibly which are shared at the same time right.

And so, you are concurrent transaction manager will do something to ensure that locking etcetera, we take care of the problem, but suppose say that suppose you say that that 2 transactions. Now this serializable this 2 transactions are serializable in case you can order them as a followed by B transaction A followed by transaction B or transaction B followed by transaction A. So, the effect of the concurrent execution should be either A followed by B or B followed by A, that is all that matters. We do not say we do not specify it has to be a comma B, it does not matter a of B or B for a that is where it is serializability.

Here also same situation. I want a situation in such a way that I can reorder things in such a way that whatever you have reorder it should be corresponding to one close synchronous execution. So, because of this I do not I can drop the notion of instantaneous. As long as I am able to show that this virtual synchronous execution is can be same to be equivalent to some close synchronous execution lambda. So, permits a synchronous execution for which there exists some closely synchronous execution indistinguishable from the asynchronous one.

So, that is basically virtual synchrony, again those processing's process groups in which figures or messages or order that way they are called virtually synchronous process groups.

(Refer Slide Time: 31:18)



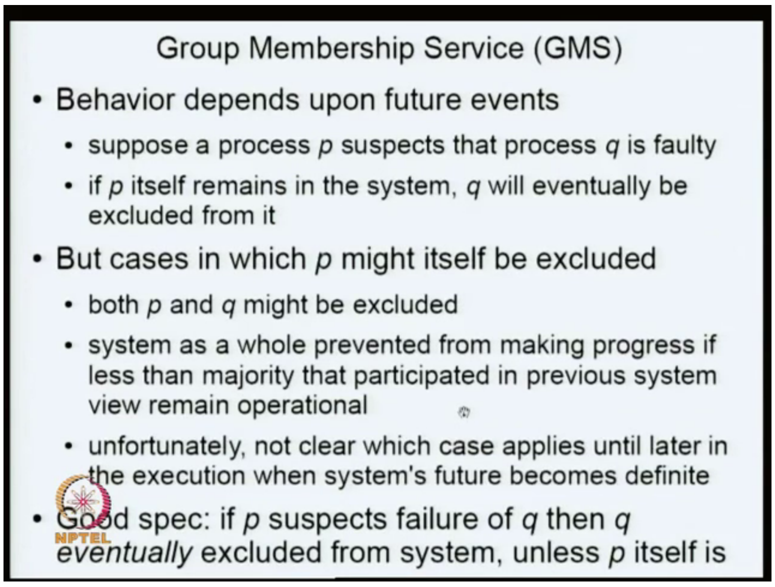
Differences with other models

- transactional serializability:
  - both VS & transactional order-based execution models
  - transactional: focus on isolation of concurrent txns from one another, persistent data & rollback
  - VS: direct cooperation betw group members, failure handling, dynamic reconfig to make progress when partial failures occur
- commit: a form of reliable multicast but also serializability & durability
- multicast delivery: weaker guarantees

Again, let us try to see some of the differences here. Now various types of models are possible. You have transactional serializability, but as I mentioned earlier in transactional it focus on isolation of concurrent transactions, and there is persistent data and rollback. The focus is on isolation of concurrent transitions, that is were the serializability comes into picture. Both virtual synchronous transactional order, right they basically order based execution. Because there is some kind of ordering you have to ensure. What is different about virtual synchrony is that there is direct cooperation between group members, failure handling and dynamic reconfiguration to make progress even when partial failures occur.

This part of it is not really part of serializability aspect, like is independent. Even commit for example, protocols we looked at it is some kind of a reliable multicast, we are not talking about concurrent transactions etcetera we are talking about agreement amongst the group there is no concurrency as perceive here perceive. This is different from this, but we are also just like transactional we are also concerned about durability. If you look at multicast durability let us say if you are talking about I p multicast. So, the guarantee some much weaker. Whereas, in the case of multicast the way we are talking about whether it is dynamic uniform or whether it is non-uniform, we are giving is slightly stronger guarantees. This I think I p cast kind of multicast for example, that is far less much less guarantee is given. Now we can define what is called as group membership service.

(Refer Slide Time: 33:03)



Group Membership Service (GMS)

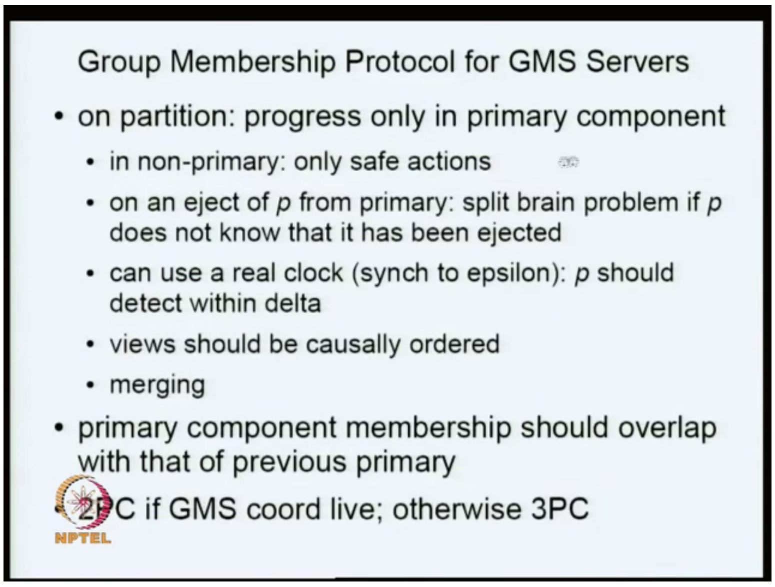
- Behavior depends upon future events
  - suppose a process *p* suspects that process *q* is faulty
  - if *p* itself remains in the system, *q* will eventually be excluded from it
- But cases in which *p* might itself be excluded
  - both *p* and *q* might be excluded
  - system as a whole prevented from making progress if less than majority that participated in previous system view remain operational
  - unfortunately, not clear which case applies until later in the execution when system's future becomes definite
- Good spec: if *p* suspects failure of *q* then *q* *eventually* excluded from system, unless *p* itself is

Now, let us some certainties here also, because one of the problems about the FLP result is that the pressure lengths patrician result is that you cannot figure out if somebody is actually extremely slow or dead right. So, at the most you can suspect somebody being not like. So, there are some complications across, this again we looked at the split-brain problem as well as the Mexican shootout that also will crop up here. So, the behavior actually depends on how the failure are supports at failures actually play them sorts out. Suppose a process p suspect that process q is faulty. Either you what you want you might want to think is it thing itself remains in the system q will eventually be excluded from it right p suspecting something. And a p itself is not faulty then if q is indeed faulty then q has to be taken out, but the problem for it is that p itself might be excluded. Again, we are talking about the Mexican shootout at same situation both p and q might be excluded.

Because each suspect the other as we discuss that time also, system as a whole prevented from making progress, if less than majority that participated in previous system view remain operational. Because sometime what happens is that some forum is not established therefore, you cannot really make any progress. Basically, you really cannot decide what to do in this kinds of situation, because quorum is not there you just have to (Refer Time: 35:02) along till something happens. So, that again quorum is established right. So, instead of this particular type of description, people have again people are try to model it carefully they have decided that instead of this particular model if p remains this way if p suspects failure of q, then q eventually excluded system unless p itself is. So, this comes out previous slightly better spec than this one.

So, various theoretical models are come up with they are try to use this kind of models. As I mentioned this area is full of complications, and lot of different theoretical models they come up there you make mild change this here and the extent you might call it, and different kinds of things will come up from that.

(Refer Slide Time: 35:57)



Group Membership Protocol for GMS Servers

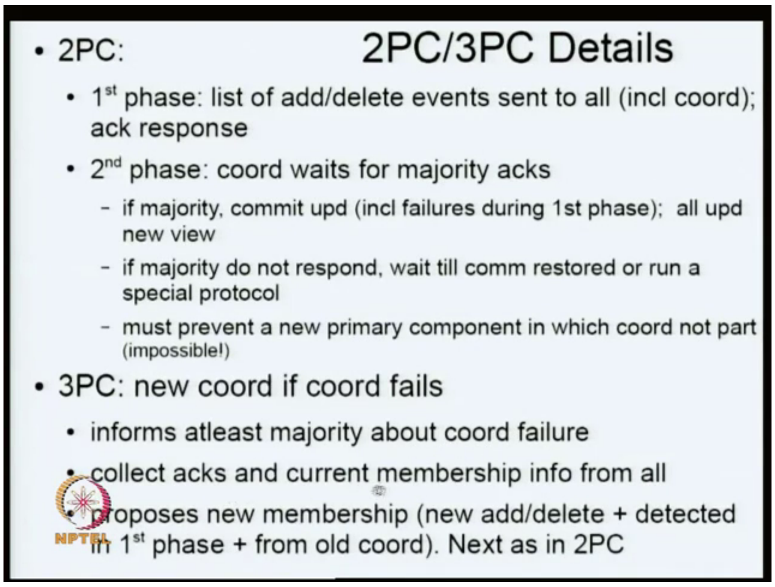
- on partition: progress only in primary component
  - in non-primary: only safe actions
  - on an eject of *p* from primary: split brain problem if *p* does not know that it has been ejected
  - can use a real clock (synch to epsilon): *p* should detect within delta
  - views should be causally ordered
  - merging
- primary component membership should overlap with that of previous primary
- 2PC if GMS coord live; otherwise 3PC

Let us just look at some of the issues in spite, basically our problem is partitions right. If there is a partition, what I would like to ensure in this protocols, progress should happen only in primary component. Suppose you are air traffic control system, various parties are trying to get the aircraft into reasonable situations, part of the system one part of it gets disconnected. So, what you would like to do is, that they should progress only in the primary component. They should not be inconsistent actions taken in 2 different partitions.

Because that can be extremely risky. So, what we are saying is that somehow, we have to decide which is a primary component. So, in non-primary at the most you can do safe actions. It means it does not update anything. At the most it can read things it can it can only do what is called safe actions. Suppose there is a process p which ejected from primary. Then as I discussed earlier, you can have what is called as split-brain problem, if p does not know. That it has been ejected, because it does not know it is ejected can go ahead, and start updating it instead of indulging only safe actions it can start updating things and can get a problems. There is also the views should be causally ordered. Again, we will talk about this later, if the partition heals you may want to merge in 2 partitions. Again, this is a complicated thing how to merge partitions with they will. And this something which also many people have faced.

If you have this called you know develops software, you have various control version systems version systems. And when you might also have the merging operation is there, same similar issue also cut of here also. You have to make this state of the system consistent there has to be merging operation also. One critical thing in this kind of systems is that, again we discuss also in context of paxos, if there is a new primary component because of some failure, there should be some overlap with that of the previous one. This is also critical. Now you will find that you need something like a 2-phase commit or a 3 phase commit protocols to make all this happen.

(Refer Slide Time: 38:41)



2PC/3PC Details

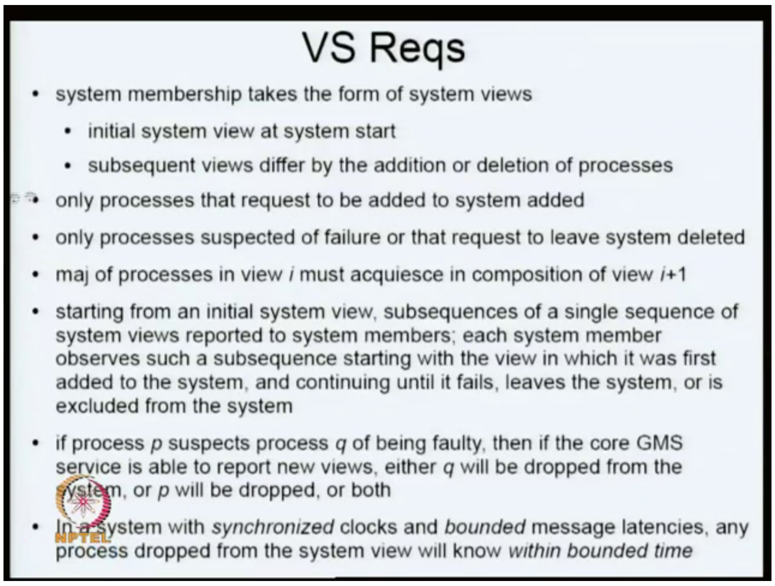
- 2PC:
  - 1st phase: list of add/delete events sent to all (incl coord); ack response
  - 2nd phase: coord waits for majority acks
    - if majority, commit upd (incl failures during 1st phase); all upd new view
    - if majority do not respond, wait till comm restored or run a special protocol
    - must prevent a new primary component in which coord not part (impossible!)
- 3PC: new coord if coord fails
  - informs atleast majority about coord failure
  - collect acks and current membership info from all
  - proposes new membership (new add/delete + detected in 1st phase + from old coord). Next as in 2PC

So, I just go and do some detail about that one. So, what is the basically we are trying to do membership, try to figure out who is there who is not there. Because some failures are taken place I am going to try the figure out who is out there. In the first phase list of add delete events sent to all including coordinator, and the acknowledgements or are anybody who gets it should ack. The second phase the coordinator waits for majority of acks. If majority acks you get back you commit updates including failures that happen in the first phase. Again, you can see this is also introduce even complications, but we will assume that some of this can be handled. And all update new view a measured do not respond wait till communicate restored or run a special protocol. Must prevent a new primary component in which coordination not part.

Because this thing principles impossible, but you should try to ensure that the new primary component, if it is not part of the if the coordination not part of the new primary component, then it creates of an complications. So, it is better to prevent it, I am not going in to detail about this point. So, in 3 phase commit, if you are going to have a new coordinator there is an election here if coordinator fails, and you informs at least majority about coordinator failure. This is a new coordinator informs, that there has been a coordinator failure, collects acknowledgements and current membership info from all proposes new membership, doing the whatever new events are taking add or delete elements. Again, it goes back to essentially can again restart from 2 phase commit.

So, basically our problem is either the majority may not respond, or the coordinator might fail. If the coordinator fails then you get blocked, right. This like in a previous commit situation. If you want avoid the 3-phase commit blocking, you have to elect a new coordinator, and basically for the same reasons we discussed previously. You again have to propose a new membership, and then you have to restart the protocol again, and see if you can get agreement you can. So, we can go back and forbidden 2 phase, and the way Birman has decide describe it basically combines 2 phase commit, and 3 phase commit. So, you get some interesting optimizations.

(Refer Slide Time: 42:14)



VS Reqs

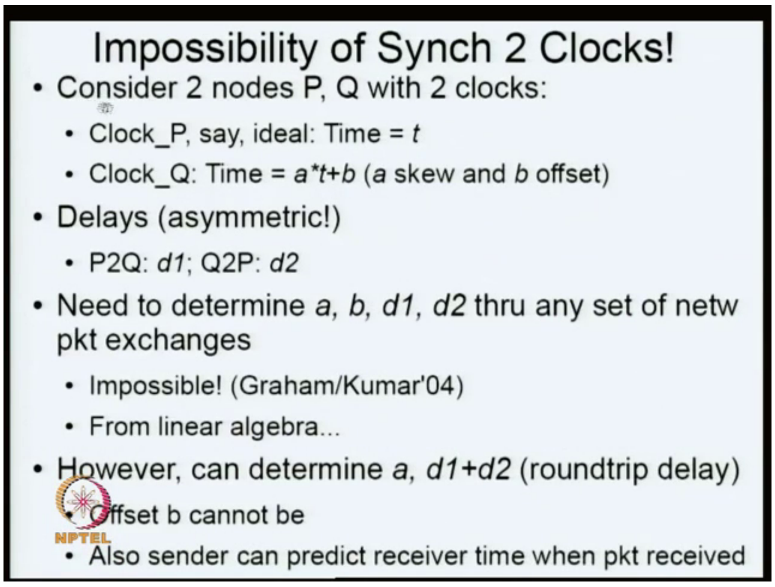
- system membership takes the form of system views
  - initial system view at system start
  - subsequent views differ by the addition or deletion of processes
- only processes that request to be added to system added
- only processes suspected of failure or that request to leave system deleted
- maj of processes in view *i* must acquiesce in composition of view *i*+1
- starting from an initial system view, subsequences of a single sequence of system views reported to system members; each system member observes such a subsequence starting with the view in which it was first added to the system, and continuing until it fails, leaves the system, or is excluded from the system
- if process *p* suspects process *q* of being faulty, then if the core GMS service is able to report new views, either *q* will be dropped from the system, or *p* will be dropped, or both
- In a system with *synchronized* clocks and *bounded* message latencies, any process dropped from the system view will know *within bounded time*

So, what are the various requirements in the; I think let me summarize requirements, here for virtual synchrony all system views, right. (Refer Time: 42:30) view is basically some idea about exactly who are living responding. Initial system view at systems start subsequent views differ by the addition or deletion. Again, these are all asymmetric things which are needed for mathematical modeling, only processes that request to be added to system are added, only processes suspected of failure or that request to leave system deleted, majority of processes in view I must acquiesce in composition of view i plus 1. So, some sort of quorum is there without quorum.

We would not do starting from initial view system view, subsequences of a single sequence of system views reported to system members. Each system member observes a subsequence starting with the view in which it was first added to system, and continuing until it fails leaves. The system or is excluded from the system. The process p suspects process q of being faulty, then the core service met basically the group and membership. Service is able to report new views either q will be dropped from the system or p will be dropped or both. So, basically idea is that this particular is lie again this another condition. That core GMS service is able to report new views. Because of our FLP kind of results, it is this is also not possible to guarantee in all, that is why this is a f here. There is a core GMS is able to report new views, another q will drop from this system or p will be dropped or both. And the system with synchronized clocks and bounded message latencies, any process dropped from the system view will know within bounded time.

Again, you will see always within bounded time etcetera. Again, all the while since this start talking about NFS and consistency problems, you notice that almost all the important guarantees are given in terms of some eventual that eventually it will happen. And here we are talking about a same thing that is synchronized clock. That is why we are able to say within bounded. If you do not have synchronized clocks we cannot give any other guarantees, and actually it turns out synchronized in synchronizing clock itself is a very hard problem. I think we will discussed it mentioned it briefly, that I will mention this slightly more detail in the next slide. This actually is a interesting problem, that in many common situations synchronized clocks are actually impossible.

(Refer Slide Time: 45:31)



Impossibility of Synch 2 Clocks!

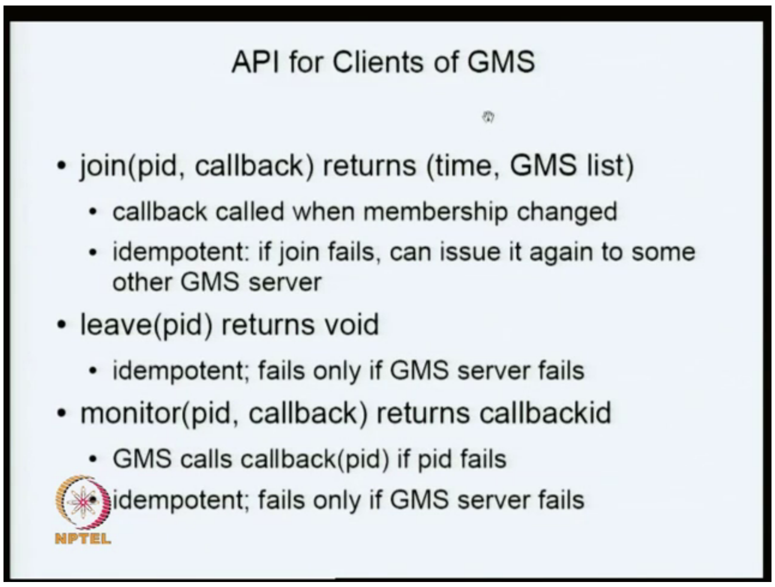
- Consider 2 nodes P, Q with 2 clocks:
  - Clock_P, say, ideal: Time = *t*
  - Clock_Q: Time = *a*t+b* (*a* skew and *b* offset)
- Delays (asymmetric!)
  - P2Q: *d1*; Q2P: *d2*
- Need to determine *a, b, d1, d2* thru any set of netw pkt exchanges
  - Impossible! (Graham/Kumar'04)
  - From linear algebra...
- However, can determine *a*, *d1+d2* (roundtrip delay)
  - Offset b cannot be
  - Also sender can predict receiver time when pkt received

Let us just look at how it is. Suppose you have 2 nodes p and q, there are 2 clocks. Let us say that clock P is ideal. So, the time is equal to t, and clock Q is a one which is slightly off. So, it is actually see is the idea this is actually 2 time. So, means basically a times t plus b. A is the skew, and b is a offset. Instead of going at 45 degrees, it is going in a slightly different angle, it was at offset. Now suppose I make the critical assumption delays asymmetric; that is, from P2Q the delay is d 1 from Q2P is d 2. Now what you have to decide? This basically come on to synchronized clocks you have to decide what a b and d 1 d 2 are. So, you do any set of network packet exchanges has been shown that impossible to do.

So, basically comes out if you drop the diagrams, and write this equations corresponding to when something was sent and something were received from other side, and reason about it you can actually gets some kind of matrices, and the matrices are turns out for determining 4 of them, you should have a rank for matrix it turns out in the rank 3. Therefore, it is not possible to get this values, but what is interesting thing is that if you either drop this in asymmetric situation, or if you are interested only in roundtrip delay then it is feasible to determine in this model. So, when you talk about this this within bounded time also is slightly problematic, because the principle this assume a synchronized clock is (Refer Time: 47:40) on. This actually not a feasible system in many situation, because the delays are often asymmetric. Why it is asymmetric? Because you could be between set on either bridges or routers and various things, and it may be that the paths you take are asymmetric.

So, it is not really guaranteed that both are asymmetric. And it is even more say suppose you are in the wireless situation, it turns out that your message going one direction depends on the condition of the channel radio channel. So, therefore, d 1 and d 2 could quite different it depends on when you send it. So, all this things are in practice quite reason of the fact that delays asymmetric. Therefore, you should that is a case, then what you are seeing here is actually also problematic; that means, that is typically this also controversy.

(Refer Slide Time: 48:42).



API for Clients of GMS

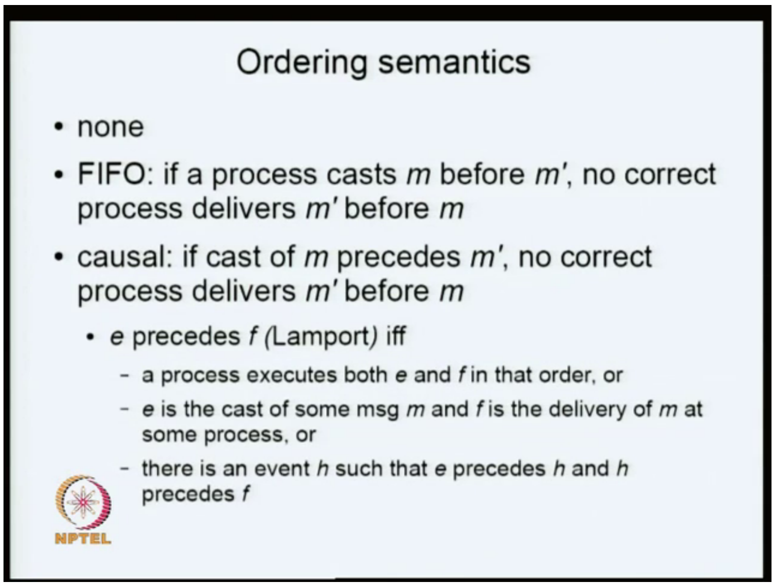
- join(pid, callback) returns (time, GMS list)
  - callback called when membership changed
  - idempotent: if join fails, can issue it again to some other GMS server
- leave(pid) returns void
  - idempotent; fails only if GMS server fails
- monitor(pid, callback) returns callbackid
  - GMS calls callback(pid) if pid fails
  - idempotent; fails only if GMS server fails

So, that is a desirable set of things. So, asymmetric things are not easy, but let us proceed all this thing. Because our idea is to come up with the engineering solutions. So, for that we can you can if you have a group membership service. You can have the following kind of API, what is it do you can say join pid callback, it returns basically the time you joined and the list of members.

So, callback is called when the membership changed for example, if p ID joined. So, it is some new they are comes in is informed to update it is used that for the new guys come. In what is interesting thing this is idempotent. If join fails, can issue it again to some other GMS server. Again, what you are assuming is there is not a this the group membership service is actually it has to be highly excellent service in itself right. So, this is itself have multiple modes, and if one of them fails somebody has to overtake out. So, what I am saying is that I in case I make this con and this I get a that response saying that it is not ok, you can again reissue it.

Because I can put it same thing with leave, I can keep trying to leave till somebody says yes you can do. What is monitor? Again, you want to find out if a particular pid something happens to it. Some basically saying I am basically saying this this is a process I am interested in, and then please monitor it this telling the service. And so, in case something happens to this, if pid fails, then GMS itself calls back, this callback on using the pid as the parameter. Again, this was also has to be design as idempotent. So, that in spite of whatever is happens this kind of took it one more time. And there is some other because we are talking about there is a code set of services nodes which are actually giving you the group membership service, and that itself has to be replicated, and none of them fails somebody else is therefore, look after it.

(Refer Slide Time: 51:25)



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*

So, there if it is idempotent, you have matter if something fails you can just repeat it. I will briefly introduce ordering semantics today, that too in the next class, we will going to inherit the detail. Now so far you talked about how to handle failures. How to ensure that a failure looks like atomic. So, that I can a something a barrier, the all the messages before and all the messages afterwards. That is what we are try to do. Now the questions about selling the message itself. And other have know semantic what isoever; let it happen whatever they might happens. It can have happens at FIFO model, a process casts m before m prime that is a same process. I am sending, I send m before m prime the no correct process delivers m prime before m; that means, everybody also get m first followed by m prime; that is, because I am able to order the things at my side everyone also should be order it at that side also with same way.

This was same with causal. If cast of m precedes m prime, no correct process delivers m prime before m. Now this process itself there is a technical definition for it. What is a technical definition? Basically, we do it in terms of events, and that events basically can be used to define the precedes relation with respect to messages. So, what is this this I can lamport has delivered has define this particular model, what exactly is this very familiar model, that process executes both e and f in that order, that is e is I execute e first followed by f, but I in my system in my node at I execute e first followed by f therefore, e before f that is very non-controversial, right. It is not problem, because I am doing in a same node I do e then I do f therefore, you should e for f suppose I it is across a nodes.

Now, these are definition with respect to across nodes. What is it? E is a cast of some message m, and f is the delivery of m at some process; that means, I sent a message e is a thing which has initiating broadcast or multicast at my node or my node for example, and f is a delivery of m at some process. Then I say that e proceeds f that is sending has to proceed a delivery receiving. Very simple model, or there is an event h such that e precedes h and h precedes f, but is I do something, there is somebody else who follows in the precedence relationship with respect to a one of these or recursively, where something which comes after that. And that itself is before f therefore, e before f that is the reason that is what we call saying that if an m cast or a broadcast m precedes m prime, the definition is this. In case this is not this particular precedes event apply they are concurrent. There is no way to order them; that means, there is no way require ever doing causally connected to other therefore, they are actually concurrent.

So, this is the order. So, basically causal what it says is that, if in some way you can show that m precedes m prime in the causal definition in this using this order this one, you have to guarantee that no correct precedes delivers m prime before m. So, the other models also I will continuing with I just want to introduce the kind of ordering that also has to be looked into as part of this whole definition, and we will go further detail in the ordering part of it in next class. And hopefully conclude it with respect to all the required things with respect to failures as well as messaging ordering in next class.

Thank you.