

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

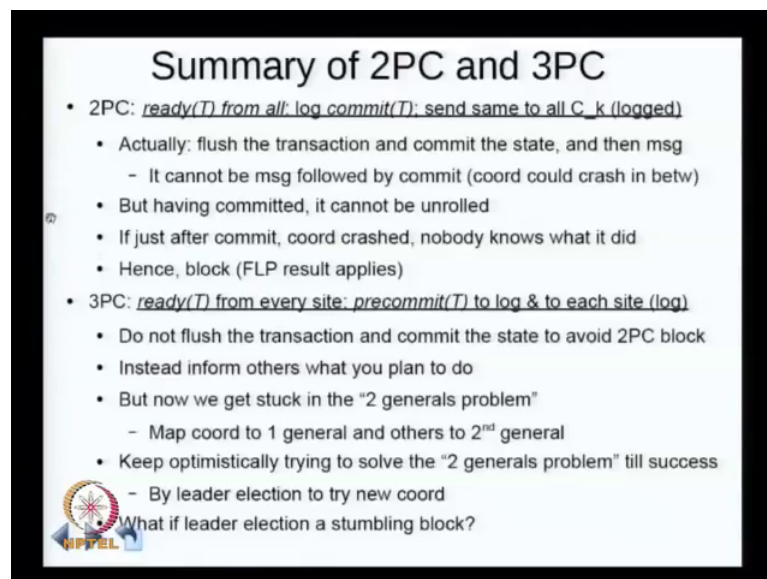
Theoretical foundations

Lecture – 33

Theoretical foundations of Distributed Storage systems _ Part 2: Commit Protocols continued, Paxos Algorithm for the Consensus problem


Welcome again to the NPTEL course on storage systems. In the previous class, we started looking at some problems relating to commit protocols, and we looked at 2 phase commit and 3 phase commit. Again to refresh a memory we had also try to understand the difference between commit protocols verses consensus protocols. So, we started looking at commit protocols. So, I will just briefly try to summarize 2 phase and 3 phase, especially as why they have the problem that we discussed.

(Refer Slide Time: 01:08)



Summary of 2PC and 3PC

- 2PC: *ready(T) from all; log commit(T); send same to all C_k (logged)*
 - Actually: flush the transaction and commit the state, and then msg
 - It cannot be msg followed by commit (coord could crash in betw)
 - But having committed, it cannot be unrolled
 - If just after commit, coord crashed, nobody knows what it did
 - Hence, block (FLP result applies)
- 3PC: *ready(T) from every site; precommit(T) to log & to each site (log)*
 - Do not flush the transaction and commit the state to avoid 2PC block
 - Instead inform others what you plan to do
 - But now we get stuck in the "2 generals problem"
 - Map coord to 1 general and others to 2nd general
 - Keep optimistically trying to solve the "2 generals problem" till success
 - By leader election to try new coord
 - What if leader election a stumbling block?



If you look at 2 phase, we mention that it gets into a blocking situation. Why is it happening? We just quickly look at it one more time.

(Refer Slide Time: 01:25)

2-phase commit

Transaction T initiated at site S_i and txn coord there C_i .

When subT completed at all sites (all sites inform C_i), start 2PC protocol

- **Phase 1:** C_i logs $prepare(T)$; sends $prepare(T)$ to all C_k
 C_k : on receiving prepare msg, either
does not commit: logs $no(T)$ and sends $abort(T)$ to C_i
commits: logs $ready(T)$ with all changes to log onto stable storage and sends $ready(T)$ to C_i
- **Phase 2:** C_i receives response to prepare msg from all C_k or timeout:
ready(T) from all: log $commit(T)$; send same to all C_k (logged)
else: log $abort(T)$; send same to all C_k (logged)
each C_k sends $ack(T)$
 C_i receives acks from all: logs $complete(T)$
 $ready(T)$ from a site: will follow coord's order to commit

NPTEL

So, this look at 2 phase commit, you will see that after this point it is all preparation. All this while we just preparing to do something; the only time when we actually do, when you really make some movement, least at this point.

See when we get ready from all, we have we decide that. Now I can actually make the move; that is previously that sort of uncertain not clear what is going to happen, but now it is all clear, I am going to make a move. So, this is a critical point in some case. Now similarly when you look at 3 phase commit.

(Refer Slide Time: 02:07)

3-phase commit

To avoid blocking in limited cases:

- eg. no netw partition; atmost K sites can fail & atleast K+1 sites up

Provide preliminary info about fate of T thru a precommit phase

Assume failures detected by coord or sites reliably

- **Phase1:** same as 2PC
- **Phase2:** C_i receives responses to prepare msg from all C_k or timeout:
any site $abort(T)$ or no response from a site until timeout: $abort(T)$ to all
ready(T) from every site: $precommit(T)$ to log & to each site to its log
ack sent to coord from each site whether abort or precommit (logged)
- **Phase3:** only executed if precommit in Phase2
coord waits till atleast K acks: logs $commit(T)$ & sends it to each site to its log
ready(T): site's promise to follow coord's decision
precommit(T): coord's promise to commit

NPTEL

You try to avoid that situation that is why you get into a pre commit situation. In the 2 phase commit we actually do this flushing of your, the coordinator flushes his state, unless no need to go back. Here what he has done is, he decides. I cannot really do it right away, because I am get into a blocking situation. So, he say, I am going to see how this situation develops, how I am going to do it. I am going to tell everybody what I am going to do. Again I will look at one more time in a slide different way. So, that we see the difference between the 2 situations ok.

So, I mentioned before in 2 phase commit, we get ready t from all, then you commit. Again just let us understand what commitments. Again as I mentioned 2 phase commit and 3 phase commit are connected with those kinds of systems, for which there is persistent state. These are distributed persistence or distributed a basis or etcetera. So, in each party has done something they have some state, that has to be flush to disc. So, this actually means, what exactly are meaning by this. They flush the transaction and commit the state, and then only we send the message. As I mentioned before, you cannot send the message first followed by commit; otherwise in the coordinator could crash in between.

And you already inform that you are doing it, but you are unable to flush it to disc, your state; that is why it always has to be flush the transaction; that is to make it is stable on a stable storage, and commit a state. You made a irrevocable step you taken, then only you send the message. So, the other part is know, what you, that all you did something. Other part is better follow what I am going, what I am doing. So, basic issue is that, if I have committed it cannot be unrolled, because all the, let us say buffers etcetera have been taken back etcetera. So, there is no way to go back, they have committed. You cannot be delivering too much. If you keep doing it, you get into a. At some point you have to etch a make a transition. So, this is the place where it is happens. So, our problem is that having committed, before I send the message I can also still crash.

So, because that happen nobody really knows what the coordinated it; that is why get into the blocked ratio. I hope this part of it is clear. And basically next thing is from your f l p result. Remember the result which I mentioned about the Fischer Lynch Paterson result. What is it say, in a distribute system consensus is not possible, even its one cortex processor; that is a drastic statement of the theorem. And basically this is what exactly that is why the f l p result applies here. So, you have to commit then only you can send the message, but it is possible that after having sent them committed yourself nobody

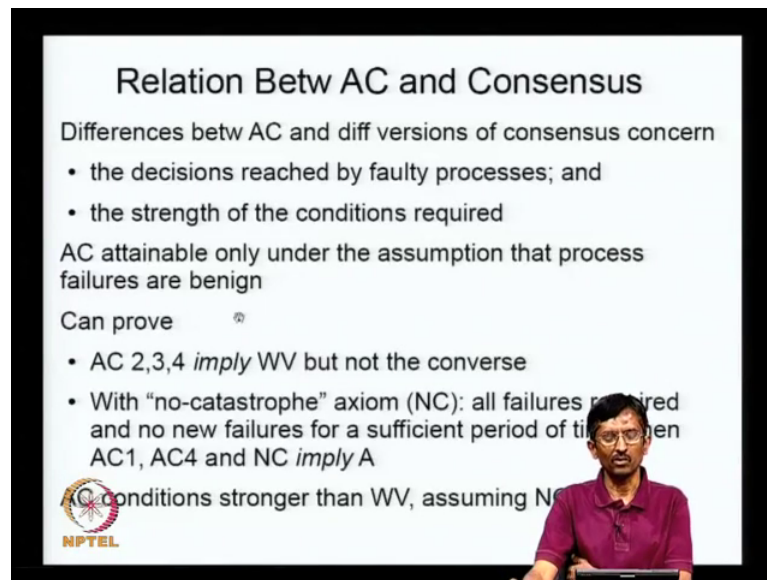
knows what he did; that is why you have the flip results of applying in this case. Now what is situated with this 3 phase commit? Here I do not want to get into situation here. So, what I want to do is. I want to sort of tell myself. I do not want to flush the transaction and commit the state to avoid 2 phase commit block ok.

I want to avoid this one. So, what I am going to do. I am instead inform others and what I plan to do, a coordinator is slightly wanting to, not taken any chances or getting block. So, it is trying to inform us. So, now, what happens is that if you take this particular stance, unfortunately gets stuck into the 2 general scan of problem. What is this problem, as you as I mentioned before, what is this particular problem. There are 2 generals, in between them there is an enemy. And both the generals have to coordinate to attack then at the same time, otherwise they will lose. Now our problem is that any messages etcetera that have to be sent, have to go through enemy territory, and the messages can be lost etcetera, and there is no solutions to this problem. Now we can see why this is the case, because now we can map the coordinator to one general, and all next others to second general. Now because of this you are not flush your state.

And we are trying to make the 2 parties agree, before the coordinator flushes his state. Essentially you have stuck into this 2 general problem you have. There is no way to escape from this. So, the only escape we have is one engineering point of view, we can say a optimistically try to solve the 2 generals problem, till someone succeed, send the since I am not lie. I am correct, but I am not lie. Here I am blocking. Here also I am consistent, but I block. Here I instead of blocking I keep trying things. I am at you might say I am in a I am blocked here, but I am here could be in a live block situation. Keep on trying new things, things do not work out and I keep on trying. So, the way I can do it is by having a leader election, to try new coordinators, I keep trying it will someday; I succeed ok.

Of course from this you can see there are 2 problems; one is that the leader election is stumbling block, because if you look at it what is the leader election all about, it is all about trying to figure out who is the leader. Again it is since it is since is similar to our commit problem. Only thing is that you may has slightly lesser stringent conditions for this part. So, again we will notice one interesting thing about 3 phase commit, as I mentioned earlier.

(Refer Slide Time: 08:22)



The slide is titled "Relation Betw AC and Consensus". It contains the following text:

Differences betw AC and diff versions of consensus concern

- the decisions reached by faulty processes; and
- the strength of the conditions required

AC attainable only under the assumption that process failures are benign

Can prove

- AC 2,3,4 *imply* WV but not the converse
- With "no-catastrophe" axiom (NC): all failures repaired and no new failures for a sufficient period of time then AC1, AC4 and NC *imply* A

AC conditions stronger than WV, assuming NC

The NPTEL logo is visible in the bottom left corner of the slide. A presenter is visible in the bottom right corner of the slide frame.

When we are talking about the difference between atomic commit and consensus. We also mentioned the following; atomic commits attainable only under assumption that the process failures are benign. Basically because every party in the proto coordinate they have some state that has to be carried forward in case they commit.

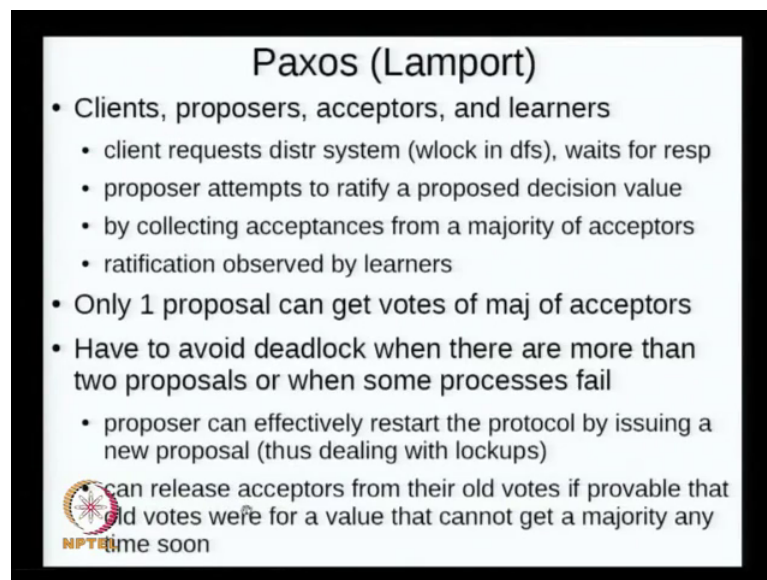
If you have any, does not end failures; that means, that you really cannot do this atomic commit, just not possible. So, there are situations where when I am talking about things like; leader election right. Here when I am doing it taking leader election, I do not really necessarily have some kind of state that has to be kept forward. I do not have to save it. For example, I do not have to flush this kind of state then I am doing leader election. So, in one can argue that this is slightly simpler than atomic commit, because I do not have some state, persistence state that has to be made persistent, as part of its protocol in case I agree on something. So, in a sense you can see the atomic commit; finally, you can devolve to the consistency protocol as a sub protocol inside it. And this kind of stuff is also important in other context. For example, if I am a, there is a storage system.

And I need to keep a replicas, multiple replicas, and there are 2 types of there are 2 messages I send, and I have to send it to 2 sets of replicas, and it is important for me that all the messages are send; a sets of messages has send they are receive the same order in which I have sent, to each are the replicas sites. So, here also there is some problem relating to, how to make sure that all of them agree that all the message have been

received, and the same order it was sent out. So, these things do not have a persistence state as part of the problem phase solved. In case it does not happen, you can restart again, you can retry one more time it is not a problem. So, it turns out consensus problem has a slightly simpler aspect to it, if things fail you can retry ok.


But in the case of atomic commit kind of situations, you really have to flush the state that has been accumulated as part of your, whatever transaction you are doing. So, it is a slightly different problem. So, what we will do is, now we will start looking at instead of looking at, a kind of commit protocols for which persistence state is so important. We will try to look at closer to consistency kind of problems.

(Refer Slide Time: 11:19)



Paxos (Lamport)

- Clients, proposers, acceptors, and learners
 - client requests distributed system (wlock in dfs), waits for response
 - proposer attempts to ratify a proposed decision value
 - by collecting acceptances from a majority of acceptors
 - ratification observed by learners
- Only 1 proposal can get votes of majority of acceptors
- Have to avoid deadlock when there are more than two proposals or when some processes fail
 - proposer can effectively restart the protocol by issuing a new proposal (thus dealing with lockups)

 can release acceptors from their old votes if provable that old votes were for a value that cannot get a majority any time soon

And one important such solution is, called Paxos. So, you will try to look at this particular problem, and this was solved by famous computer scientist Leslie Lamport. It was also solved the other parties previous to this, but here a very colorful way of talking about it. So, I think distributed Paxos etcetera have stuck, but 2 or 3 other researches also have come with single solutions before.

So, let us look at what this particular algorithm is about. You have clients, proposers, acceptors and learners. Clients are those which are waiting for some. For example some resource which is shared in multiple parties. For example, somebody wants a right wlock in a distributed file system. Since is a right wlock only one party should be allowed to be given this permission. So, clients of those who request, this kind of requests. And we do

not care who actually gets the request, as long as its only one person guess. Again this is example of mutual exclusion and. So, clients make request some mutual exclusion, and they wait for response. Then we have a set of proposers, who attempt to figure out who should be given access to the wlock for example,. So, that the party who should get it is the value for example, if 5 people are waiting for it you can say number 5 or number 2. ok.

So, proposers are basically the parties there will be a few of the proposers, essentially the reason why a multiple proposers is that. In case there is a failure of proposers. The other proposers who know that this particular request has been made, you have proposers. So, that you remember that request has been made, even if some other proposers die, and they try to figure out what value should be decided upon. So, there are 5 people waiting to get a light right wlock, you should say one of them gets it, we do not care who, but one of them should get it and by collecting acceptances from a majority of acceptors ok.

So, proposers are saying what value should be proposed, and there will be a majority of. There will be acceptors, who will essentially access, its fine to go with 4 5 whatever. And then once it has been accepted, it is possible that there other failures in system, and you cannot remember what really happened, and that is basically rectified by learners. And the learners are basically the party who are some kind of replicated information somehow, so, that the clients can get the information. So, it is a more, like very general model. Often time many of this things are collocated, but for the proposers service thing we can assume, that are various types of inter stating system, clients makes request, proposers solve. Remember what kind of request have been made.

So, that even if there is only single proposer, it might you can dies it is a problems. So, that is why there are multiple proposers. Acceptors are the parties who actually have some state, and they will try to remember what happened in the past also. Proposers will not have any state; persistence state, but acceptors will have. So, in case something has been decided in the past, they can look up the state and say yes, this particular thing was asked from bad stuff happened, but I still know what happened in the past, I can provide that that is why this guys are state; persistence state. And learners are the parties who actually can, this some kind of replication of that information about what of actually agreed upon. So, only one proposal can get votes of majority the acceptors. So,

essentially there is some kind of quorum here. Here I am talking of majority, but it can be some type of quorum ok.

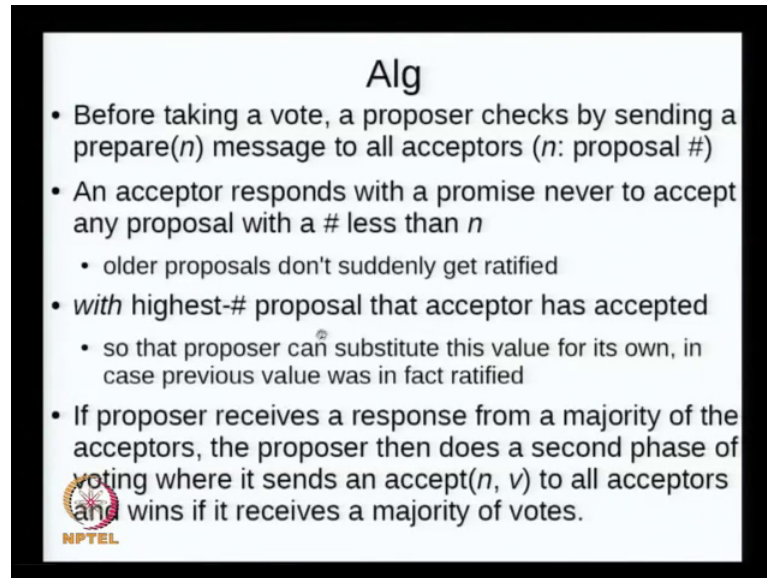
Some of problem is that, if you look at it; what are the problem in the previous case. What you are trying to do was, to do a leader election. Suppose it is a leader election is stumbling block. So, instead of doing leader election, I basically say let anybody start it, does not matter who I do not try to go through leader election algorithm, but say I will say anybody who sees that something is stuck, he can proceed, he is not going to do a leader election; that is a basic thing that Paxos does. So, essentially what you have to do is, when there are more than multiple parties who suddenly say that I see some problem, let me start a new round, or saying that let us the value be at this particular, I am going to restart the protocols so that any value can be decided ok.

So, if there are multiple proposals, they turns out that you can wait for each other, and therefore, there can be some situations simulated deadlock as I say. So, to break those things, you can, proposer can effectively restart a protocol, by if you see that the previous proposals, previous attempts to negotiate a value, were not making headway than you can just drop them and restart it. And if in the past some acceptors have decided some values, and those values cannot get the majority. If it is, if really clear that those values will not get majority, then you can ask the acceptors to flush their state, and to start a new round, and then try to see that they can come to a new consensus again. So, this is a high level idea about what is being done. Essentially it is slightly different from Paxos. Sorry slightly different from 3 phase commit protocols in multiple ways; one is that, commit protocols, there is an issue about state that has to be flushed, persistence state ok.

Which is critical as part as part of the, let us say as part of the protocol. Here we need to have persistence state only to decide upon the value, but we do not care about what value we actually pick up. In case something does not work out, I am willing to go with a new value, it does not matter. Whereas, in the commit protocols I cannot do that for example, if a hotel has been booked, and we are trying to proceed that what hotel are booked is important, I cannot just like that drop it. Unless there is no agreement at all then we have to drop it. If somebody try to [mo/move] move forward and they already said that this particular hotel is important, you cannot drop it. So, it is some difference in the 2 types of problems. So, here what we are doing is, we can have without going through leader election, we try to restart the protocol again.


And one way to do it, is by making sure that, if any intermediate if in the past some people have decided on some particular values, if you can show that, those values cannot be accepted by majority, then it is easy to drop them there, is basically the idea um.

(Refer Slide Time: 19:21)



Alg

- Before taking a vote, a proposer checks by sending a $\text{prepare}(n)$ message to all acceptors (n : proposal #)
- An acceptor responds with a promise never to accept any proposal with a # less than n
 - older proposals don't suddenly get ratified
- *with highest-# proposal that acceptor has accepted*
 - so that proposer can substitute this value for its own, in case previous value was in fact ratified
- If proposer receives a response from a majority of the acceptors, the proposer then does a second phase of voting where it sends an $\text{accept}(n, v)$ to all acceptors and wins if it receives a majority of votes.

 NPTEL

So, again this, I am going to do it in slightly 2 or 3 attempts to make sure this is very clear. So, I will. Firstly, I take a high level idea. So, before taking a vote, a proposer checks by sending a $\text{prepare } n$ message to all acceptors, n is a proposal number. In the sense one can say it is like let process 5 get the lock. Let note number get particular lock number something, thus this is the proposal ok that is a proposal.

So, basically there are various proposals in the system, it will be more atomically increasing let us say. And acceptors responds to the promise never to accept any proposal with a number less than n . So, basically idea is that, older proposal do not get suddenly ratified. There are some dormant or dead person protocol, dead proposals this should not be accepted. So, basically some guys proposing, some n , and there are various acceptors, who will, if they accept this particular n , they will basically say that I am going to whatever I have done in the past, I will make sure that I do not accept anything less than n ok; that is, I will, its somehow, because of reason why I have this problem is, that there could be multiple proposers doing at the same time ok.

So, they might be sending something else also. So, their number might be less than n . So, in which case, I am going to promise saying that I will not accept those. And then in

addition, since he has got persistence storage, in case in the past, he accepted a proposal. He will send the highest number proposal that the acceptor has accepted. So, the proposer can substitute this value for its own in case previous value was in fact, rectified. So, our problem is that, some new guys has come in, he is trying to propose some value, but it turned out that in some previous situation, a value actually was proposed and plus accepted by a majority, but before it could be, let us say communicate it to everybody, the proposal in sort of died. So, know. So, thing is that in some sense, the majority was found, but the presiding officer died or something of that kind ok.

So, now if that is a case, a new presiding officers comes in. He sees that he is able to figure out by talking to most of the people, but in fact, some particular thing was already accepted. So, if the acceptors sends back what proposal number that was accepted in the past, then the proposal can see if the majority actually got it, in which case you can say this is a proposing a new value, I will take whatever was accepted in the past. Again we will go through it one more time to make sure that this is clear. So, if the proposer receives a response from majority of the acceptors, the proposer then does a second phase of voting which sends it, where it sends an accept n comma v to all acceptors and wins if it receives a majority of votes. Again let us go through one more time, what exactly I am doing. First of all we have a proposer whose sends a prepare n message as I mentioned the important thing is, if you look at 3 phase commit.

You are basically, the way you are trying to make progress there is, to solve a leader election problem, but this itself seems to be, as they recalled as a consensus problem itself, this is does not simply and simpler. So, idea is to, instead of devolving to leader election, you try to do you, let people propose without worrying too much about if other proposal are there. In the case of 3 phase commit, you do not do that because there is state that has to be saved. So, it is a slightly different problem; that is why it makes same 3 phase commit to, try to go through leader election. So, a prosper, he sends a prepare n message to all acceptors, and then acceptor, he either responds with the promise, never to accept any proposal with a number less than n ; that is, he is basically saying that, I was involved in something in the past, things did not work out.

So, now you got, you have come around and said I have a new proposal. See nothing in the past whatever I know about, has not worked, I am going to accept yours, but I am also going to say, that it is true that, have not heard from anybody whether the previous

proposal got accepted or not. I didn't accept, but I do not know whether there was any majority, I could not figure it out so, but I accepted. So, I am going to tell you what I accepted, what number I accepted. Why is this going on? This is because it is possible that he has accepted a lower value, but he never heard about the factors that were accepted, the majority was actually there, but somehow all kinds of failures happened, multiple failures happened, he never heard about it. So, that is why he is taking a stand so that a new thing has come to me, I am going to accept it, but at the same time I am going to tell you that I was involved in some previous round.


When I accepted this thing, but I do not know what really happened to it. So, now, the proposal here, he receives a response of majority of acceptors, then the proposer does the second phase of voting, where it sends v to all acceptors and wins if it receives a majority. Basically what happens is that if all the guys in the majority, they all say that we are all part of an old round, we all accepted this guy, because that somehow we never heard what happened after that. So, this proposal here from everybody, saying that there is also a majority. In fact, one who knows it is a proposal, then this proposer has to take that value and send it back, send it out. If in case there is no majority, then he is free to propose a new value v . Is this clear. So, this is roughly the idea. So, we will. So, what are safety properties that Paxos guarantees? First is, it is non-trivial. What does it mean? It means that only the proposed value can be learned ok.

(Refer Slide Time: 26:13)

Paxos Safety Properties

Holds regardless of the pattern of failures

- **Non-triviality**
 - Only proposed values can be learned
- **Consistency**
 - At most one value can be learned (i.e., two different learners cannot learn different values)
- **Liveness(C; L)**
 - If value C has been proposed, then *eventually* learner L will learn some value (if sufficient processors remain non-faulty)




It is trivial in case is everybody says always zero or 1 ok, its a trivial thing. So, its not one of those things. Consistency, at most one value can be learned; that is 2 different learners cannot learn different values, very critical. These are absolutely critical thing called distributed files systems, and distributed databases another kinds of systems which require mutual exclusion sometimes or of. So, consistency is very important. Liveness a value c have been proposed, then eventually learner l will some value. So, if sufficient processors remain non faulty. Again there you can see that there is an eventual thing out here again that is why it is did not like, it is not a, Liveness is not guaranteed here. So, essentially if what we will talking about right now is that, eventually there will be some value that will be finally, accepted is sufficient process remain non faulty; that is when.

So, again we have going back to saying that, we cannot really guarantee like this, but we will if we get to long enough it will happen, this is basically what we say.

(Refer Slide Time: 27:47)

Basic Paxos

- Each instance decides a single output value
- A Proposer P should not initiate Paxos if it cannot communicate with at least a Quorum of Acceptors Q
- A successful round has two phases:
 - Phase 1a: Prepare
 - P ("leader") sends a prepare message proposal N ($>$ any previous one by P) to Q
 - Phase 1b: Promise
 - If $N >$ any prev proposal # recd from any Proposer by an Acceptor, then Acceptor must return a promise to ignore all future proposals $< N$.
 - If an Acceptor accepted a proposal previously, it must include prev proposal number and its value in its response
 - Otherwise, an Acceptor can ignore the received proposal.



So, again I will go through in some more detail to see exactly what is going on. There are various versions of Paxos, and we are going now talk about the basic version. For efficiency sake we can do lot more additional things, but we will look at basic one. So, every time we run this basic Paxos, we can say that each instance of this basic Paxos decides single output value again this is, like a consensus, basically with sort of consensus. So, here we have a quorum of acceptance. So, proposer, there are multiple proposers, and they has to be a quorum acceptors. Typically quorum can be majority, and

why this is important. If you have a majority; that means, is that, its always a case that 2 majorities will always have somebody in common. If we take 2 majorities, that has to be somebody in common ok.

That is how information leaks from one round to another round. Without this particular aspect we really do not have any guarantees. So, if you have a quorum, then the basic idea is that, in case something was committed in the previous rounds, then 2 quorums will have a common party. See there is a common party, when a new proposal comes along, then that information is going to, go back to the new proposer; that in fact, something was committed in the past ok. So, basically this has got 2 phases; phase one and phase two. Again this is a somewhat similar to what we saw in 2 phase commit or 3 phase commit. I think is looking closer to, its elaboration of the 3 phase commit 3 phase kind of model. In phase one there is a proposer p , let us we can also often called leaders. Leader sends a prepare message proposal n ok.

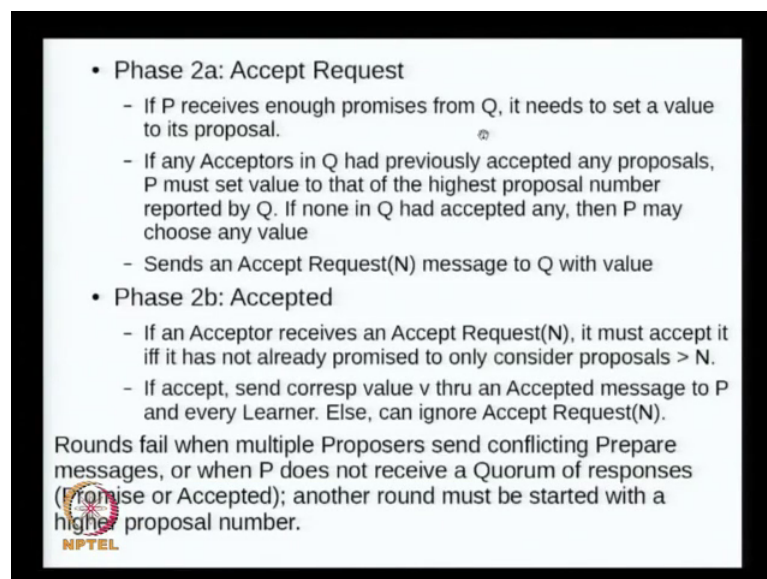
And important that the p should know that he cannot be sending any message which has a same number as previously I send, has to be greater than any previous one send by p himself to q , basically its sending it to quorum of acceptors. So, that the bunch of people who are there he is going to send it to them. Now once it has gone there to the quorum acceptors. Now it is a turn of the quorum acceptors to promise something. What are they promising? If the number n that has been received in this round, is greater than any previous proposal received from any proposer by an acceptor, then acceptor must return a promise to ignore all future proposals less than n . Again we discuss it before it is basically that there are concurrent proposers now. So, what it is means is that. This particular acceptor was involved in something in the past, and he is going to now promise that he is not going to accept.

If he accept this particular promise; that means, he is not going to, he is going to ignore all other future proposal that could come, because they can be delayed. See various they could be. The assumptions here is extremely general, the messages can be duplicated, it can be delayed, can be arbitrary delayed, dropped all kinds of things. So, some previous proposal could come to me now, and that number could be smaller than the one which I have ok. So, the next acceptors must return promise to ignore all future proposals. Again you can notice this is similar to in 3 phase commit. The pre commit phase is basically some kind of a promise, it is basically promising to everybody else, that I am ready to go

forward, I want to flush, but I am going to form to all of you guys, that I have agreed, but until I hear from you I will not flush; that is a basically false, something similar is going on here. If an acceptors accepted a proposal previously, it must include previous proposal number and its value in its response ok.

So, now what is happened is that. Again as you discuss before. If this acceptor actually was part of a previous round, and he had actually said I am going to accept it, but he is somehow has not heard about how it progress in the past. He just not heard anything after that. Then it is possible that, by sure let us say sort of circumstances the. Actually that particular previous thing around actually succeeded and they agreed upon it, it is just that, I did not I know how it again; that is it one might has to include previous proposal number and its value in its response. Otherwise the acceptor can ignore the received proposal, previous drop it on the floor, does not have to do anything ok.

(Refer Slide Time: 33:13)



- Phase 2a: Accept Request
 - If P receives enough promises from Q, it needs to set a value to its proposal.
 - If any Acceptors in Q had previously accepted any proposals, P must set value to that of the highest proposal number reported by Q. If none in Q had accepted any, then P may choose any value
 - Sends an Accept Request(N) message to Q with value
- Phase 2b: Accepted
 - If an Acceptor receives an Accept Request(N), it must accept it iff it has not already promised to only consider proposals > N.
 - If accept, send corresp value v thru an Accepted message to P and every Learner. Else, can ignore Accept Request(N).

Rounds fail when multiple Proposers send conflicting Prepare messages, or when P does not receive a Quorum of responses (Promise or Accepted); another round must be started with a higher proposal number.

NPTEL

So, this is second part, but then the set third would be accept request. If p receives that is a proposal receives, enough promises from q, it needs to set a value to its proposal you get it. The question is what value ok.

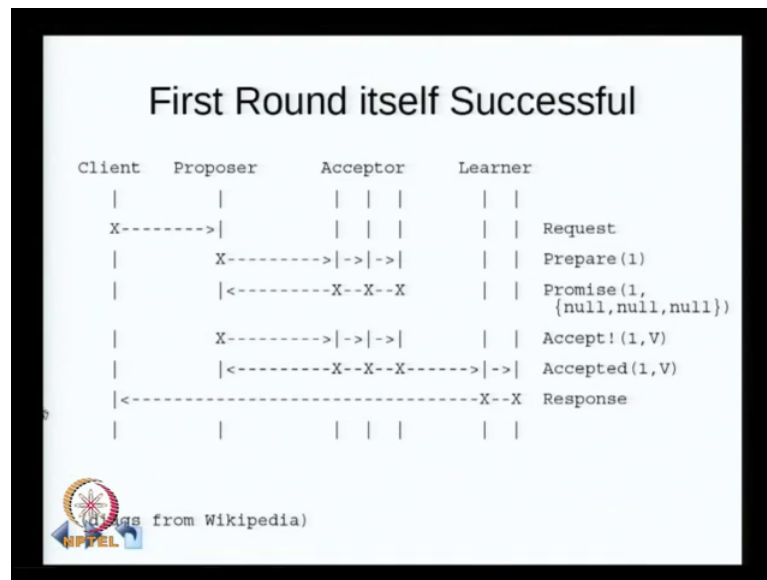
So, there is enough quorum, now in a quorum. So, it needs to set of value to its proposal. Any acceptors in q had previously accepted any proposals, p must set value to that of the highest proposal number reported by q. If none in q had accepted any then p may choose any value. This one is easy if none q had accepted any then p may choose any value, that

perfectly fine, because there is no nobody recognize the acceptors. So, if any acceptors in q had previously accepted proposals, p must set value. Basically it turned out that there was a quorum actually for some previous value, then it is duty bound to put that value. In that case it either puts that value, or the one it is; it selected, because nobody, there is no quorum for any previous values, it sends an accept request to q with value.

In 2 b basically now the proposer has actually send an accept request n with the value, then all the acceptors they. If they receive a accept request it accept it, if it has not already promise to only consider proposal greater than n . So, notice that, it must accept it if an only it is not already promised only consider proposal greater than n . So, again it is a n wlock in aspect, like what we saw in phase 2 phase 1 b. So, if it accepts then you have to send the value v , you can accept message to p and every learner. Else you can again drop the request, completely basic noted. So, the roughly the access algorithm, basically the rounds fail, when multiple proposers are there and they send conflicting prepare messages, and there all. They get inter first each of his request, and each party can decide that, I cannot I do not want to be the part of this round. So, then the rounds fail for a reason, because there not enough people to agree on a particular value ok.

So, you basically have to restart the round with a higher proposal number. What is just thing about this is that, this particular protocol, it was survive any failures from here. Whether it is proposers, acceptors, it does not matter who fails, how this fails does not matter at all, it can actually survive those things, because it guarantees that it is if you have particular value has been decided upon, the majority of people, that will be picked up again. There is a consistency condition is guaranteed. And this is very critical because in a if you are talking about locking situation are mutual exclusion situations, the consistency acceptors are critical; otherwise we have, we can have in consistency system, but the basic problem here is that, once you have this kind of system, is no longer guarantee like this again; that is again an issue.

(Refer Slide Time: 36:58)



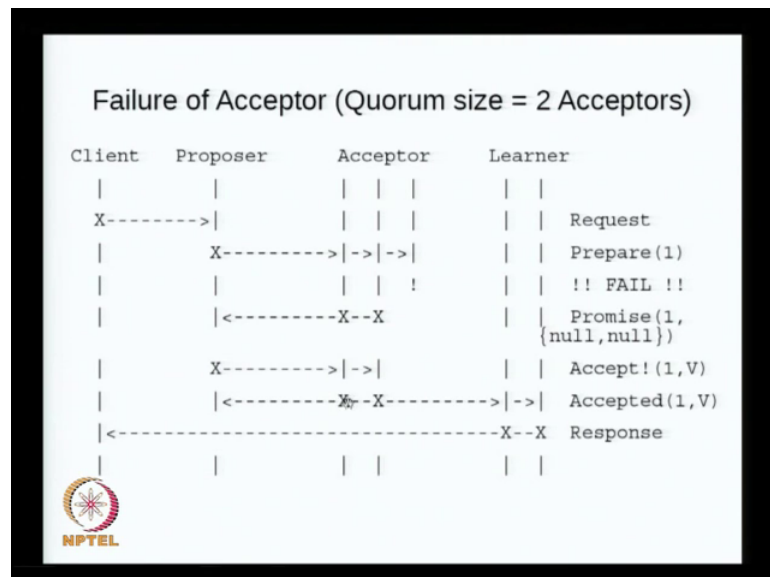
I have taken some diagrams from Wikipedia where it gives us say. So, we will look at quickly some of these cases. This is a client which wants particular service, it will mention that. If you look at Google, there are something called a chubby service, these basically the clients, these are guys are making what is a equivalent of chubby codes. These are basically making request like that, give me that particular lock, give me that particular value that all of us agree upon; that is what this is. So, proposers, there is one proposer who hears this request, they could multiple of this guys. This particular hears it, and is tells the acceptors right. I am going to propose a value one. Do you guys all agree. So, it is basically is in prepare one, and all this parties, now say promise one and they have not seen anything in the past. There not been in a part of any round, for some rounds has happen so far back in time, that everything has been become quotient.

So, we will say null in that case, this nothing there. So, we will basically saying that, I promise that I will, let us say I am basically saying that I am willing to go ahead with (Refer Time:38:30) proposer, and I do not have any previous things that have, all being part of. And once all this messages go back, the proposal in turns says. Now that I got all right responses from all 3 of you guys. I will send and accept showing that has proportional number one. It will value which I decided. Again here proposer which has eigenvalue v. We cannot any of this guy, because they all sent null .And then once I get that again the acceptors now essentially handshake back, saying that we accept what you

propose. Here is a basically comment say accept, it is basically we are saying yes we accept, and the null is basically keep track of the information.

So, that in case it request, it can be responded to back to the client ok. This is roughly the outline of the sequence, in case the first one itself is successful. So, very obvious case, but what we really have to worry is, what happens in the proposer does something and dies somewhere, somewhere here, or some acceptors dies somewhere or the learners die etcetera. We have to figure out what are happens. Here I am not go through all the cases I am just going to go through few of them.

(Refer Slide Time: 39:52)

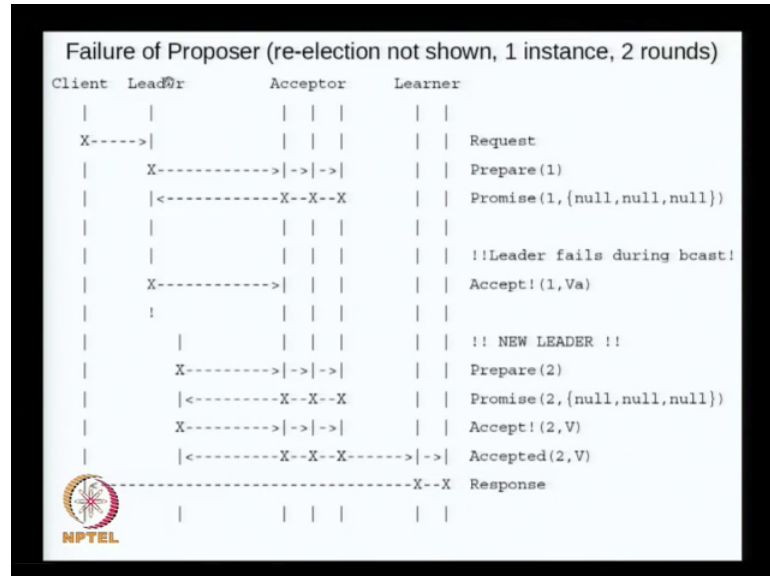


So, let us look at a case where this a failure of acceptor. It is also simple case, but we say quorum size is 2 acceptors. Similarly here also client makes a request, and then proposer proposes a number one as a proposal number. Since it all the acceptors, and it turns out that one of the acceptors dies, but there is a quorum still, because it is a quorum and both this parties basically say that, I was not involved in any previous round, I have value null right. I am willing to go with your proposal number one ok.

That is why this is, and the proposal finally, sends back saying I decided the value v for you, take it that is what this is. And here basically what is happening is that the acceptors now having got the value v they basically saying that they are sending saying that yes I solve a value v, I am going to take v as the value, and acknowledging it. And no once are

there to be able to relay information back to client. So, that in spite of whatever happens here, one of this there is some called additional levels of redundancy or application here.

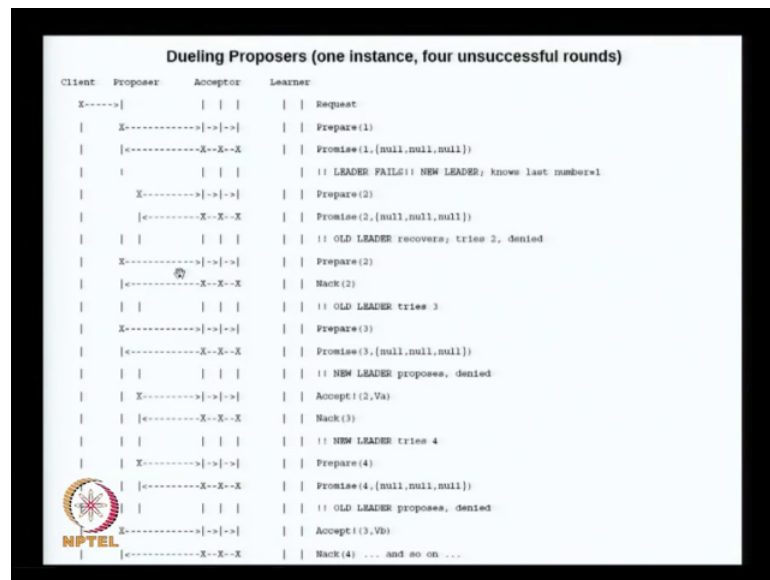
(Refer Slide Time: 41:26)



This essentially it is consider the information back to client. This is again is a simple case what about, the failure after doing something he dies. Again same story, clients makes a request x as prepare one, and then all the parties here, they all say the promise saying that, yes I heard you they were not part of any other previous round.

So, we will go with your proposal, with proposal number one. So, this send it back, but while this trying to send thing back, he dies. So, basically what happens is that, this part is do not get this value, whatever was accepted ok. So, what he was initiating, it was trying to say accept 1 v a, but somehow it did not get through. So, because this particular proposal die, you can have some other proposal, come in the picture, who were seen that some of there is something not in this system. We do not care when it comes up, how it comes it can be anything. He decides to, say that I have a new proposal prepare to, and this is similar to what happen in the previous case. So, here I will list this; Wikipedia article talks about the re election, but it can be actually concurrent this actually can be somewhere concurrent.

(Refer Slide Time: 43:25)



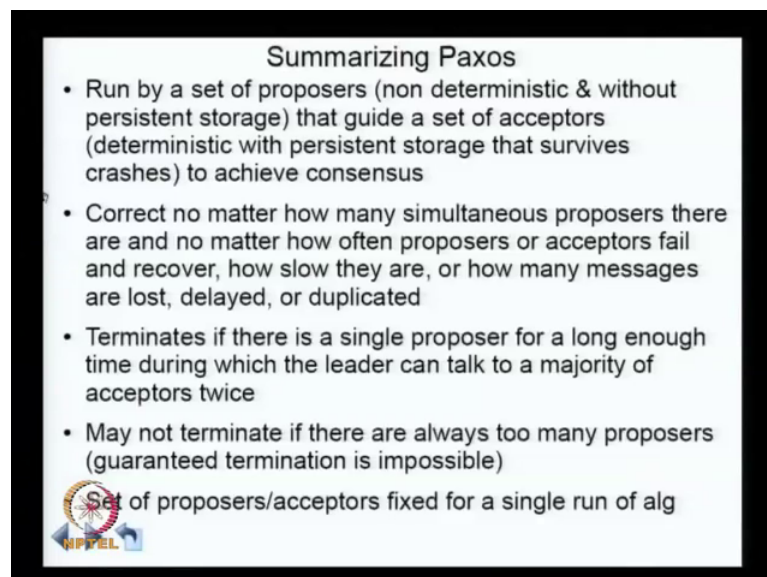
Because this guy can decide to, arbitrary decide something is working somewhere, and decide. Some will come to that kind of situation, where there are multiple proposers operating at the same time. You can also have slightly more complicated situations. This one is slightly more involved. Here what happens is that, just like previously this client make a proposal, he says prepare one, and all this parties have basically saying yes I am ready to go with you, but this leader fails, the newer comes in, he starts the new protocol again, saying prepare to. And he sends all the second part, because all this guys are not heard about what happen to the proposal number one, when you say any proposal, they decide we will go with this new guy; that is what they doing here promise to null, null, null, because they dint accept anything. So, far ok this is promised that. So, ok.

So, they send back this particular promise. Now the old leader can recover, and he can try to, because he knew about one. So, it is going to try two, but it says prepare for two, then on this acceptors they can send an, saying that we no longer comfortable what you are proposing. So, that is what this is the new acknowledgment, and then the only, the tries what did I do. So, now, the, only the tries 3. So, prepare 3 is sent out. Now again these parties, this acceptors I am not sending progress with respect to 2 also for some reason. So, they will now say, I saw a 3, but now I will no longer accept anything kind of two, because there is not going anywhere. So, it is now says I am promising, to see if 3 can be made to go forward; that is what this is, and now the new leader came in now it is a accept, but if this is accept.

Because this guys already said, that they are trying to see how the proportional number 3 is going to fair right. They cannot accept what the new leader has proposed after the first leader died. So, that is why there is a lack again here, and the new leader tries 4 and it is a prepare 4 for some reason. I can prepare 3 was not going anywhere, they will probably decide to go with round 3. So, you can actually have the situation where multiple proposers keep on overtaking each other, and it might not really converge that is possible, but in point in case that is a minute, there is any convergence and majority have the guys agree on a value.

Because of the quorum kind of property even if one party agrees on right, the other the any new quorum that starts, they will know about that particular value. It is a same principle that is formed in committees. If you look at a government system will find out lot of committees in variably, there is always few one or 2 parties always common to certain, multiple committees, and because of that what happens is that, any decision taken in one committee this always known to other committees also, this is a part of the way most governmental systems work.


(Refer Slide Time: 46:51)



Summarizing Paxos

- Run by a set of proposers (non deterministic & without persistent storage) that guide a set of acceptors (deterministic with persistent storage that survives crashes) to achieve consensus
- Correct no matter how many simultaneous proposers there are and no matter how often proposers or acceptors fail and recover, how slow they are, or how many messages are lost, delayed, or duplicated
- Terminates if there is a single proposer for a long enough time during which the leader can talk to a majority of acceptors twice
- May not terminate if there are always too many proposers (guaranteed termination is impossible)

Set of proposers/acceptors fixed for a single run of alg

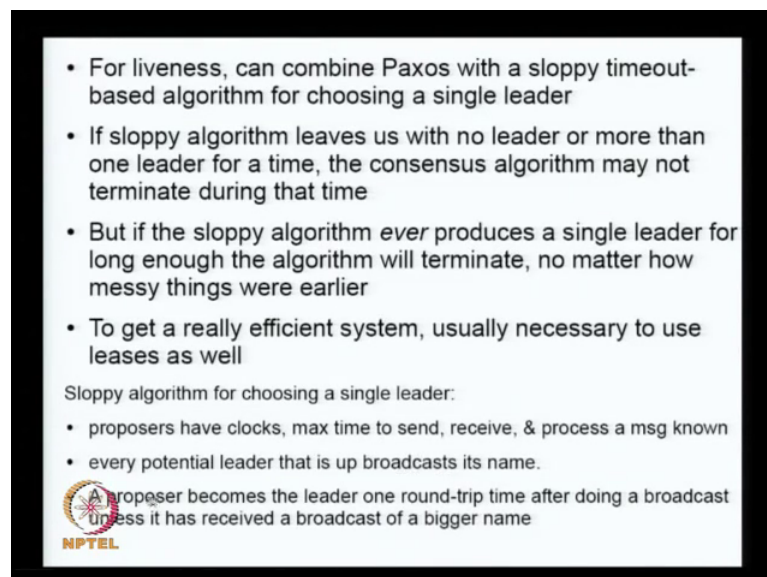


So, let us this again summarize Paxos, is run by set of proposers they are highly non deterministic, they can start any time they like and without persistent storage, but guide a set of acceptors, this parties have or determines persistence storage, that survives crashes, because they have to remember.

What they were doing sometime in the past. If for example, they were part of particular proposal and that agreed to commit the particular value right. If you agree on particular value they had remember what were do. So, that that is why they need persistence storage. What is interesting about this is correct, no matter how many simultaneous proposers there are, and no matter how often proposers acceptors fail and recover, how slow they are or how many messages are lost delayed or duplicated, it is a very interesting property its non trivial. What is important also is a terminates this particular protocol. If there is a single proposer for a long enough time during which the leader can talk to majority of proposers twice. Again you can see why this twice because we look at the phase one and phase 2 we need at least twice. Again if you remember the time d synchronous model we talk briefly mentioned, there also the model is that, it is mostly stable and once in a while unstable ok.

So, single if it is stable enough then you can get something, you can get forward progress again at. So, on the similar is going on here also, may not terminate if there are always too many proposers, guaranteed termination is say anyway not possible ok.


(Refer Slide Time: 48:25)



- For liveness, can combine Paxos with a sloppy timeout-based algorithm for choosing a single leader
- If sloppy algorithm leaves us with no leader or more than one leader for a time, the consensus algorithm may not terminate during that time
- But if the sloppy algorithm ever produces a single leader for long enough the algorithm will terminate, no matter how messy things were earlier
- To get a really efficient system, usually necessary to use leases as well

Sloppy algorithm for choosing a single leader:

- proposers have clocks, max time to send, receive, & process a msg known
- every potential leader that is up broadcasts its name.

 A proposer becomes the leader one round-trip time after doing a broadcast unless it has received a broadcast of a bigger name

NPTEL

So, it turns out it can be something slightly better, if you want looking for liveness, you can combine Paxos with a; that is called as sloppy timeout based algorithm for choosing a single leader. So, the basic idea is that, if you are able to choose the single leader, then that particular persons value can be make to stick; that is basic idea. The sloppy

algorithm leaves us no leader for more, or more than one leader at a time, because a partitions etcetera it is possible, that there can be more than one leader ok.

So, for that reason it may not terminate, but if the sloppy algorithm ever produces a single leader for long enough, the algorithm will terminate, no matter how messy things were earlier. For example, in example the sloppy algorithm for choosing a single leader (Refer Time:49:25) suppose the proposers have clocks max time to send, receive and process a message known. Suppose we know this values, then every potential leader that is, up broadcast its name. Now a proposer becomes a leader, one round trip after doing a broadcast, unless it has received a broadcast of a bigger name basically what it is doing is, everybody broadcast, and then because it knows about the max time to send, receive and process messages known. It decides after one round trip, that it is a leader. So, mostly the things were reasonable in the system, only one guy will ruling, because everybody is following this protocol.

That it has if it has received a broadcast of a bigger name it is says, I am not anyway I cannot be the leader. So, the biggest guy wins, but if so happens is that there is some network problems and what not, or the system is very unstable, then multiple parties can believe that within, because I have some kind of max timeout right, have some max amount of time here. I can decide that I am not heard anything bigger than my name therefore, I become a leader. While it turns out that multiple parties can come to that conclusion.

And so therefore, again you have a situation with multiple proposers with multiple and they need to have to dwell it out, but if the system is reasonably stable, then only one of them are essentially win, and you can proceed. So, in a sense you can slightly make it easier, instead of the completely, proper situation with the Paxos, where anybody can propose any time. If you use this sloppy method, you can essentially try is this system is reasonably in a sense it is state, with a too many failures, it will actually very good. So, this kind of algorithms are used, in many large scale storage systems, and as I mentioned Google has in therefore the various clients, they have a locking service, which I am told these are Paxos we connect.

So, many other various other large scale persistence or web scale systems also used, Paxos kind of models and. So, I think I will conclude here today, and I will continue with

a similar set of problems, but not close to what is called group communication systems, where you have to order the messages, and this again is quite critical for storage systems, because they might want to keep some material term which is consistent amongst multiple nodes, and they have to exchange messages to come to that consensus, and it turns out that if messages sent out, going different orders for 2 different SMS I send. Then they could be some confusion about what really happened on their receiving side. So, there is an important aspect in last case storage system figure out how to do, reliable message delivery so that multiple messages I send, they going in the same order, or similar types of constraint exceptions have to be guaranteed on the receiving side.

And we look at that one we will again connect that particular problem, to the kind of problems we saw before, either commit or consensus kind of problems.