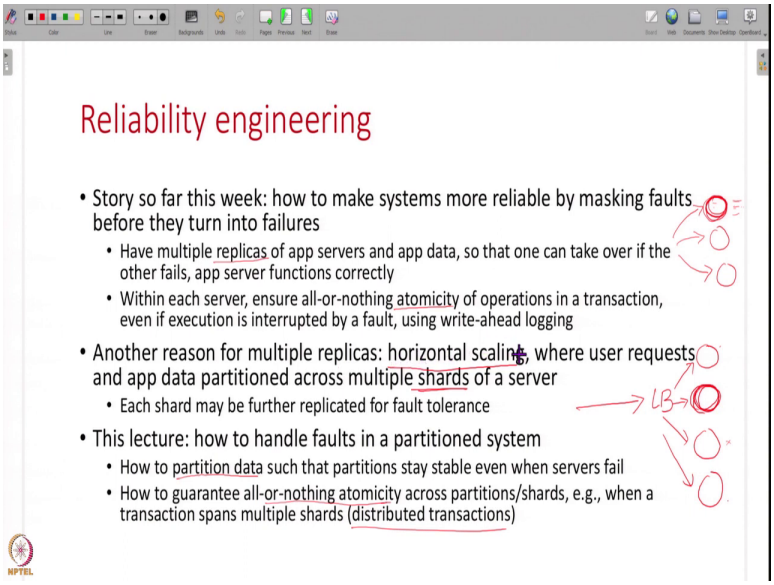
Design and Engineering of Computer Systems
Professor Mythili Vutukuru
Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Lecture 54
Distributed Transactions

Hello everyone, welcome to the 39th lecture in the course Design and Engineering of Computer Systems. So, in this lecture we will continue our discussion on fault tolerance and reliability engineering, we will study a topic called Distributed Transactions. So, let us understand what this topic is.

(Refer Slide Time: 0:35)



So, the what we have seen so far this week? What has our story been? We have seen how to make systems more reliable by you know if any faults occur in the system due to either hardware faults or software faults or power failures any such faults occur how do we mask that fault and make the system more reliable. So, that this is not noticeable as a failure to the end user.

For example, we have seen that you know instead of doing all the processing in just one server what if the server fails, so therefore you have multiple replicas of the server either you know active-active or active-passive or whatever and you know you repeat the operation at all of these replicas so that the state of the system can be saved even if a failure occurs and the user would not notice that failure.

And also, within each server also we have seen that if the server is doing multiple steps then we have to ensure atomicity within each server also. So, that if when you know in the series of steps in a transaction of the failure occurs in between those steps then the other steps also will continue and you get all or nothing semantics. So, we have seen replication and we have seen atomicity so far.

So, the next topic that is important for fault tolerance is that of how do you ensure fault tolerance in a horizontally scaled system. So, if you recollect from a few lectures back we have studied this concept of horizontal scaling there also we have said that you know you need multiple replicas for example if your server can handle a load of only hundred requests per second but you are getting a traffic of you know thousand request per second, then what do you do?

You will try to improve the performance of the server as much as possible but beyond a point this capacity may not improve so then what you will do is you will add multiple replicas of the server and you will you know put a load balancer or something in between and the load balancer will distribute traffic to all of these replicas so if thousand requests per second are coming in 100 100 100 you have 10 of these replicas and you can handle all the incoming load.

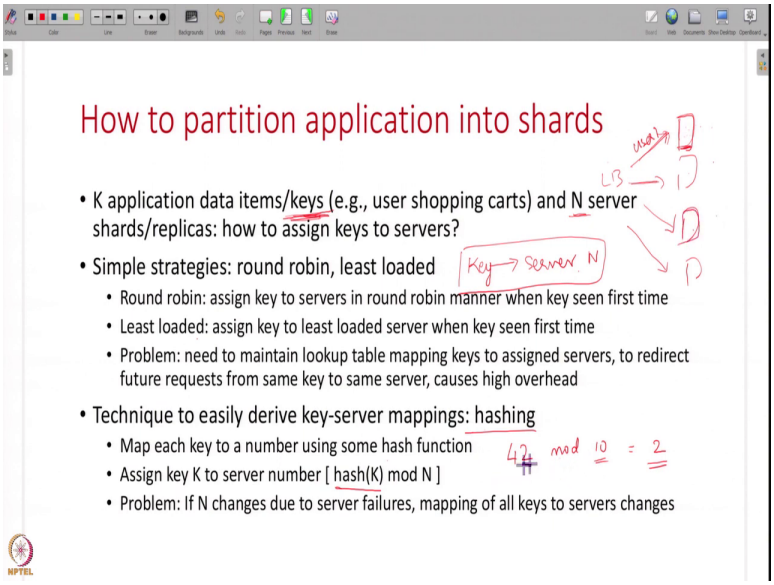
So, this is also another reason why you have replicas and you know these are also called partitions or shards of a system you know you partition the application data request everything you partition them to these different replicas. So, note that these when we use the term replica here versus when we use the term replica here there is a difference. Here each of these replicas is different that is why let us call them you know shards or partitions each of these is a slice of the server handling of part of the traffic, whereas for fault tolerance when you have replicas all of these replicas are doing the same work repeatedly for redundancy these are two different concepts.

Now, when you have horizontal scaling and you have faults in the systems, we need a few things to happen. For example, we want to partition the data and request such that a lot of these partitions do not change even when servers fail you know suppose a server here fails then you do not want to disturb all of these other servers also you want to do this partitioning in a way that is stable and you also sometimes may need atomicity across partitions, for example if you have a transaction that does some work here and some work here then across these servers you need

atomicity, those are called distributed transactions you know transactions that span multiple partitions or multiple shards.

Earlier the transactions we saw were in a single system where you log to disk and you are guaranteed do some logging and guarantee atomicity but now we are talking about transactions spanning multiple partitions of a system, these are the two topics we will study in today's lecture which is basically at the intersection of you know horizontal scaling and fault tolerance.

(Refer Slide Time: 4:35)



So, first the first question comes up how do you partition an application into multiple shards? So, we have seen some of this before when we were studying load balancing and horizontal scaling. So suppose you have an application that has some number of data items or keys, we will use the term keys here this is nothing but say user shopping carts you know you store them as keys and values there are some keys which are say user shopping cart ids and you want to distribute these keys among the n server shards or replicas, how do you do this?

How are what are the various ways in which you can assign this application information to the various servers. You have to of course partition only then you know across these n servers if you you know give some keys here some shopping carts here, some shopping carts, some shopping carts here and then the load balancer can kind of redirect the traffic of these shopping carts to the server it can partition the traffic and you can have horizontal scaling.

So, the question comes up how do you assign these application keys or application data items to the server replicas, there are many strategies we have seen this before one is of course a simple round robin strategy which is when a shopping cart when the user starts a shopping cart the first shopping cart goes here the second you users shopping cart goes here third goes here fourth goes here you can do like a round drop in whenever a key comes for the first time in the system you assign it to some server some key k is assigned to some server n.

But the second time that key comes you cannot once again assign it to a different server if some user one shopping cart has been created at this server in the round robin manner the next time that user one comes he has to go to the same server again so you have to remember which key to which server you have assigned you have to remember these mappings so that the next time you can use the same mappings.

So, you can use round robin you can use least loaded instead of you know going round robin or this server is handling a lot of load, therefore I will pick a least loaded server like that you can use many different strategies but across all of these strategies the problem is that you need to maintain these mappings you know for every key which server it has been assigned to for the first time when the key was seen for the first time you have to remember this mapping because in the future once again you have to send the same key to the same server if your server is storing all the shopping carts on its disk or in some database locally, then that user shopping cart request should go to the same replica in the future also only then it will work correctly.

So, if that is the case you have to maintain these mappings which is a high overhead you know if you have a million users there is a big large data structure with million entries and every time a key comes you have to look through this have I assigned it a server all of that is high overhead. So, ideally you want a technique where you do not store any information but you somehow know which key is assigned to which server.

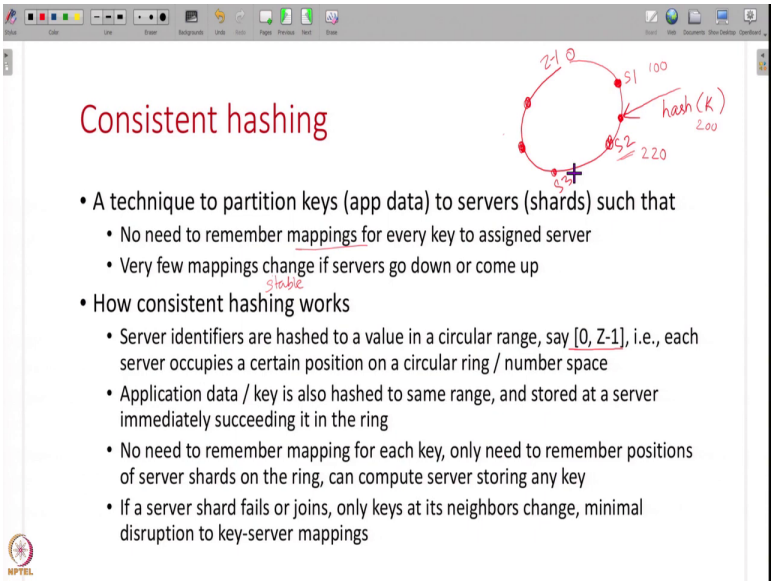
So, one way to do that is hashing, so you take a key say you know the shopping cart id of the user you take that key you hash it to some number you hash it to some number say 5 or 10 or 100 something you hash it to some number using some standard hash function and if there are n servers you simply do this hash mod n and you assign it to that server, suppose if your hash value is 42 and you have 10 servers you do 42 mod 10 and the remainder is basically 2 so you assign it

to server number 2. In this way, given any key you do not have to maintain any table you can simply do this hash do the modulo operation and you know which server to send it to again in the future you see the same key you do this calculation you send it to that server.

So, this is an easier way that does not require maintaining all of these mappings but what is the problem here the problem is that what of this n changes, what if one of the server fails? You know we are talking about fault tolerance this week so we have to think about failures what if the server fails, then suddenly now for the same key you are doing mod 9, so this works out to what? 6 so now suddenly so far this key was stored at server 2 now you are saying store it at server 6.

So, anytime a server fails all the keys will get remapped to a different set of servers now so far the shopping cart was maintained here now the server has to transfer the shopping cart to somebody else, it is all very messy. So, using this hashing idea the way I have described here it is very messy when server failures happen and servers fail or are added and so on if this n value changes.

(Refer Slide Time: 9:22)



So, what the modern systems do most of real world systems do is they use a way of hashing that is called consistent hashing, this is basically a technique to partition keys or application data items to multiple server shards such that you do not have to remember any mappings you can

derive the mappings whenever you want at the same time these mappings are fairly stable they do not change a lot of servers come up and go down also.

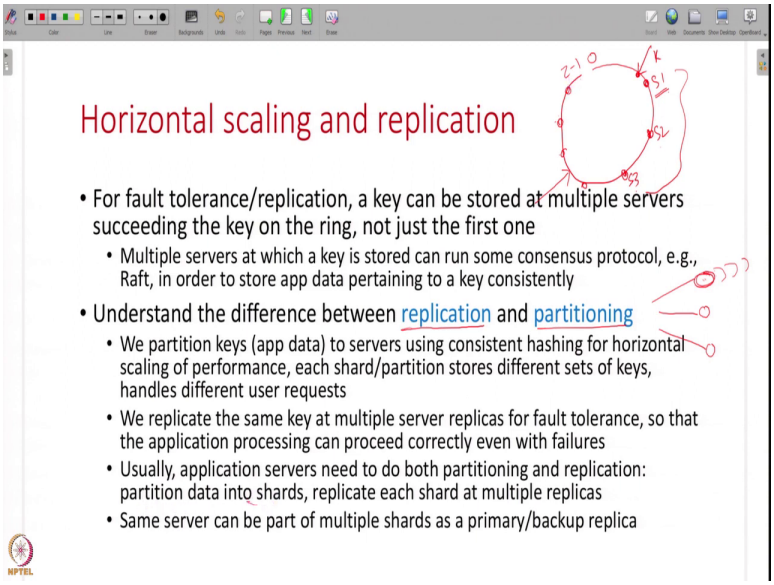
So, the idea is very simple how does consistent hashing work you know you take all your server identifiers you know say the ip address or some identifier of the servers you hash them into some range some say 0 to 1000 or 0 to 999 something into a thousand value range something like that from 0 to some z minus 1 some large number you take this range and you hash your server identifier into this range so hash of server 1 turns out to be say 101. So this is server one it is located so you can think of this as a ring from 0 to z minus 1 and server 1 is located here the hash of server 1 is this value hash of server 2 is this value hash of server 3 is this value and so on all the server shards you place them you visualize them over a ring like this.

And you take your application key also you know you take your application key hash it also on to the same range the hash of your key also you hash it into the same range 0 to z minus 1 and the key will land somewhere over here then every key will be stored immediately at the server following it on this ring. So, what does that mean? You just remember what the servers id identifiers are where the servers are located on a ring where all the server shards are located on a ring that is all you remember, whenever a key comes you hash it you see on the ring what is the server immediately following it on the ring suppose you know the server one is located at position 100 server 2 is located at position 220 and your hash works out to 200 then the server immediately following it is server 2 this key will be stored at server 2.

So, you do not have to remember for every key you know this key this server this key this server you do not have to remember the mappings at the same time this is stable, why? If this server fails then only these keys will change they will get assigned to the server instead of this server, all the other servers whatever keys they are handling they are all stable everything did not change only this set of keys got remapped to a different server.

So, there are minimal disruptions, in case of servers failing or joining. So, this is why it is called consistent hashing it is a stable hashing to map keys to server shards nodes in a system such that you do not have to remember a lot of state at the same time it is fairly stable in the face of failure, so this idea is widely used in systems in order to distribute any you know set of keys to a set of servers.

(Refer Slide Time: 12:45)



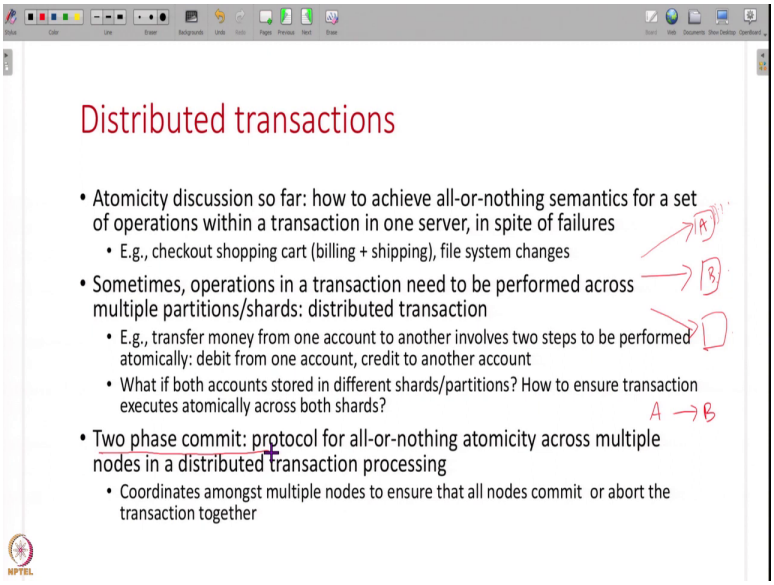
So, now let us understand the connection between horizontal scaling and replication you can use the same consistent hashing idea and you can do both horizontal scaling and replication, for example if you are mapping all of these you know servers and keys to this ring your server is some server 1 is located here server 2 is located here server 3 and so on all the servers are located on this ring and you know a certain key k that hashes to this value you can store it at server 1 or if you want fault tolerance you can say I will store it at server 1 server 2 server 3 I will store it at 3 different servers following it on the ring.

So, you can do both partitioning as well as fault tolerance. So, you can basically have multiple partitions and each key instead of storing it at just one server you can have multiple replicas of the server and you can store the key at multiple replicas. So, you please understand the difference between replication and partitioning, in partitioning the goal is to assign keys to one server so that the key is stored at that server and different servers will store different keys whereas with replication the goal is to store the same key at multiple servers for redundancy and run some kind of you know consistency consensus protocol.

And typically you will do both in real systems you will do both so you will map keys to servers and you will map a key to multiple servers so that all of these servers the key can be replicated at you know you can say I will replicate this key at these three servers some key that comes here I will replicate it at the next three servers following it on the ring. In this way you can use the

consistent hashing idea to do both partitioning and replication, you can partition your data into shards and you can replicate the data at each shard at multiple other server replicas.

(Refer Slide Time: 14:34)



So, the next concept that we are going to see in a distributed system like this where data is partitioned the other thing that we have to guarantee is atomicity across these partitions or across these shards, for example, we have seen examples of where atomicity is needed you know all or nothing atomicity, for example, you are doing some check out, file system changes and all of that all of those were limited to a single server all the examples we have seen.

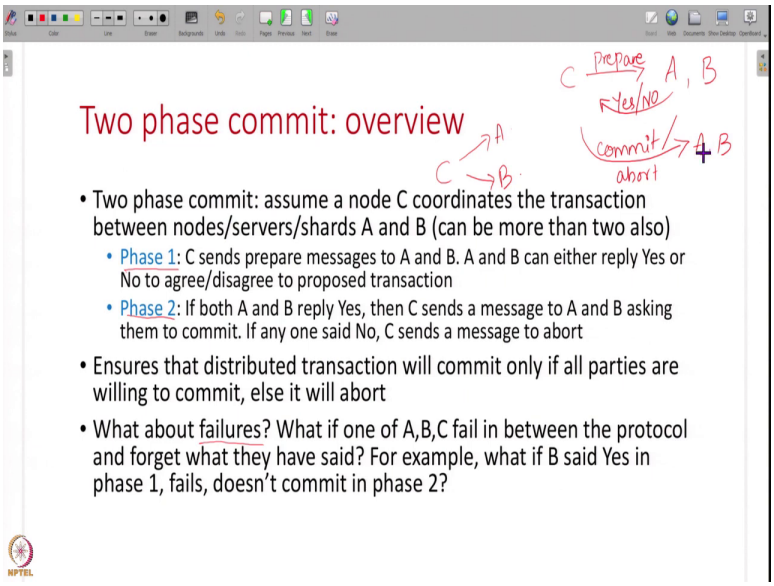
Now, what if you have to do a transaction across multiple partitions suppose you know you have a banking server that is you know partitioned the account information of users across multiple shards or replicas like this you know some accounts are here some accounts are here some accounts are here and you have to do a transfer money transfer from ah account a to account b and account a is stored here and account b is stored at this server.

Then what you have to do what you need is a transaction spanning these two different partitions where you have to deduct money here you have to add money here and this has to happen atomically either it happens in both places or it does not happen at all, that is a distributed transaction which spans multiple replicas or shards each of which can have you know each of

these shards can be replicated for fault tolerance and all of that in this complex system you want to do a transaction you want all or nothing atomicity.

So, the protocol used for that for these distributed transactions is what is called a two-phase commit protocol, this protocol guarantees all or nothing atomicity across multiple different nodes or multiple different shards of a application and it coordinates across multiple nodes. So, the basic idea is the same of what we have seen of you know right ahead logging earlier but it is a little bit more complicated by the fact that now you have to coordinate across multiple different nodes.

(Refer Slide Time: 16:25)



So, let us briefly see how two phase commit works like from the name it is obvious that there are two phases in the protocol, so assume that you have a node C that is coordinating a transaction between two different nodes A and B, the transaction has to has multiple steps some of which run at a sum of which run at B and of course this protocol can be extended to multiple large you can have three four how many other nodes you want it is not just for two nodes.

So, how does this protocol work? In the first phase C will send out what is called a prepare message to all the nodes to all the nodes A B and so on it will send out a prepare message this prepare message is saying that look here is the transaction I want to run are you ready for it are you prepared for this transaction, will you execute this transaction then the nodes will reply in

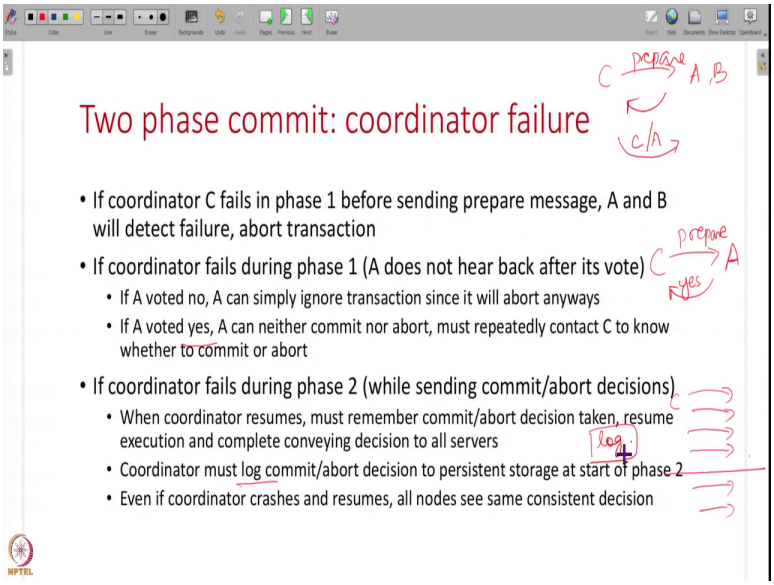
phase one they will reply either yes or no, if for some reason they cannot execute this transaction you know the user does not have enough money in his account to deduct in which case that server will say no this transaction would not work for me.

So, in this way your nodes can vote yes or no for the transaction across multiple different entities then in phase two if all the nodes vote, yes then the coordinator will say ok good we can go ahead with this transaction if at least one of them says no then the coordinate has to abort the transaction so then in phase two the coordinator will send the decision of whether to commit the transaction or abort the transaction to all the nodes.

And when the nodes now receive the commit message they have to commit they would have voted yes only then you will get commit if anybody wanted no you will get only abort message so all nodes would have voted yes for commit to happen then if they voted yes in the first phase they must go ahead and commit the transaction because everybody else is ready now everybody should jointly commit the transaction.

So, this is the two phases in the two-phase commit protocol and this will ensure that distributed transaction either all of them will commit or the transaction will be aborted nothing will happen. In the simple description it is okay but now again the theme of this week is you know fault tolerance, reliability failure so what if there are failures what if this node fails in between what if this node fails in between you know how do we guarantee that atomicity will still hold even in spite of failures. So, now let us consider the various cases the failure cases and see how two phase commit behaves in these cases.

(Refer Slide Time: 18:55)



So, first let us consider the coordinator failure, so we have this protocol coordinator in phase one it is sending a prepare message then it is getting back replies and then it is sending a either a commit or a abort decision. Now, of course if any failure happens you know before this phase one completes while sending prepare message before the transaction starts its okay if some nodes have heard some have not heard then its okay they did not you know vote yes or no or anything so everybody will abort the transaction that is okay.

Now, if a failure happens in the middle of phase one you know some nodes have voted some have not voted then a failure happens now consider this node A you know the coordinator asked for it to prepare and the node A voted something either yes or no. Now, if node A voted no and then the coordinator failed, then what it is okay A will you know through some heartbeat or something it will detect the coordinator failed, so it's okay, anyway A did not want to do the transaction and now the coordinator has failed so its happy it will say okay the transaction is aborted anyways. Anyway, there is no way the transaction can continue because I voted no.

So, A will forget about the transaction it is done no complication there. Now, what if A voted yes? C asked coordinator asked are you ready and A said yes and now the coordinator has failed, now what should A do? It cannot simply forget about the transaction no? What if the transaction is going on what if somehow A is not able is not connected to the network a is not understanding what is happening as everybody else doing the transaction I voted yes then it is my responsibility to you know complete the transaction. So, in such cases A cannot just ignore it cannot commit or

abort it cannot go ahead and commit also, why? Because maybe somebody else voted no, the coordinator is down it is not telling me what to do.

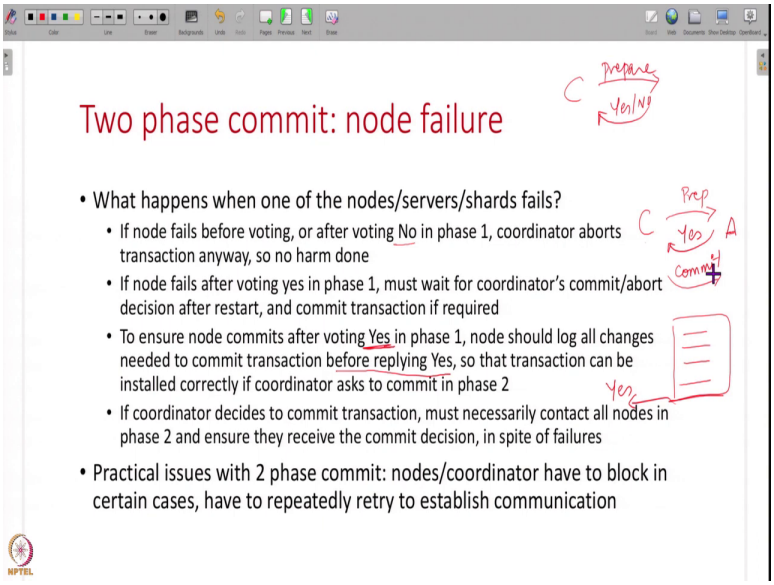
In such cases a must wait it must block it must repeatedly keep contacting is the coordinator up is the coordinator up keep checking find out if the transaction finally in the last in the second phase of whether it committed or aborted and then execute the decision if it is commit it has to execute the transaction. So, in this case if the coordinator fails when some of the nodes are voted yes then the coordinator must come back and contact those nodes and those nodes should wait to hear from the coordinator about the decision of the transaction.

Now, what if now phase two you know something has happened people voted yes or no the coordinator has made a decision commit or abort and in the middle of sending those commit abort decisions the coordinator has failed, then what the coordinator should of course do some logging you know before starting to sending out these commit abort messages the coordinator should actually write down somewhere in a log it should write down the decision these were the votes I decided to commit or I decided to abort the coordinator should make that in write that decision down in a persistent log and then start sending the responses to all the nodes, why?

Because if it crashes it has to remember what was the decision taken and continue you know you told if there are 10 people in the transaction if you have told four people about the commit about decision you cannot just stop there no everybody should hear about the decision if it is especially a decision to commit everybody should know about it, therefore if the coordinator has communicated with four people and crashed here it must restart and complete the rest of the communication.

Therefore, in order to preserve its state across restarts the coordinator must log its commit abort decision to some persistent reliable storage before doing this entire process so that if it crashes in the middle of the process all the nodes will see the same decision of the coordinator. So, once again this idea of write ahead logging is being used here any decision you make before you execute the decision you first log it so that even if you fail you can repeat the decision reliably. So, this is about coordinator failures now what about if nodes if these different you know replicas or shards or you know these nodes of the system they fail in between this two phase coming, then what?

(Refer Slide Time: 23:01)



So, again let us see various cases. So if a node fails before voting itself you know it never said yes or no or if it voted no in the phase one you know the coordinator is sending a prepare message and the vote the nodes have to vote yes or no and now in this phase if a node has voted no and it has crashed then it is okay does not have to you know bother about this transaction anyways why because anywhere the transaction will abort nothing will happen, so it can ignore.

But if a node you know for the prepare message if the node voted yes and then it crashed then what then you have to follow up on this transaction you have to know it you must restart you must you know try to contact the coordinator find out what happened did that transaction commit or abort and if it has committed then A should also commit the transaction.

So, A must remember all of this after it restarts how does it do that therefore before voting yes you know before voting yes before replying yes you must do some logging, it must do a write ahead log of all the changes that this transaction involves note them all down in a log commit this entry and only then you reply yes to the coordinator.

So, that even if you crash once you come back up you talk to the coordinator the coordinator then asks you to commit you have all the information ready with you to commit, you have to prepare you have to do the pre-commit phase that we have studied before and only then send a yes so that in the future of course you cannot commit right away you do not know what the other

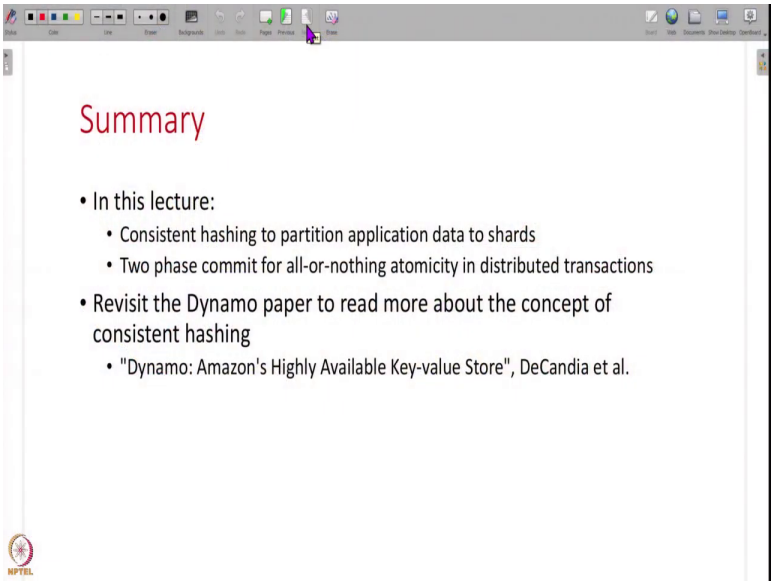
people have said whether they voted yes or no so if you crash here you come back up check with the coordinator if it is a commit then you go ahead and commit install your transaction.

And the other thing to note is that if the coordinator decides to commit a transaction and the nodes have failed it is also the coordinator's responsibility to ensure that any such node has failed it will have to contact the coordinator know the commit abort decision tell everybody this commit about decision you cannot simply leave it saying okay the node failed I do not know what especially if your transaction is committing if it is a transaction that is aborting anyways its okay.

So, this is the high level overview of two phase commit it has many different complications especially with node failures and so on and across all of this you have to remember this very important thing that there are many issues around failures you know if nodes fail coordinator fails in some cases they might block you have to repeatedly contact the other guy and ensure that whether this commit or abort decision happens correctly.

So, this is a standard protocol used for distributed transactions of course there are many improvements to two phase commit and many variants of this are used today but any real-life system that has multiple shards of application data and has to you know guarantee some transaction all or nothing atomicity across these shards will be using some such distributed commit-based protocol like this to ensure that all the changes happen atomically.

(Refer Slide Time: 26:25)



So, that is all I have for this lecture in this lecture I have talked to you about two things mainly one is how do you partition application data across shards in a way that is stable even if nodes fail and so on. And the other thing is how do you do all or nothing atomicity in a distributed setting like this in a distributed transaction using the two-phase commit protocol.

So, you can read this dynamo paper that I have pointed to you earlier this also has more detail about the concept of consistent hashing you might study that more and this idea is widely used in many real-life systems today. So, please read up a little bit more on it to understand it in more depth. So, that is all I have for this lecture, let us continue our discussion in the next lecture, thank you.