

Design and Engineering of Computer Systems
Professor Mythili Vutukuru
Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Lecture 52
Replication and Consistency

Hello, everyone, welcome to the 37th lecture in the course Design and Engineering of Computer Systems. In this lecture, we want to continue our discussion from the previous lecture where we said that we need replication for fault tolerance, but then the question came up, how we do this replication in a way that all the replicas are consistent with each other. So, in this lecture, we want to understand how we can achieve replication with consistency, so, let us get started.

(Refer Slide Time: 00:48)

The slide is titled "Replication and Consistency" in red. It features a list of bullet points and several handwritten diagrams in red ink. The diagrams include: 1) A flow from "old state" to "new state" via an "input". 2) Three circles representing replicas, with arrows showing them all receiving the same input and transitioning to a new state. 3) A diagram showing one replica with an 'X' and an arrow pointing away from the group, representing a replica that is out of sync. The bullet points are as follows:

- One way to build fault tolerant systems: **replicated state machines**
 - Requests from clients (inputs) are processed by all replicas
 - All replicas start with same state, handle same input, so will stay in the same state always (assuming deterministic processing)
 - Used to build active-active replicated systems
 - Used to build reliable distributed data stores for active-passive systems
- Challenge in building replicated state machines: **consistency**
 - What if a replica was down and didn't receive some input?
 - What if a replica received some inputs in a jumbled order?
 - How to ensure all replicas are always consistent, i.e., in the same state?
- This lecture: mechanisms for replication that guarantee consistency

So, we have seen this concept before of replicated state machines, when you do active-active replication, that technique is also called replicated state machines. You have multiple replicas, all of which are in some, old state, they are maintaining some application state, then you get some input that is you get some requests from the user, say, to add an item, or delete an item from a shopping cart.

And all of these replicas handle the same input and they all go to the same new state, new application state. So, all replica start with the same initial state, they handle the same inputs from the user in the same order and therefore, they will reach the same new state and again, the next input, the next input, the next input, and so on. So, this will keep on happening, so all the replicas are always in sync with each other.

So, this is what happens in active active systems. But if you are using active-passive, of course, you are not doing a replicated state machine, the passive replicas are not at the same state as the active all the time, but then you are using some data store or something to store your checkpoint using a reliable data store. And within that data store, once again, maybe you are doing a replicated state machine.

So, these replicated state machines are sort of the fundamental building block of building fault tolerant systems, whether it is active active or active passive inside the data store does not matter, this idea is very, very important. And when you build these replicated state machines, it is very important to have consistency.

So, why will not the system be consistent if you are, giving the same inputs to all of them, they always will stay in the same state, where is the issue of inconsistency coming up? It comes up because of failures or faults in the system, life is not perfect. So, sometimes, when you are sending some input, one of the replicas may miss the input, all these other replicas received the input, but this one replica missed getting that input because it was down it had some, temporary failure, and it missed that input.

So, therefore now this one guy will be in a different state from all the other replicas and from here on, it will keep handling input, but it will never come back to the same state or sometimes there could be reordering in the network, packets can get jumbled up at some router and inputs are received in some jumbled order. And because of that you might reach a different state.

So, there are faults in systems in real life, because of which these replicated state machines may not stay consistent with each other, may not stay in the same state all the time. And in this lecture, we are going to see how you can do replication in a way that guarantees consistency. So, before we understand how to guarantee consistency, we have to understand what we mean by consistency, we have to define that correctly.

(Refer Slide Time: 03:49)

Consistency models

- Many definitions / models of consistency, some strong and some weak
- **Atomic consistency** is example of strong consistency model
 - All inputs / operations (e.g., add/delete/view items in shopping cart) executed at all replicas in exactly the same order
 - If an operation Y (e.g., view shopping cart) starts after operation X (e.g., add item to cart) finishes according to some global clock, then Y should always see the result of X
- **Eventual consistency** is example of a weak consistency model
 - If an operation Y (e.g., view shopping cart) starts after operation X (e.g., add item to cart) finishes, then Y should see the result of operation X eventually
- Spectrum of consistency models from strong to weak
 - Example: causal consistency model says that same order of operations/inputs maintained only between operations that impact each other (e.g., operations on same shopping cart) and not across all operations

Handwritten notes: X: add item, Y: view cart (with arrow from X to Y), +

So, there are many different what are called consistency models, definitions of consistency, some are very strong definitions, and some are very weak definitions. For example, one consistency model is what is called atomic consistency. This is a very strong consistency model. This is what we intuitively mean, when we say consistency, which this model says that all the replicas must receive all the inputs in exactly the same order, and all input should be received at all replicas in the same order, your inputs can be a request to add items, delete items in a shopping cart, or in some other server, the billing orders, the shipping orders or the request to upload videos, whatever it is, whatever your system is, whatever kind of requests it is handling, all those requests should be executed at all replicas in the same order, then they will all be consistent.

And there is also the additional condition that if some operation Y starts after operation X, suppose your operation X is add item to a shopping cart, and your operation Y starts after operation X and this is view item or view the shopping cart. Suppose this is the scenario, the user has made operation X and then the user makes a request for operation Y and Y starts after X finishes according to some global clock, there is some clock somewhere and after access finish then request Y has then Y should always see the result of X that is, if you have added an item to a shopping cart, and then you view the shopping cart, that item should be visible in your shopping cart, that is what this very complicated definition of atomic consistency is saying, which is common sense, you do an operation and then you look at the system, that operation should be reflected in the system, that is what we expect. That is the definition of atomic consistency.

Now, there are also other consistency models, for example, there is something called eventual consistency, which is an example of a weak consistency model. In eventual consistency says that, if your operation Y starts after operation X finishes, then it is not necessary that Y should see the result of X immediately, it can see the result of operation X eventually, what does it mean?

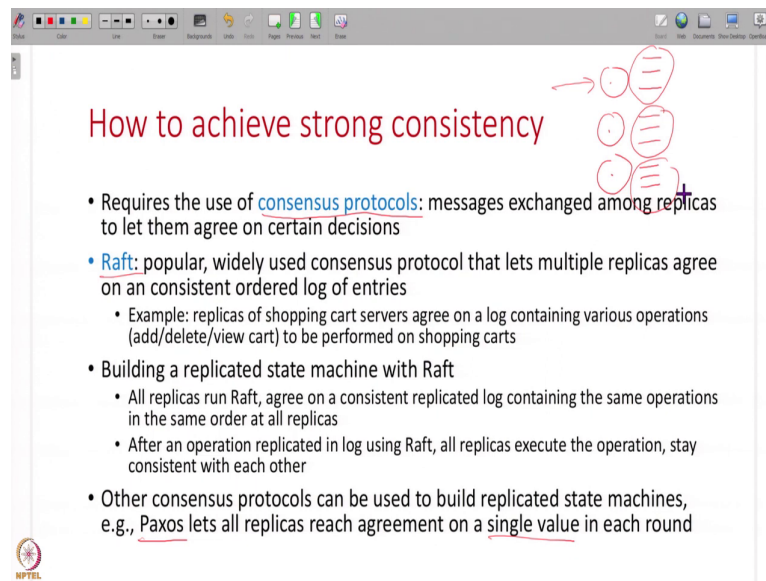
It means that, I do not need to see it immediately right away, if it is okay, if you show me after, a few seconds or a few minutes or something like that, after some delay is also okay. But eventually you should show it. What is eventually you do not know, it could be now it could be next second, next minute, tomorrow, next year, you do not know, the definition does not specify it, it simply says the system should be eventually consistent.

That is called eventual consistency; you can see that this is a much weaker guarantee, as opposed to atomic consistency, which is a much stronger consistency model. And of course, these are not the only two definitions, you have the entire spectrum, there are many other consistency models from weak, a little stronger, a little stronger to very strong and so on.

For example, there is something called causal consistency, which says that all the operations that impact each other, should be in the same order. For example, all the operations on the same shopping cart should be executed in the same order, but operations across different shopping cart, it s okay, if you jumble them, I do not care, you can kind of mix them and mix them up. And, it does not have to be fully consistent across all replicas.

So, in this way, there are many different consistency models that the theory that distributed systems theory has defined, and your system can choose to implement any of these models and accordingly how you replicate it will depend on whether you are trying to aim for strong consistency or you are okay to compromise for weaker consistency. So, now let us see how to achieve strong consistency. So, we will see examples of both we will see how you do replication to have strong consistency and how you do replication to have weak consistency. We will see examples of both in this lecture.

(Refer Slide Time: 07:48)



How to achieve strong consistency

- Requires the use of consensus protocols: messages exchanged among replicas to let them agree on certain decisions
- Raft: popular, widely used consensus protocol that lets multiple replicas agree on a consistent ordered log of entries
 - Example: replicas of shopping cart servers agree on a log containing various operations (add/delete/view cart) to be performed on shopping carts
- Building a replicated state machine with Raft
 - All replicas run Raft, agree on a consistent replicated log containing the same operations in the same order at all replicas
 - After an operation replicated in log using Raft, all replicas execute the operation, stay consistent with each other
- Other consensus protocols can be used to build replicated state machines, e.g., Paxos lets all replicas reach agreement on a single value in each round

So, for strong consistency, usually the protocols that are used are what are called consensus protocols. So, what does the word consensus mean in English, it means agreement, fully being in agreement with each other. So, these consensus protocols exchange messages among the replicas in such a way that all the replicas agree on the state of the system, and they are strongly consistent with each other.

So, very popular and in today's systems, a widely used consensus protocol is what is called the raft protocol. And what this protocol does is if there are multiple replicas, it let all the replicas agree on a log of entries. So, every replica has a log, and the same log is replicated consistently across all the replicas, all of them have the same view of the law.

So, what is this log? This log can be anything it can be for example, in a shopping cart server, the log can be all the operations to be performed, add an item, and delete an item, all the operations on your shopping carts that can be your log. And this raft protocol, if you run this raft protocol at all these replicas, it will ensure that all the replicas, it will exchange messages in such a way that all replicas see the same consistent log across all of them.

Now, once you have this log, it is easy to build a replicated state machine, all replicas start in the same initial state, whenever any request comes, that is put in the log it is replicated across all of them consistently and then that operation is executed. In this way all your replicas start with the same initial state, they get the same inputs from the log in the same order, and they will execute those inputs therefore they will all reach the same final state and your state machine, your replicated state machine is maintained consistently across all replicas.

So, using this raft, it is easy to see that once this raft protocol is there, it is easy to see that you can build any replicated system, any strong replicated system you can build by using this log as a building block. Now, there are also other consensus protocols. For example, there is another protocol that was an older protocol from Raft, Raft is sort of an improvement over Paxos, but Paxos was this older protocol that lets all replicas reach an agreement on a single value. It does not have this concept of a log, but it is used for single value. And of course, you can keep doing multiple iterations of Paxos to agree on multiple values.

So, in this way, there are many consensus protocols using which all the replicas agree on some set of things, and therefore they stay consistent with each other. And if you need strong consistency, you have to run some such consensus agreement protocol across your replicas. So, we will briefly now see how raft works. Of course, if you take a distributed systems course, you will study all of these consensus protocols in a lot of detail, but in this lecture, I want to just briefly touch upon the main ideas of how you achieve strong consistency using consensus protocols. So, what is the basic idea of raft?

(Refer Slide Time: 11:02)

Strong consistency: Raft (1)

- Basic idea of Raft: replicas exchange messages to maintain a consistent replicated log (same entry at same index in every log)
 - Replicas elect one leader, rest are followers
 - Leader receives inputs from clients, propagates to all replicas in the form of log entries
 - Once leader has replicated entry at majority of nodes, entry considered committed, applied to state machine, confirmation returned to client
 - What if majority of replicas cannot be contacted? No response to client
 - Example of a quorum protocol: contact a quorum before returning response
 - Raft instance with $2f+1$ replicas can tolerate up to f failures
 - Leader failure: followers detect via heartbeats, elect new leader, start new term (old leader can come back up and join as follower later)

At a high level, of course, all the replicas maintain a consistent replicated log, there is a log of entries, this log has the zeroeth entry is this, the first entry is this, the second entry is this. In this way, there is a set of a sequence of entries, this entry can be anything, it can be any operation, like add an item, delete an item, do this, do that whatever your entry can be anything.

And this log is replicated consistently across all the nodes in this system, raft will ensure that. So, how does raft work? If you have multiple replicas, one of these replicas is usually elected as the leader. And all the other replicas are followers; it has the concept of a leader and a follower. And this leader is nothing but it is one of the replicas is just elected by the replicas.

And what the leader will do is the leader will receive inputs from the clients and it will propagate those inputs to all the replicas, the user has add an item to a cart, this leader will add this operation to the logs of all the replicas and ensure that this operation is propagated into all the replicas in the form of a log entry.

So, anything the system has to do the leader will make it into a log entry added to the log tell it to all the replicas so that all of them add it to their logs. And once in this way any operation is replicated at a majority of the nodes then the leader will say, okay done, now, this entry is safe, it can be committed, it will be applied to the state machines and a confirmation will be returned to the client.

So, the important thing here is this, the majority of the nodes, so, suppose there are five nodes in the system. So, this leader will not just if it only replicates at one nodes or two nodes, then it will not send a response back to the client saying your operation is done, it will replicate at a majority of the nodes, at least three nodes, three out of these five nodes in the system have to have this entry in their replicated log and then apply this to their state machines and, completely process this request, only then a confirmation is returned to the client. What if it cannot contact the replicas, what if there are lot of failures in the system and it cannot replicate? Then it will not send any response to the client. So, either the value is replicated at a large number of nodes or the leader will say sorry, I cannot handle this request.

So, such protocols are examples of what are called quorum protocols. So, what is a quorum? A quorum is a subset of people who reach an agreement. That is the meaning of the English word quorum. So, raft is a quorum protocol, because it waits to contact a majority of nodes, build some agreement on a majority of nodes and then only send a confirmation that the request is handled.

And if it cannot reach a majority of the nodes, if there are $2f + 1$ replicas in your system, if it can contact at least $f + 1$ of them, that is a majority more than half if it can contact $f + 1$ systems, then rafter leader will say okay, fine, the operation is done, but if it cannot contact $f + 1$ systems, if there are more than f failures, then the system will not work properly that is it can tolerate up to f failures, but not more than f failures.

And of course, this leader also will keep changing periodically, the followers they have, they monitor each other via heartbeats and elect a new leader. And then somebody else if this leader fails temporarily, somebody else will become the leader. And then this old node can join as a follower again later on and so on.

So, in this way, all these replicas are working as either leaders or followers and every time a leader changes, it is called a term, a new term has started a new round has started in raft that is a terminology used. So, this is the basic idea. So, now how does raft guarantee consistency if nodes keep failing? All sorts of bad things can happen? So, how does raft guarantee consistency?

(Refer Slide Time: 15:16)

Strong consistency: Raft (2)

- Replica failures can cause logs to diverge
 - Some follower may have briefly crashed and missed a few log entries
 - Old leader of previous term can have some extra uncommitted entries that it did not manage to replicate before it crashed
- How does leader reconcile such logs?
 - When leader propagates entry k , it also mentions its entry " $k-1$ ". Follower updates entry " k " only if its entry " $k-1$ " matches with that of leader
 - If a follower's previous entry does not match, leader will rollback to the point where logs match and help follower catch up with all previous committed entries
 - Leader tries to sync all follower's logs to its own log
- Leader's log is the authoritative source, so it is crucial to elect good leader
 - All replicas vote for node with most up to date log (with all committed entries)
 - Leader elected successfully only if it gets majority of votes ($f+1$ out of $2f+1$)
 - Any two majorities always intersect, so at least one node with up-to-date log of previous term will be available to be elected as leader in next term

For example, when replicas fail, many bad things can happen, the logs can diverge. So, suppose here are all the replicas in the system and the leader is sending some updates, all these updates, the leader is sending to all the nodes. And suppose some node here failed briefly and it missed this update, it does not have this update then these other nodes got the update.

So, you could have some follower that crashed and missed some entries, it did not get a few log entries. So, the leader is replicating at a majority of the nodes, it cannot ensure that everything is replicated at all the nodes all the time, that is not possible failures keep happening. So, some followers may have missed some entries or some followers can have some extra entries.

How is that possible? One node was the leader in a term and it got a lot of entries, but before it could replicate them, this leader died. Now, all the other followers, they do not have all these entries, now one of them becomes a leader, then this guy's log is very short, whereas this guy who comes up later, the old leader becomes a follower.

Now, this guy has a very long log with all these uncommitted entries. So, in this way, the logs of the replicas can diverge either some followers can have fewer entries, some followers, which were previous leaders can have longer entries than the present leader, so these logs will diverge.

So, what does the leader do in such cases, the leader always reconciles all of these logs and basically will sync all the followers log so its own logs. So, the leader is the one who is actively maintaining consistency here, the leader will say, all of you listen to me, here is the log, you all please update your logs.

And the way it is done is whenever the leader is sending the k th entry of some say 10th entry of the log, it will also mentioned the previous entry, the $k - 1$ entry, and followers will update only if the previous entry also matches. For example, this leader here has replicated four entries to everybody but this follower here missed the third and fourth entries.

Now, when the leader is sending the fifth entry, it will also say here is the fourth entry if that matches, you install the fifth entry. So, at this guy, the fourth entry matches he will install, but at this guy, the follower will say no, my fourth entry is blank; I do not have the fourth entry. So, I will not install the fifth entry then the leader will say okay, how about do you have the fourth entry?

No, do you have the third entry? No., do you have the second entry, yes. So, the leader will identify, will go back identify at what points the logs are in sync and then it will keep on then it will say okay do this then do this then do this. So, it will roll back to the point where the logs match and it will help the follower catch up it will say, you have that now, here was the next one, here was the next one in this way all the previous committed entries will be told to this follower who has missed a few entries.

In this way, the leader is always trying to tell the followers whatever they have missed and similarly for follower has some extra entries that it did not commit and which have to be erased then such entries also the leader will roll back at the follower. So, basically leader is always ensuring that all the followers have the exact same log as itself, it is trying to replicate

an entry but also ensuring that not just that entry, but the entire log is consistent. So, the leader's log is basically what everybody will end up following. So, it is very important to ensure that you elect a good leader.

So, what if you elect a leader who does not have a lot of entries and somehow this suppose this node was only having two entries, it does not have any of these entries and now this guy becomes the leader and this guy will tell everybody, everybody you listen to me this is the log you remove off all your entries that is not right. Now, all these entries could have been committed confirmation sent to the user, you cannot always roll back entries as an how you wish that is not correct. Therefore, it is important to elect a good leader and who is a good leader? How do replicas vote for a good leader?

All the replicas will vote for the node that has the most up to date log. So, some replicas here that have missed a few log entries you will never make this guy the leader. You will always elect a leader who has the most up to date log who has not had failures, who is always up a lot of the times and has the most correct log, has all the entries.

Only such guy will be elected as a leader. This will ensure that when the leader is propagating its log to everybody else the correct log with all the entries is getting propagated, but not a log with incomplete entries. Then how is this guaranteed? How do you know that there is always a leader with the most up to date entry available for election? It is election time? What if you only have a bunch of stupid replicas who do not have all the log entries? What do you do? How do you elect a good leader?

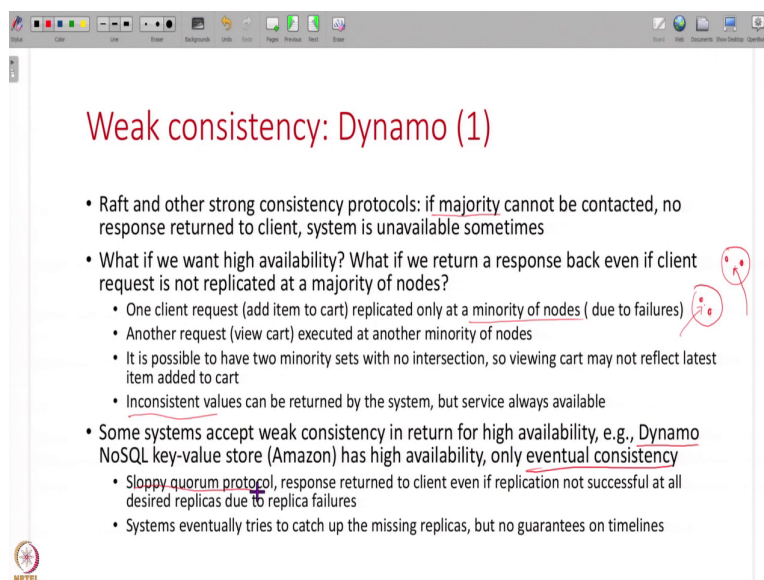
So, that is where the beauty of this protocol comes in; you have this notion of a majority, the leader election or anything committing an entry, anything requires a majority of votes. And in any two majorities, there will always be an intersection, you cannot have two majorities like this that do not have any common elements, this is not possible. Any time you take a majority, and at a later point to take another majority, there will be an intersection, for example, if you pick three out of five nodes, always and another three nodes you pick there will always be at least one node that is common.

So, now suppose this guy was the leader in one term, and some replica here was the follower, and it got all the entries, the next time this leader failed, another majority has voted and this old follower became the new leader, then there will at least be one node like this at the intersection of both these majorities who has seen all the updates from the previous majority, and gets a majority of the votes in the new term.

So, they will always be such a node, it will never be the case that whoever was there in the previous round with up to date entries, they are not available now to be elected as leader, every time an election is happening, there will at least be one node, who has seen all the updates, who was part of the majority in the previous term, has the most up to date log and therefore is available for election in this term and will be elected as the leader, because replicas vote for the guy who has the most up to date log and now this leader with the most up to date log will now propagate his updates to all the followers.

So, because any two majorities intersect, you always have a good leader available with the up to date log who will then ensure that this up to date log is what is continued across in the next round. So, this ensures that the raft always has good consistency, strong consistency properties.

(Refer Slide Time: 22:23)



Weak consistency: Dynamo (1)

- Raft and other strong consistency protocols: if majority cannot be contacted, no response returned to client, system is unavailable sometimes
- What if we want high availability? What if we return a response back even if client request is not replicated at a majority of nodes?
 - One client request (add item to cart) replicated only at a minority of nodes (due to failures)
 - Another request (view cart) executed at another minority of nodes
 - It is possible to have two minority sets with no intersection, so viewing cart may not reflect latest item added to cart
 - Inconsistent values can be returned by the system, but service always available
- Some systems accept weak consistency in return for high availability, e.g., Dynamo
 - Sloppy quorum protocol, response returned to client even if replication not successful at all desired replicas due to replica failures
 - Systems eventually tries to catch up the missing replicas, but no guarantees on timelines

And across all of these consistency protocols, this is always a feature if a majority cannot be contacted for some reason, you cannot replicate your log entry at a majority of the nodes then you will not return any response back to the client the system becomes unavailable, but you will never replicate at a minority and then go back and tell the client that the request is done why because in the future, then in this minority nobody may be available for the next term and some bad leader can get elected you do not want that. Now, then the question comes up sometimes I want high availability, I want my system to always return a response to me in spite of failures.

In such cases, what you will do is some systems then they will say we will accept weak consistency, and we will return a response even if we could only replicate it a minority of the nodes. Note that in such cases, you will not have strong consistency, you will either have strong consistency, but sometimes your system is unavailable if a majority of the nodes cannot be contacted or if you want high availability, you want your system to always respond back to you then you should be okay with replicating at only a minority of the nodes and sometimes you may not get consistent results.

So, for example, you can say that client request is only replicated at a minority of the nodes and I will return a response back. In such cases what will happen now, two minorities, you can have two minority sets, you can have two nodes here, you can have two nodes here out of five nodes, where these two nodes and these two nodes do not intersect, and there is no common node that can happen.

So, you added an item to the shopping cart and it was replicated at these two nodes then these two nodes failed, then you view the shopping cart your request goes to these two nodes. And now what do you see, whatever item you added is not available in these nodes right it was never replicated at a majority of the nodes. So, it is possible for you to see inconsistent values, it is rare the system will try to ensure that everything is replicated everywhere it will try to ensure consistency but it is not guaranteed.

So, there are some systems for example, Amazon has a key value store called Dynamo, which has high availability, which will always return a response but it may not be consistent all the time, you only have eventual consistency. That is, you will try to catch up all the replica so these two guys if the other replicas have failed, they will try to tell them about this adding the item to the shopping cart later on, but there are no guarantees on timelines eventually you will be consistent.

So, such protocols are also called sloppy quorum protocols you are doing a quorum, you are talking to a subset of the nodes to propagate the decision and replicate the operation and so on, but you are being a bit sloppy about it, okay if I cannot contact everybody, so be it, but I will return a response back to my client. So, this is an example of eventual consistency, weak consistency, but high availability design.

(Refer Slide Time: 25:17)

Weak consistency: Dynamo (2)

- Systems with weak consistency can have conflicting values of application state
 - Shopping cart of user has items A, B
 - User adds item C, replicated only at a minority of nodes (due to failures)
 - User adds item D, replicated at a different minority of nodes
 - When user views cart, can get back "A, B, C" or "A, B, D" or both
 - Note that this can never happen with Raft: at least one node will have seen both updates, as any two majorities will intersect
- Application can decide how to handle inconsistent values
 - Merge shopping carts to have superset of all items "A, B, C, D" +
 - Trickier to merge two different versions of bank accounts

So, note that systems with weak consistency, they will return, they can sometimes return conflicting values of your application state as I was telling you the example earlier. So, suppose your shopping cart has items A and B, all the nodes in your system, have the shopping cart A and B.

Now, if you add an item C to a shopping cart, it can only reach a minority of the nodes. Okay fine, the system will still respond back to you saying item C has been added. Now, you add another item D to your shopping cart, it did not reach, all these nodes were down then it reached some other set of nodes and they added the item D to the shopping cart.

Now, what is happening when you view your shopping cart if you talk to these nodes, they will tell you your shopping cart has items ABC, you talk to these nodes they will tell you your shopping cart have items ABD which is correct, you do not know both are valid options. Note that this would have never happened with something like raft which guarantees strong consistency, why?

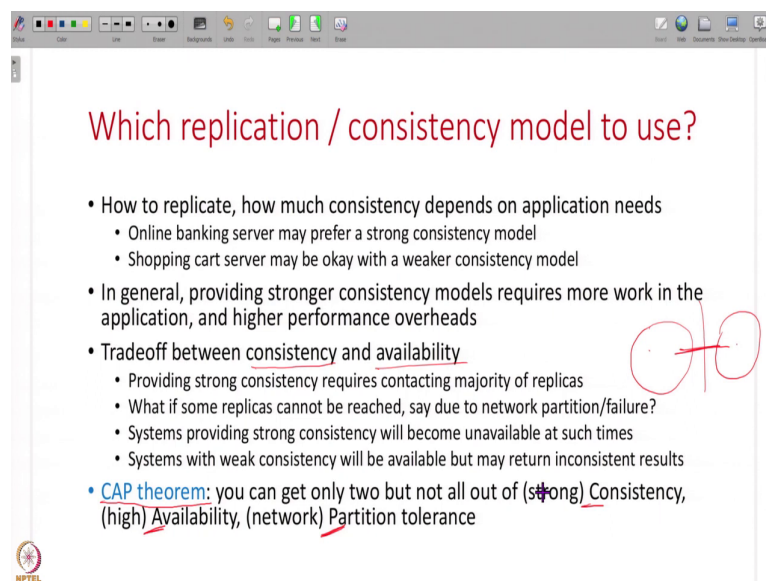
Because between these two sets this ABC and ABD said there will be at least one common node that has seen A, B and the C update and which will see the D update, there will at least be one node that will intersect when you add an item C you will replicate at majority, when you add item D you will replicate it majority there will be some intersection.

Therefore, you cannot have these divergent values your system will have consistent values and this node that is the most up to date node will be the leader it will ensure that this update C is propagated to the next term also and your shopping cart will have ABCD.

But if you are using a weak consistency protocol that will return a response back even if a minority of the nodes have been contacted then you can get two different values then what do you do? So, this Dynamo was developed incidently to handle to manage shopping carts at Amazon and here they say that with shopping cart this is acceptable sometimes an item is missing in your shopping cart, it is okay the user will not be terribly upset. And if you get two different shopping carts like this ABC, ABD then the system can even merge them it can add up all the items and say your shopping cart is ABCD.

It is easy to handle inconsistent values for some applications like shopping carts, but for some applications like bank accounts, your bank balances either X rupees or Y rupees we do not know which, will you be happy? No, you will be terribly upset. So, for certain applications, this inconsistent value, I am returning to all possible values you do what you want that approach will not work. So, which replication and consistency model will you use, it will heavily depend on the application's needs.

(Refer Slide Time: 27:56)



Which replication / consistency model to use?

- How to replicate, how much consistency depends on application needs
 - Online banking server may prefer a strong consistency model
 - Shopping cart server may be okay with a weaker consistency model
- In general, providing stronger consistency models requires more work in the application, and higher performance overheads
- Tradeoff between consistency and availability
 - Providing strong consistency requires contacting majority of replicas
 - What if some replicas cannot be reached, say due to network partition/failure?
 - Systems providing strong consistency will become unavailable at such times
 - Systems with weak consistency will be available but may return inconsistent results
- CAP theorem: you can get only two but not all out of (~~strong~~) Consistency, (high) Availability, (network) Partition tolerance

So, some applications need to have a strong consistency model and therefore, they will use a consensus protocol like raft or Paxos that is more work it involves more overhead, but they will use that in order to guarantee strong consistency. Whereas some applications are okay with a weaker consistency model then they will do a sloppy quorum kind of protocol and they will accept a weaker consistency in return for better performance of the system. So, what is the trade off here? The trade off is between consistency and availability. If you are providing strong consistency sometimes when you cannot contact a majority of nodes, you will have to be unavailable you will have to say sorry, I cannot handle this request.

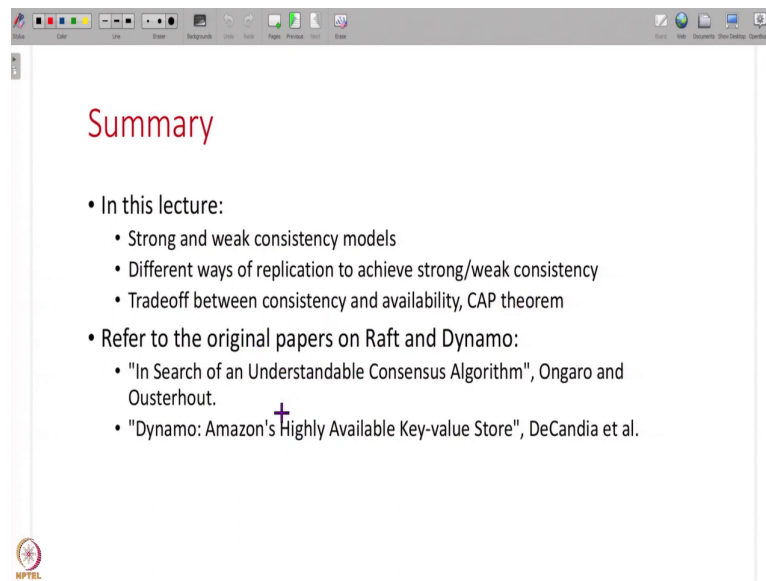
So, sometimes there are something like network partition, you cannot, your network splits into two parts, there is some failure of some link and your network becomes two parts and the nodes here cannot contact the nodes here. When such things happen, you if you cannot contact a majority of the nodes you will say sorry, I cannot handle this request you become unavailable. But systems with weak consistency they will be available even if there is a network partition, there is a network failure you cannot contact everybody they will be available, but sometimes they might return inconsistent results.

So, there is this very famous theorem called CAP theorem, if you take a distributed systems class, you will learn more about it in a lot more detail. But the basic idea is fairly intuitive, there are three properties there is consistency, which is all the application state is the same no matter which replica you look at, then there is availability, which is your system always handles your request and gives you a response back it does not say I cannot handle your request that is availability, then there is partition tolerance. That is if your network fails, you cannot contact some nodes and all of that your system is still working that is partition tolerance.

So, the CAP theorem says that you can only get any two out of these but you cannot get all you cannot have it all, you have to make your trade offs. If consistency is very important to you and of course network partitions will keep happening in real life, then you cannot get availability, you pick a design where you are strongly consistent, but there is no availability all the time. Or if you say I do not care so much about consistency then you always return a response, you are highly available, but sometimes you will return some inconsistent values from your system.

But you cannot have a fully consistent system that is available all the time, and yet is tolerant to network partitions and network failures, all these three things at a time you cannot get. And of course, if you take a distributed systems course, there will be a detailed discussion of this, but I hope that this discussion gave you a high-level picture of why the CAP theorem is true in real life.

(Refer Slide Time: 30:52)



Summary

- In this lecture:
 - Strong and weak consistency models
 - Different ways of replication to achieve strong/weak consistency
 - Tradeoff between consistency and availability, CAP theorem
- Refer to the original papers on Raft and Dynamo:
 - "In Search of an Understandable Consensus Algorithm", Ongaro and Ousterhout.
 - "Dynamo: Amazon's Highly Available Key-value Store", DeCandia et al.

So, this is the summary, in this lecture we have studied what is consistency, what is strong consistency, what is weak consistency? Then we have seen what are the different ways of doing replications to achieve strong consistency and weak consistency, and we have discussed this tradeoff between consistency and availability in the CAP theorem.

So, as extra homework, you can actually go back and read these original papers, the original Raft and Dynamo papers, which talk about strong consistency and weak consistency. Of course, not all the details of the papers can be understandable by you at this point, but you will get a high-level picture of how you achieve strong consistency and how you build systems with weaker consistency models. So, that is all I have for this lecture, let us continue our discussion in the next lecture.