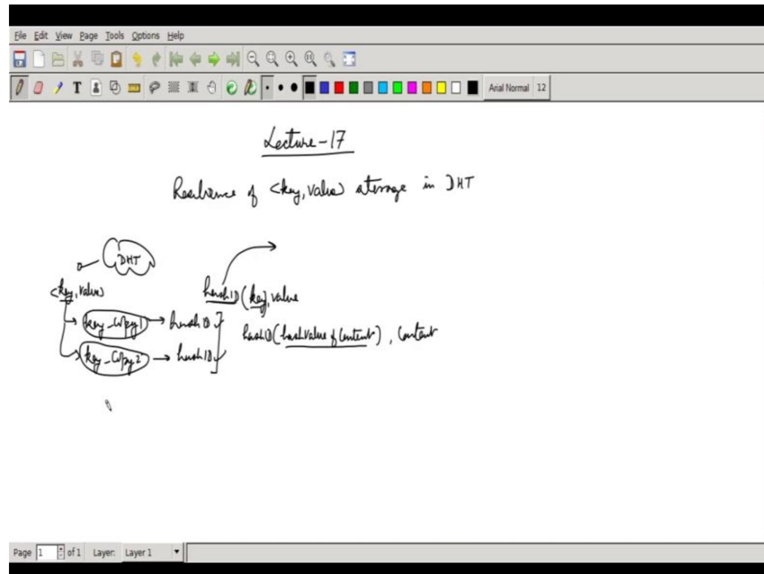
Peer to Peer Networks

Professor Y.N. Singh
Department of Electrical Engineering
Indian Institute of Technology, Kanpur

Lecture 17

Resilience of <Key, Value> Pairs

(Refer Slide Time: 0:14)

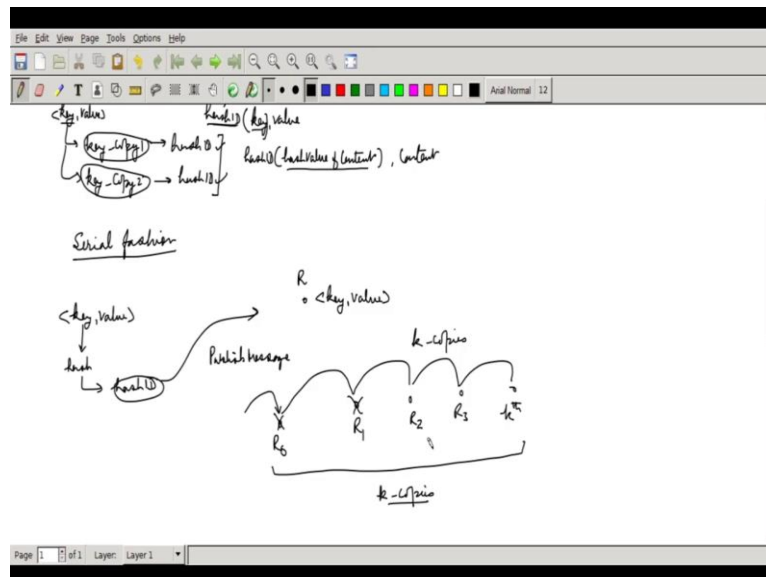


So, welcome to lecture number 17 of Peer-to-Peer Networks noc. In this lecture, we will be looking at the resilience of key-value pair storage in a DHT network. In the previous lecture, we looked at how a source, when he is injecting into a DHT network a key-value pair, can take the key and then define that key copy 1 and compute the hash of this from their one hash ID.

And he can also compute another hash ID by assuming that it is going to be key copy 2. And from there, get another hash ID, which is how we can have multiple root nodes for the same key value. So, multiple root nodes will be maintained in the same key-value pair, and that time, we had talked about that the, it will be the key value which will be there or there can be the hash ID of the hash value of the content.

So this will be used as a key for the content, or we can define the keys, and the content will be there concerning this. Similarly, for this also again the key, we will be finding out the hash ID from here by doing the hashing, and this hash ID (whoever) whichever is the root node will be maintained in the value. We can have multiple root nodes and monitor each other and keep on (publishing) republishing the entries if any node dies. So that was a parallel thing.

(Refer Slide Time: 2:25)



There is another possibility which we can explore. We call it a serial base, serial duplication. So, this resilience is coming because of the periodization essentially. We are not going to maintain copy 1, copy 2. Whatever the key value is there, so I want to store it, take the key and compute the hash of this. As a consequence, I will get a hash ID.

Then I will use this hash ID to find out the root. I will send a published message. So this will ultimately turn up to a, at the root node, where the key and value pair must be stored. Whenever somebody wants to search again, he will do the same procedure and make a query here. But the problem is this root node can die off. So who is, how this is going to be recovered.

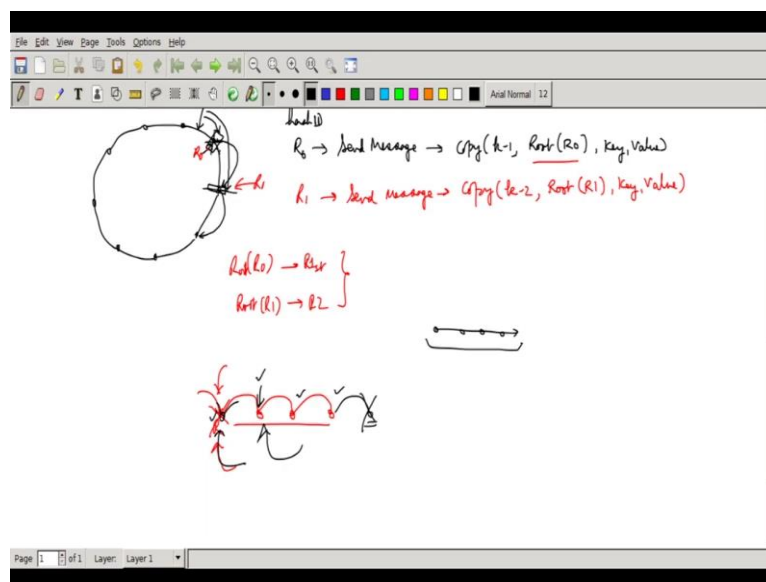
So we started with this question last time. So the way this can be handled is, this root node, this root node can now find out among his neighbours. There are two possibilities. It somehow knows that if this node dies off, some other node will become the root. So if I call it a root 0th root, this I can call as a first root. Even this both die off, then there will be a third node, who will become the third node, who will become the root.

I can call it 0th root, first root, second root and so on. So I can have, in some way, I can have k^{th} root also; this is R3. So, what I can do is the first copy comes here. This guy somehow, if it can figure out it can maintain the copies at all these places. So it can maintain k copies if it

wishes. Whenever the root node dies off, the query will automatically go to another place where the key-value pair will always be there.

So I can maintain k copies for each key, each key-value pair that could be one possible straightway of handling the situation. Now, this actually can be very quickly done if it is a chord kind of system. So, when we were implementing Brihaspati and thinking of chords, we thought this could be an excellent way to do things. So we started with this, but later on, we have changed it because we are using a different algorithm, not the code.

(Refer Slide Time: 5:00)



So what happens in a chord? So in the chord, all nodes are arranged in a ring. And it is an asymmetric distance which we are following. If the hash ID is pair, whoever is the next node, whoever is the closest node from hash to the node ID. So whichever has the least distance, so this guy will become the root node in this case, but suppose this root node dies off, in this case automatically the next person will be this.

Now there is an exciting property; if I look at this particular node ID, this node ID is the root node will be this guy. If this guy has a root node, I have to figure out; this will be this person. It will be pretty simple that if I have the root node for a hash ID, I can send a message and this message I can call a copy. And I can also even identify which particular, how many copies have to be created. I say there is already a copy, so k minus 1 further has to be created.

What will be the destination of this message? The destination of this message will be the root of R itself. I (call), I should call it R0 and then whatever is the key-value pair which I want to store that will go. So this will be received by R1, which is the next guy if this is the root node, so this is going to be received by this person so which is R1 now.

So when R1 receives it, R1 will also now send the same message. It will keep the same copy, and it will say again it will be a copy, but now it will reduce the number. Further, only k minus 2 copies get to be created. The destination where the message has to go has to be the root of R1 now. So this exciting property that R1st root is the root of R0 and similarly root of R1 will be R2 and so on. This property gets always exists in a chord system.

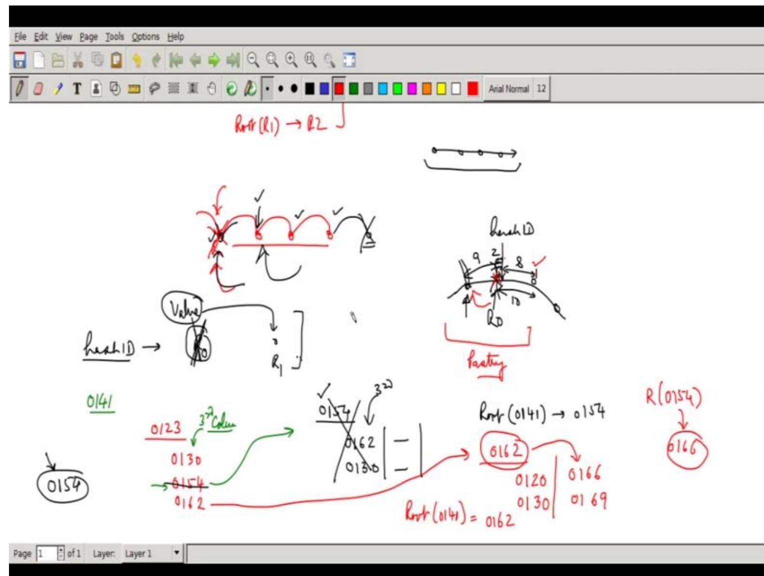
So this is actually can be very quickly done. So this way, I can maintain k copies. Now what we can do is that each one of these entries. So you have one entry here. There will be now entries that are going to be created. So let k be equal to 4. In this case, there are four entries. So this guy has a root node. So all these guys also will keep on republishing the entry. So whenever they republish the entry, it will always land up with this particular root node.

Periodically they can all do it. Every time it comes here, this root node's responsibility whenever it gets a message a republished thing, and there is a timer, before that it has to send this copy message again. So this overhead of the protocol will be required to maintain the k copy. The moment it dies off so the queries will automatically come to this other node. So the queries will come to this particular node.

So whenever there will be, these guys will be republishing over this itself will be republishing, this will be coming back to again terminating to this particular node only. If this node comes back again, it is alive, so this republishing will be happening and will be terminating here, and then again, it will create the k copies. So the last guy who is, who was the fourth copy will ultimately it will not be receiving the republish.

When the copies are being copied, a person will not receive messages; it will be removed. The next k node will always be holding on to the copy automatically, whichever is the root. So this is a straightforward thing that can be done with a chord algorithm. Now the question is, is this logic is also correct for even other algorithms or not.

(Refer Slide Time: 9:15)



For example, if for a hash ID if I look at tapestry for example, for a hash ID if I find that there is going to be a root node is R. So if this guy dies off, for whatever values which is it is holding for those values will the root node will be the next person or not. In chord this (auto), this is always ensured. If this guy dies off all the entries held by this node for those entries, this person will be the root node.

So in the case of a tapestry or a pastry, it is possible that if my root this R dies off, next, whatever is the root node for this R, R0 so this is R1. Will R1 will be the root node for all the values held by R0. This not true. This is only true for chord case, but not right for pastry or tapestry. So I can give a straightforward example of this and show that this is not possible in pastry or tapestry.

For example, in the case of a tapestry, there are nodes held in this fashion. So you have a hash ID which is going to come here. So in the case of tapestry, it is always a (close) numerical closeness. So this is a numerically close thing, so this will be the root node, this will be R0. So next, we can figure out a node that is going to be closer. So I can make some minor change here, and I can represent by a distance. So this is, for example, say 10 is the total distance here.

So this is 8, and this is 2. So you may have a distance here, which is going to be at say 9 distance away. This is 9 from this guy. So for this R0, the root node will be this person, but

for hash ID, when you will, when this guy dies off, the root node will be, so when this person dies off. So from the hash ID, this will be 8 distance away, and this will be 11 distance away, so this will be the root node. While for R0, this was the root node.

So, henceforth the earlier logic, which is right for chord, is not valid for pastry. Similarly, I can show that even for the tapestry algorithm, this will not be true. So let us take a case that there is a node called 0, 1, 2, 3, I am again taking a 4 digit system, and this has a table, its table 0 1 3 0, it has 0 1 5 4, it has 0 1 6 0. And the hash ID which is in question with me is, which I am trying to search it is a 0 1 4 1.

So how, what will be happening in this case. So it will now be 0 1 4 1, so 0 1 matches so that it will search for this particular entry. This happens to be the third column because 0 1 matches this and has to be the third column; only third entries change here. So we will now try to find out 3 4 5 6. So it has to be this particular entry where the query has to be handed over.

So it will come to 0 1 5 4, so 0 1 5 4 will have an entry say 0 1 6 2, 0 1 3 0, that is the third column entry actually and it also can have, it may not have a fourth entry there is no fourth entry in this case. So there is no entry with 0 1 5 onward; there is no entry. So, when it now searches for this thing, it will find out it is the 4; the third digit here lies between 3 and 5.

So this itself should be a root node. The root node for 0 1 4 1 will be 0 1 5 4. Now 0 15 4 dies, then what will happen? Let us see. So 0 1 2 3 contains the 0 1 5 4. So, 0 1 5 4 is dead. So once this is dead, what will be the entry? There is no entry here; it will now search for 3 and 6. So let me figure out the root node for 0 1 5 4. I need to find out whether the root node of 0 1 5 4 will be the new root node of 0 1 4 1.

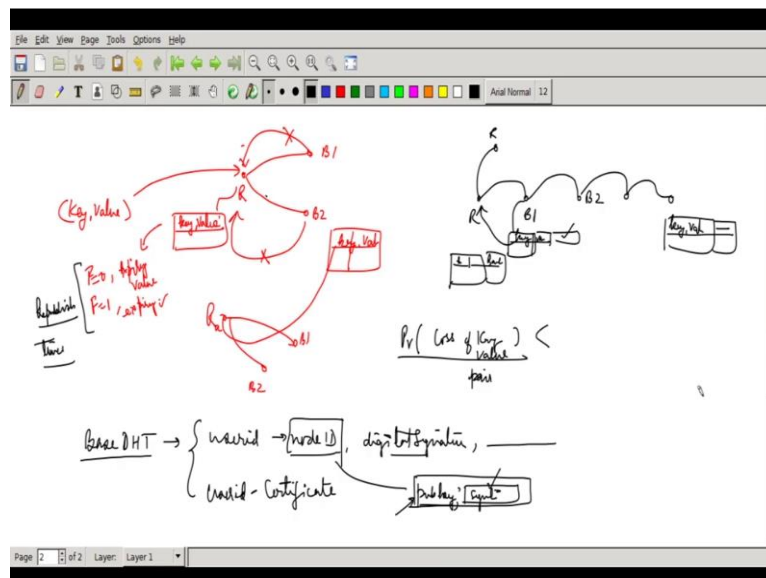
When I look for 0 1 5 4 in this new entry, this entry is all gone. 0 1 5 4 no more exists because it is dead. So 0 1 5 4 when I will search it will go to 0 1 6 0. So 0 1 6 in fact, I should call it 0 1 6 2 in my example I have kept, so I am going to keep it 0 1 6 2. So 0 1 6 2 if I create a table, so 0 1 6 2 can have entry 0 1 2 0, 0 1 3 0, for example, it can be there. In this case, it will find out that 0 1 5 4 if it comes to 0 1 6 2, this can further have an entry 0 1 6 6, 0 1 6 9.

So 0 1 6 2 when we will try to find out 0 1 5 4. It will find that this matches this itself will be root node responsible, so it has to move to the next column and 0 1 5 4. Now 4 lies between 2 and 6 so it will be handed over to the query from here from this 162 will come to all the way here. This guy will now be handling the query to 0 1 6 6. So 0 1 6 6 will become the root node.

And it has an entry such that this is only where it will get terminated. So this will become the root node for 0 1 5 4. I will now look for what happens with 0 1 4 1, because 0 1 4 1 will come here it will also go to 0 1 6 2. So once it goes to 0 1 6 2, it will also reach here at the same place. So once it comes here 0 1 4 1, we will find out it matches 0 1 6 2.

When it looks at 0 1 6 2, it goes to the last one because there is no other entry. And here, where there is 1 will match, 1 will match, so 0 1 6 2 will be the root node in this case. So the root of 0 1 4 1 will be 0 1 6 2. The root of 0 1 5 4 will be 0 1 6 6; because of the last digit variation, this will happen. So again, the same logic which was right in the chord is not valid in tapestry also. So I cannot use this method.

(Refer Slide Time: 17:05)



So we have to now choose for some alternative approach, so an alternative approach which can work even in other kinds of DHT algorithms will be one possibility that your key-value pair is sent to a root node. This R has to now search for from its routing table. It will find out some other nodes which can act as a backup. So it can create two backups. It is a possibility.

That is one possibility to get two parallel backups, and it will send the copies to them, so they will hold the copy and hold the copy. And their job is to monitor R. So R continuously maintains this key-value pair here in its database and mostly serves with that. But B1 and B2 will periodically monitor if R is alive or not alive. The moment R is dead, they cannot get a heartbeat from R. They will republish the entry.

The same key-value pair will be republished, and the moment this is republished, it will go to another root node, which is an alternative that will find out their B1s and B2 backups will be maintaining the copy, and they will be monitoring. Now this entry should practically every time republish happens. This entry has to go back to the R. R, in turn, will send this entry back here or reset the timer, expiry timer for B2 entry.

So backup entries are going to expire and purged. Here key-value pair this, that expiry is different, this will be again going to have the same flag F is equal to 0 or 1, whether it is a final kind of expiry or it is an expiry will be reset of with every access and then the expiry value which will be there. Expiry timer maximum value is the fixed expiry that will expire when F is equal to 1, when it is final.

This expiry value will always be reset whenever it is going to be accessed. So this is one way it can be done. Another alternative is that when the root node gets the information, it sends to the backup node and tells him that you are the backup one; it maintains the entry and monitors this guy. So whenever this guy fails, it will do the republish. B1 also finds out somebody and say that you are the backup for me.

So they need to maintain, and this guy needs to know where my backup is there actually, for this one, where is the backup, for every key-value thing. It also now creates a backup, which the guy is maintaining. So when this guy maintains an entry for key-value pair as a backup entry, it will also maintain a backup. We can create any number of backups. The last guy will not in the table have key-value pairs.

But the backup entry will be null in this case, so there is no more backup. Only these are maintained. So, each one of them is supposed to keep on monitoring R. So all of them will not fail, all of them will publish monitor R, and all of them will republish if the R fails. So the

chances that all of these will fail is very low. So we have to, we can only, we cannot give a guarantee.

We can only say the probability that things will be lost can be arbitrarily low. Lower the value you want for this probability of loss of key-value pair, so the higher the number of replicas you need to create. So this is. This is a serial-parallel combination; this is a serial, serial combination that actually can be used. So, this is how the resilience actually can get created, and of course, whenever the R's routing table is going to change, and it can find out (someone) somebody else is going to be a better root node that will, anyway, will be happening because of the timer, republished timer which is here.

Remember, republished timer and expiry timer are two different things. There is also going to be a republished timer. So we should not confuse between these two. So R will also actually keep on doing the republication. If R finds out some other guy has become a better root node, it will stop sending this whenever this republished timer expires; it will not reset the timers with all these things.

So they will automatically get purged, and this new root node has to create its backup chain. It will but retain value for some time before it purges the entry form here. If it dies off, the request again will start coming, so it need not learn again. So it will still maintain even after it is transferred the value, it will maintain it for some time for resilience purpose. So that is how essentially the resilience gets created. So far, what we know is that DHT is pretty work, works without any issue; the only problem is the resilience we have created.

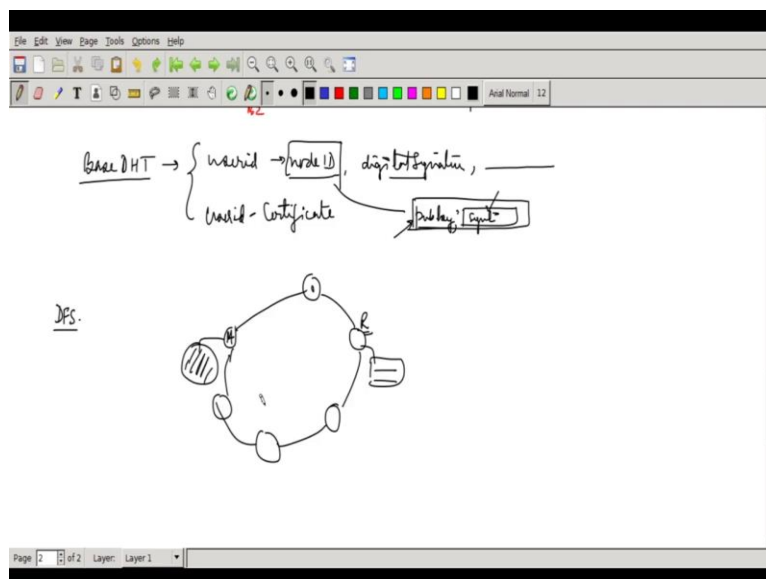
In the base layer, base DHT layer, usually, this is not the issue. The base layer you usually will be either maintaining a user ID to node ID mapping, this will be again digitally signed by user ID so that you will have a digital signature and the same user can be there on multiple nodes also so there can be, you can put a certificate here, but we need not. Digital signatures are good enough, and we can also now hold on to the user ID to the certificate mapping for the user.

So node ID can always see a going to be a hash which is going to be, the node ID is always a couplet where the public key of the node ID is the node ID. The corresponding private key signs the public key part so that signatures will be there. So that is what is going to be a total

node ID. So this will ensure it is always randomly generated; otherwise, you cannot generate this particular, and hash will not be. You will not be able to generate unless you have a corresponding private key.

And a private and public key exist, they have to be randomly generated. So you cannot build a in this case. So this we had discussed earlier also. Only these two entries will be there. So roughly for each node, we have only one, two entries on average. When the base DHT, you will not be running out of memory because these are very small data structures, can be maintained by everybody.

(Refer Slide Time: 23:24)



You will never be running out of memory, but I want to create a distributed file system when we go to an application. So every node is maintaining some storage where the file fragments are held further. So every node does it. Every node allows every user to have this own file system space whose file fragments are stored in this DHT network. It is a DFS; we call it a storage DHT.

The nodes which are participating in storage DHT, they can run short of space. Now they are not doing only the base, in a base layer like we have only two possible entries they can have files, they can put the movie, they can put all kind of system so people can run out of space so how to handle that situation where this is the root node for something, but it does not have space and since the root node is there so queries will always come here. So it has to take the

help of other people to find out the storage and use it, but why somebody else is going to donate their storage.

So, I think the logical reason for this is that if you donate the storage chances, you will get the storage should be higher. So it becomes a kind of a tit for tat mechanism because of which the free-riding will not be handling. Freeriding implies that you do not want to share any storage of yours, but you take others' storage. So we can avoid that, so in the next video, we will be looking at this particular algorithm, and we will start with the discussion on DFS and how this algorithm is actually can be built to ensure that there is no free-riding in the system.