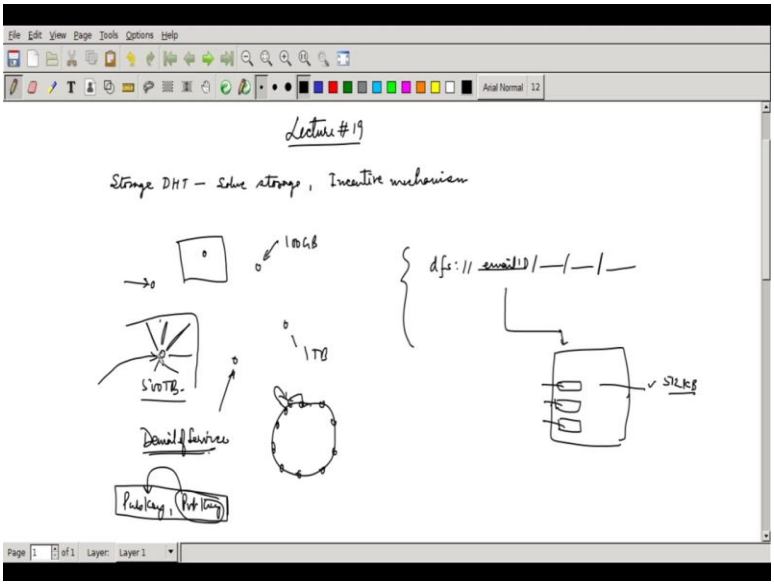
uPeer to Peer Networks
Professor. Y. N. Singh
Department of Electrical Engineering
Indian Institute of Technology, Kanpur
Lecture No. 19
Storage Space Problem and Incentives to Share Storage

(Refer Slide Time: 0:13)



This is lecture number 19. In this lecture, we will discuss distributed file systems being created through peer to peer mechanism. So, the basis of how the DFS will be made was discussed in the earlier lecture. But for a distributed file system, we need storage space. In the previous lecture, there was an assumption that all nodes will be contributing to the storage space equally, but that is not proved.

Sometimes a node may be there, but it will not have a sufficient amount of storage space, so what should happen in that case? How to handle this kind of situation? The technique which I will be describing here is very specific to Brihaspati 4 system. We create a storage space, storage DHT, so all the nodes contributing the storage form their own DHT network, and we call it a storage DHT.

So, there is another layer that gets formed in the multilayer DHT system of Brihaspati 4. So, we solve storage, but the people who contribute to the storage need to get some incentives. We will also be talking about incentive mechanisms to encourage people to contribute to the storage; they should get something instead.

So the idea in this session is that if you contribute more storage space to the community, more storage will be provided to your community's file system. So, if you contribute less, you will get a less share out of the total cooperatively created storage system. So, let us see how this can be done.

If there is no storage DHT and nodes are participating, what can happen in this case? So, most likely, what will happen is everybody is now creating a file system, they will be making DFS a kind of two values will be in this fashion, there will be email, this is the same way as I have discussed here.

Then, of course, directory, subdirectory, file name and then, of course, this will be creating a venue search for this key, you will get a descriptor file, and from there, you will get the multiple fragments. Each fragment's hash ID, a hash of the content, will be there, and you will use this hash to search for where the file is going to be there. And then we will have then copy one or copy 2, copy 3 of each of them getting created.

But since all these, when computing hash IDs, they will be uniformly distributed in the whole hash ID space. And since these nodes are also uniformly distributed. So, everybody is supposed to get almost the same kind of storage requirement. Each fragment is of fixed size, 510 kilobytes or 1000 kilobytes, 24 kilobytes. So, everybody will get the same share of storage requirement, so everybody should equally contribute.

If all machines are of equal capacity, then, of course, there is not much of an issue. But since we do not expect everybody to have equal capacity, what has to be done? So, somebody has a very lower storage space here, having 100 GB only, somebody provides 1 Terabyte, somebody will have 500 Terabyte, so how will we handle this kind of situation?

One possibility is that the hash ID is for which your node is responsible, the number of hash ID's in this area for which this node is responsible, this volume of the hash ID space which is closest to this or for which this is a root node, that can we can make different that actually. But will that be feasible? Or if you look into the chord system, all the nodes are equally spaced almost, statistically.

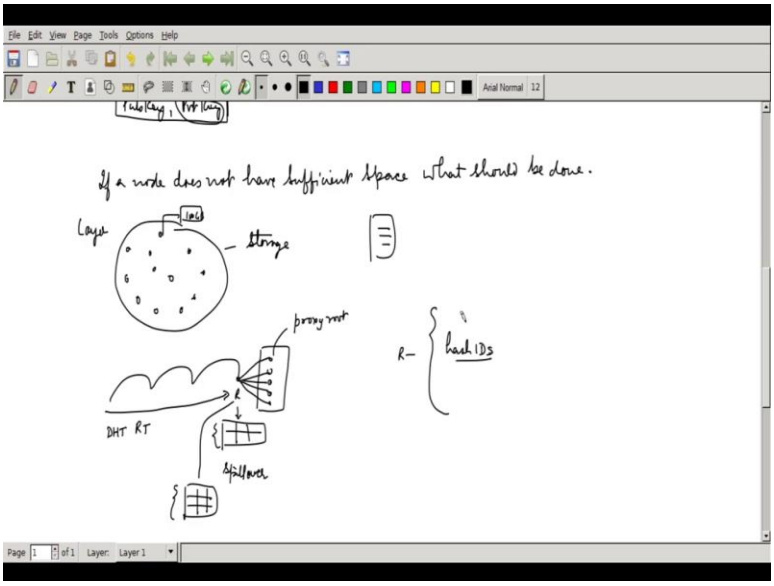
I am telling you that, for only some very few entries because this is lower storage space, very few entries should come here. So, I am saying that this node should be closer to this previous

node, so let very few entries will only be stored here. So, their spacing has to be dynamically adjusted. With different amounts of key-value pairs or file fragments, different amounts of file fragments will be stored at different nodes, which I will be matching with whatever storage they can provide.

But that is not feasible; the reason being that node IDs are not being cannot be explicitly configured. They are always randomly generated; we did that so that we can avoid denial of service attacks. We had provided this in one of the earlier lectures, so we wanted to take care of this, so that is why we started with a random generation; we started that node ID will be the public key. And the corresponding private key and this public key will be generated together. So, this can only happen to a random generation process.

And this private key will be used to sign this one so that we can prove it is indeed generated through the same process. And because of which it is a random key, and this private key is always maintained at the node so that it can actually for anything where node ID has to be proven, to be held by that node can be proved by this private key. This mechanism whereby I can have a different hash ID is rooted at one particular node depending on the storage being not feasible in the Brihaspati 4 system. So, we have to do away with that.

(Refer Slide Time: 6:18)



So, what could be the alternative if we cannot do this? We have T to have an alternative approach. So, the question is, if a node does not have sufficient space, what is it going to do? So,

if a node does not have sufficient space, what should be done? We do that everybody will have the storage space, so we create the nodes that can allow one to participate in the storage system.

They will now create a separate layer, a separate DHT layer. And those node IDs will have their routing table, so only the neighbourhoods in this people, in this set of nodes that are participating in providing storage service so that this node will maintain routing table to the nodes present in this particular layer. Of course, the base layer will also be present, but the base layer will mostly look into user ID to certificate or user ID to node ID mapping to maintain those.

But similarly, we will have this layer also storage, but this now will be providing storage. So, anything which has to do with the file system, like this the file fragments, universal file system part, where the content hash ID, the content hash will be used to find out the content. And using this key in file namespace of a specific user, specific file pointer, based on that finding out its descriptor and then, of course, the file fragments and treating them.

So, all these detailed data will be kept in this storage DHT. Now, in this case, it is possible that some nodes may not have sufficient space; they have contributed, this guy has only contributed, say 100 GB, and it gets filled up. Usually, everybody will get equally filled up as the storage comes into use because even if you put up a one single 1 GB file, it will get fragmented, and it will be scattered around across the whole storage DHT. Statistically, everybody will be filling up at the same rate, so the guy who has got the lowest sees this 100 GB is the lowest storage and will get filled up first.

Now what to do in this case? So, there will still be a request to come here, so where they need to go? Now we have devised a new mechanism; we call it a spillover table. So, there will be hash IDs that will go and then root it through DHT and which will go to a root node. So, once the key-value pair the query reaches here or a public request reaches here, this guy says I do not have a sufficient space; what has to be done?

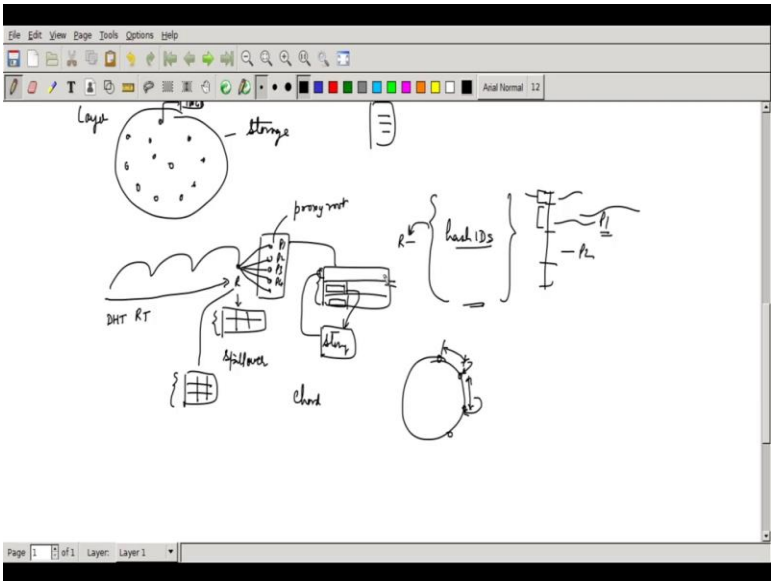
It will now actually look into its routing table, the routing table of this particular so as DHT layer. We will now send a query to various nodes and ask them that I am running short of space. Can anybody be willing to give me space? Now that is where this currently. Let us assume that everybody is willing to contribute and sort out the issue; nobody is free riding.

So, R is not doing the cheating. It has always filled up space. So, everybody is honest; let us assume that. Later on, we will come to a reputation mechanism by which no there is no incentive and cheating. So, the algorithm is specific to our system. Once these extra nodes find out, we call them proxy root because they will act as a proxy root for the entries supposed to be held by this root node.

So, then act as a proxy layer. And let us see what is going to happen in this case. So, now this node, root node, whenever the query will come, we will have to keep track now whether the key-value pair is with me or with a proxy unit. Now, which proxy root is going to have it? So, at this point, it is going to be through routing tables created through the DHT network, through that the routing will happen. Once it reaches the root, the routing will be happening through a spillover table because space has been filled.

Now the query is being spilled over to the place to a node which is having space. For every, now the question is for which all key-value pair it should put the values here? All the terminating requests need not go here; some of them will be there in the current indexing table or key-value pairs in indexing in the storage locally. And remaining has to be put there, but how will that be done? So, we have to find out this root node is responsible for which all hash IDs. For which all hash IDs it is responsible.

(Refer Slide Time: 11:43)



So, when R is sitting there for any DHT network, R actually if depending on by looking at its routing table, it can figure out which all hash IDs it is the root node. Now, these hash IDs for which this will be the root node when they are continuous or not? So, in the chord, if you look,

the chord table if you make it, if you have nodes here, you can see this is a continuous block for which this is the root node.

This is a continuous block for which this is the root node. This is the other fragmented, so I actually can know now, well I do not have space, I have some other proxy root so that I can find out P1 P2 P3, depending on whatever is a requirement it should send the message, and some people will come back. They will say this is your storage, which I can provide; this is what I can provide.

So R1 have now in this protocol we will know that I have all that all are providing. And based on that, this hash ID range has to be a partition, this continuous range; this is the range. Depending on how much storage is there with whom, it will say okay, this much I will keep, this much the first P1 will be keeping, this is a second P2 will be keeping and so on. So it will do a split.

Now what it will do is it will now create a spillover table entry, so for this range, it will now specify this range. For this range, you have no local search, for this second range, you should move to post P1, so there will be P1, node ID can be there, or P1's IP address and port number can be there. Node ID is also acceptable; in that case, the destination address will not be hash ID, but it will become P1, the node ID will be there, destination address that replacement will be done in the query while we move here.

And P1 knows the query has come to me, and then it will essentially look into this whether I am the root node or not or whether the entry is there in my spillover table. Once the query goes to P1, so P1 will also have a table. Similarly, it will say that for one spillover table, which particular entries will be there; it will say for these entries that look into my storage in my range.

If it provides storage to others, we usually expect no spillover table to take care of all its entries. So, there is always a rule that you should always take care of all your entries first then only give the space to others; if you are running short, you tell the other root node for which you are taking care that you do not have space and you will find out some other people where it can re-rooted. So, that procedure has to be done.

So, and then of course, if there can be now entries from the small side that can come and says that I am maintaining entries for this, I am not root, and these entries are there with me, it is not with somebody else. It may be sometimes feasible that it is coming to you and running short of

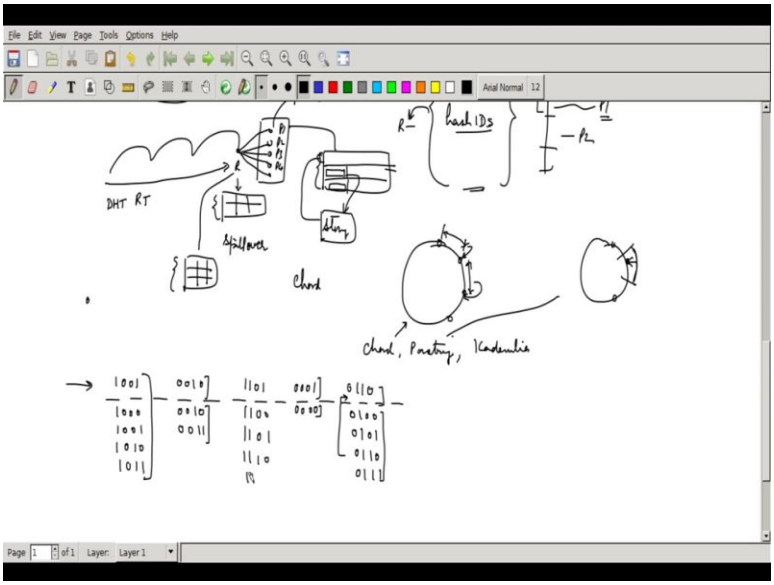
space, we have told him. Still, you have to keep the entries; you can make a mechanism whereby even you can further do another spill over and go to another root node.

You are not storing it, but then that is very inefficient. So, in Brihaspati 4, we have not implemented that scenario to go the utmost to what we call proxy root. This is also acting as a proxy of this particular root. That is why we call it a proxy root node. So, only one step should be able to go there. So, and anybody maintaining this table, either you will have it, or you will not do not have it.

If you do not have any entry, you will respond to the source that entry is not available. And I am sending it on behalf of this particular R. If R does not have the entry, it checks with the proxy root; if the range is telling that this particular key ID range or hash ID range has to with some other guy, he will look at the entry; if he does not have the entry, he will say I do not have the entry, or he will have the entry, he will pass on the entry.

Now the source will know that who has given me the entry, even figure out that P1 has given me the entry, not the R. Now that is essential because P1 has to be given credit for storing the thing, not the R. So, that is will be required for our reputation system.

(Refer Slide Time: 16:19)



Now briefly coming back to this continuity, this continuity is there in the chord, which is guaranteed, pastry this is happening, even in Kademlia. We can verify this happens in Kademlia; for example, you check, you been have, put some nodes there, try, for example, I have 100, I am

going to put 0010, I can put node say 1101. I can put 0001, so let these be the Kademlia nodes for which entries these will be responsible.

So, I can now find out. I saw distance for all hash IDs, which are feasible and see with whom they are closest. And then look at that range, look this range will always be contiguous, and this is, of course, is obvious. So, for example, look here, so this for Kademlia I am doing it, similarly, for tapestry, it is, of course, evident because it is going to be on the circle. So, this whole range, this is going to be the root in case of the chord.

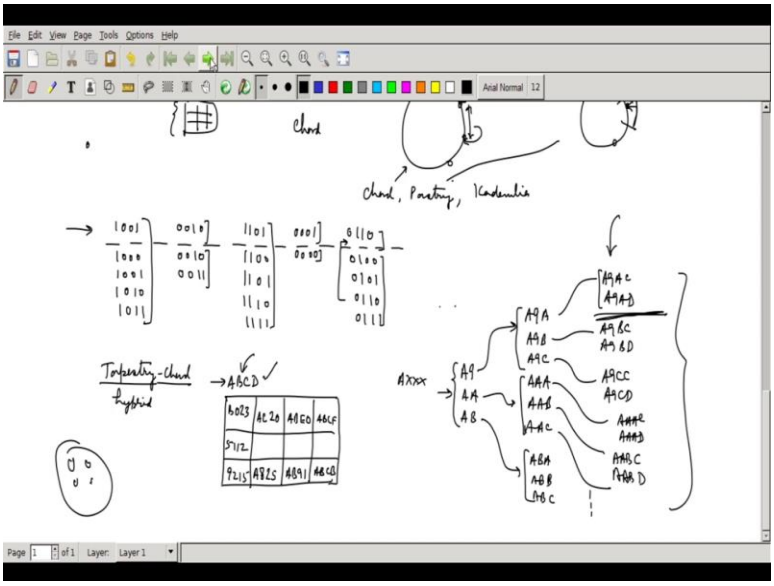
For tapestry, if you have a node here, half of this whole range will be the root node for that. So, this will also be a contiguous block. Kademlia, we can now verify it here, so if I start this all 0s, so all 0s when I do, I will look at which is the longest match. So all 0s match with this one. So 001, it is here only. So, I am just going to do this longest match business, and with that, I can find out who will be the root node.

And then, of course, 0010 so that will come here, 0011 so this will also come here, you can see, I will not be coming back here again. Now 0100, 0100 will be matching closely here, so then 01 and you can now understand why this is happening. Is it a busy longest match thing because of that, 101, 0110, 0111? Now next entry will be 1000, so this will be having the longest match here.

So, now I will be going back to this, this is longest, this is a contiguous block which is there, there is a contiguous block, there is a contiguous block and including this at this entry also. You can actually, this entry is going to be there. So, remember this entry is nothing but the same thing, 0 distance. So then 1001, so which is the same, so 1010 so this will be 1010 will also be coming here only because this is the only entry left.

Then 1011, the moment I say 1100, this is not the match, it will be this 1100, so this entry is done. And then, 1101, 1110, 1111, all are continuous blocks. This is not true if you are using tapestry; for Brihaspati, it is not proved actually.

(Refer Slide Time: 19:40)



So, for Brihaspati, we are using a tapestry algorithm version technically. Still, our routing tables are smaller because we use a chord kind of arrangement on every column. This we have discussed earlier in one of the lectures. If we try doing this on, say tapestry kind of thing, so let me take a node ABCD; I am taking four nibbles, four hexadecimal numbers, only four-digit node ID. So, let us look at the first thing. So, what entries I am going to have the five entries B0.

So, this entry, the way, is going to be a Brihaspati tapestry called hybrid, I should call it, and of course, the next entry will be anything which is going to be midway. So, this can say 5112. So, I am keeping something arbitrary than in 9215. Then obviously, after that, my between 9 and A whatever is going to be there, for that ABCDs or any node which is starting with A will be responsible.

So, anything which starts with, after 9 we get 10. So, there is only A, the entries that will be present. Only for nodes starting with A this will be responsible; I have to go to the second column in this case now. The second column I am just taking slightly differently; I am taking AC 2 0; I am not bothered about the middle value does not matter. But here I am, taking a different value; I am taking 8. So there is A 9 AA and AB for which now this range will be AB range, ABXX will be responsible, the domain will be responsible.

So, I have just taken this particular thing intentionally. So, for A9, AA and AB, the nodes, which are starting with this, for them, actually ABCD can be the root node; for A9 and A8, somebody from ABXX has to be the root node. So, ABCD is one potential candidate, so I will find out

which entries will be there. So, next, let us look at the third column. In this case, I am intentionally again putting the values in the same fashion ABE0, and then we have AB91. And I put ABCF and here ABEB.

So if I look at, now once the for A9 AAAB, I will come back here if I am at ABCD root node, and I am searching for something with A9 I will come here, and I will now start looking at the third digit, third digit is going to be there between 9 and C everything has to be there. So, I can now say that A9 and, of course, A9A A9B and A9C, for which I will be essentially handing it over to ABC only this particular node. Similarly, for AAA and AAB and AAC, and similarly for this B, you will get ABA ABB and ABC.

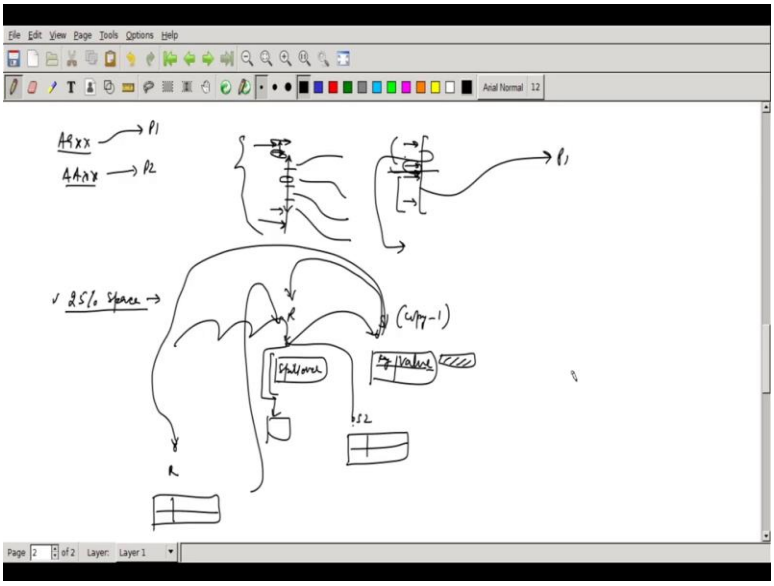
So, these three and I will put XX, so my ranges have already got a split, there will be some range which is not going to be having a root at ABCD, it will be having a some other, some other node will be the root for that. Now, let us look at the fourth one, the fourth entry. The fourth entry will be now ABCF and ABCB. So, BC and D for which this is going to be responsible. So, I can now write A9AC, A9AD for which ABCD will be responsible.

Similarly, I can look at this and as A9BC A9BD. Again this node will be responsible. There is a gap here now, which is left. So, EFG, EF A9AE, A9AF for which this node will not be responsible. Some other node will be there.

So, the entries are not going to be contiguous for the tapestry chord hybrid. So, we cannot use the simple strategy of saying, okay, whatever you find out, for whom you are responsible and bifurcate that hash ID space for which you are responsible because this hash ID space for which ABCD responsible is not contiguous. This is fragmented, so some other, some other places this is going to be responsible for that.

Similarly, for A9C, you can do A9CC, A9CD; for each one of them just we will put C and D and AABC. So, you can appreciate how this is happening, and you can keep on doing it. This hash ID ranges for which hash IDs for which ABCD will be responsible, and it is not contiguous. So, we are improvising on our algorithm here. And what we do is we now use a mechanism.

(Refer Slide Time: 25:40)



So, that is one possibility I can always say A9XX based hash ID will go to node P1. And, of course, AAXX will go to P2. That is one way of doing it, so prefix this thing. So, I know that A9XX, there will be some hash IDs coming to me for which this guy gets responsible for this, this gets responsible. Still, I will not be able to do a clear division. I have to find out how many are there, and then each block has to be assigned to different; I have to identify the blocks that could be one possible way.

But what we have done, we have done in at least slightly different ways. So, what we do is we identify what is the lowest key actually which has come so far being for publication, and what is the highest key which has come for me for the publication or query. This is what we call a range for me, and this is what is being bifurcated or trifurcated and then is being assigned to different nodes.

I am assuming that there will be uniform gaps, so when I am doing bifurcations, I am not aligning something for which I am not the root to somebody else, and of course, no storage will be used there. That should not be happening, this actually can happen, but we live with it as of now. If somebody's hash ID is coming out of this range, I will change this my pointer of the lowest to the value given by this.

Similarly, I can update this value depending on whatever is coming, and the range will keep on adjusting for which I am figured, I am figuring out I am the root node. So, I will know the lowest value for which I am the root node and high, which is the highest value for which I am the root

node. And I am essentially bifurcating that space to the nodes. We can even do a much more sophistication; we know that these are the entries, how many entries are available.

And when I figure out they are more; I will do a midway bifurcation. I know which entry is here, and this range will be directed to P1; this range will be there with me. And when more entries come, which is somewhere in between, I have to allocate and update one of these, whether I am keeping it here or here, depending on the size. So, this is the algorithm that we have been using. So, these ranges are being dynamically adjusted in our case.

So, the rule which is followed is that every node before it will create a spillover table. So, this spillover table creation mechanism, so where the ranges are being specified. So, usually, every node should keep 25 percent space available. So, when you reach about 25 percent, you will start creating a spillover table at that time. You will start a query to find out the people who can give you the storage space in the form of a spillover table.

Then, of course, queries come to you, and from there now, you will have a spillover table. And based on the ranges which are defined here, you will either search locally, or you will then hand it over to your proxy root. So, you maintain the key-value pair and respond to the requester directly, the guy who has made the query. What this S node, which I am a storage node, this R node, needs to keep on republishing this entry periodically.

So, there has to be a republished timer, so once it will do republish, it will go to this current node always, and this, in turn, it will always look at the spillover table and pass the things to it. If the entry already exists, it will now reset the timer; if it is a new entry that means the R has now decided on some other spillover, depending on the consumption. If the value goes above 25 percent even after this, it can further determine more nodes S1 and S2.

And then actually now do a bifurcation further, a spillover table changes. So, next time when you republish, it goes to a new node. It will not come back and then ultimately, your timer; you will keep this entry for some time; if it can refresh, it is fine; it does not, you will just purge it. And by the time this entry must have been placed at some other node. So, this is the way it is going to be republished again and again.

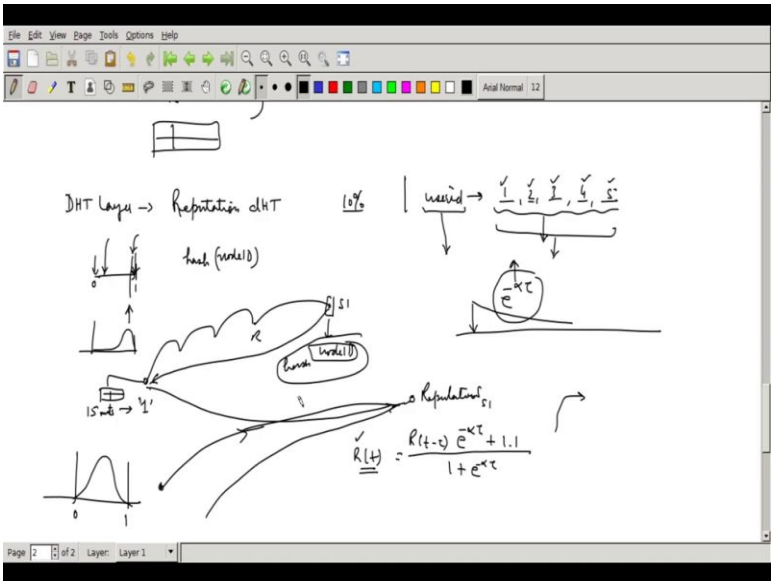
It also needs to know whether it is if this is copy 1 or copy 2. If it is copy 1, it will be doing a republish for copy 2 in that case. So, there will be another node sitting in here. So, it will now do

a republish for copy 2. This will ultimately end up and terminating here, and this root node may not have proxy root being defined so that it will store locally.

This also has a republish timer, so it will now do a copy three publication; somebody will be doing a copy 1 publication, which will come here and which in turn will get replenished here. So, it is a cyclic thing. If we are doing parallel fork, for resilience purpose, so the shortage of space has now taken care of in this fashion by borrowing the space from S1 and S2 but what S1 and S2 will get out of it is the question which remains.

Now, to solve this problem of that people should not free-write. So, we now do we also create another mechanism; we call it a reputation management system.

(Refer Slide Time: 31:14)



Now, to ensure that nobody free writes, people who contribute could get their fair share in the whole storage DHT. So, we need to build up a reputation management system. So, some nodes will be now can build up another layer, another DHT layer; we call this DHT layer as reputation DHT.

So, reputation is a number that goes between 0 and 1. So, depending on if somebody who will have a higher value means that he is cooperating, giving, and providing access to others, he gives you the storage space for others to use. Somebody who is given less storage space will have a

lesser reputation system, so the higher the storage you contribute, the higher your reputation, lower you contribute, lower you will have.

And accordingly, you will end up also getting your share in the total storage which is being provided by the DHT network. There will be only some nodes; roughly 10 percent of the nodes will be good enough because each node then has to maintain an average reputation of 10 other nodes. And this is, of course, through hashing. So, whatever is your node ID, we will just compute the hashing of this. We have to do some more tricks because a single user might end up having multiple devices. Each one of them will have a different node ID and put it in the space together.

So when I am trying, whenever some file is going to come for storage, it is going to come from a user ID, not from a node. So, we have to search for this user ID what all node IDs are assigned to it; we have to compute their reputation thing. Based on that, their reputation, which we have, basically how much storage they are pumping in will decide nearly the reputation.

So, higher storage means a higher reputation; lower means lower. Then we can compute an average of this. And that average can be used to compute how much whether the file which is being published by user ID has to be or the fragment has to be stored or not stored at a node. So logically, the way things will happen is whenever a user subscribes, find out a fragment it will access, it will go to the root node, it will go then go to the spillover table will go to the storage node. Storage node will supply the file back to him.

Once the file comes to him, it will merely then inform for this S the notice node ID; there will be somebody who will be a reputation server, that we will just compute the hash of it. And this will be rooted in the DHT layer. There will be another routing table corresponding to this reputation DHT. So, this particular message will be going towards somebody who will be a reputation server for S1.

So, the moment this guy gets this will, we will also limit how many it can send and how many it will, every time it will send 1. If you just send a signal, I asked for a fragment. I got the fragment, and it can only send every 15 minutes A1; that is what we have been keeping. This is assessing a lot of files; only one entry will be going, and then, of course, it will also be maintaining a cache memory.

Next time it will not be accessing it will be only fetching it from the cache, local cache every node is supposed to maintain as I have discussed in the previous lecture. And this, one whenever it comes here, so there will be an older reputation value that might have been taken at say some value t minus ъ, so t is the current time. This value we will indicate by because whatever is there in time the reputation should go down if nobody is accessing them from S1 is not providing file, earlier it has a very high reputation value, it should decay, decay.

So, we will define $e^{-\alpha ъ}$ as a decaying function. So, α is roughly kept so that within 24 hours, this value comes down to 0.1. So, we will have some α ъ; the value will this is what will be the coefficient by which, which will become the weight of the order value plus the new value which has come as 1. I will also keep the weight one, and then I will do the weighted averaging.

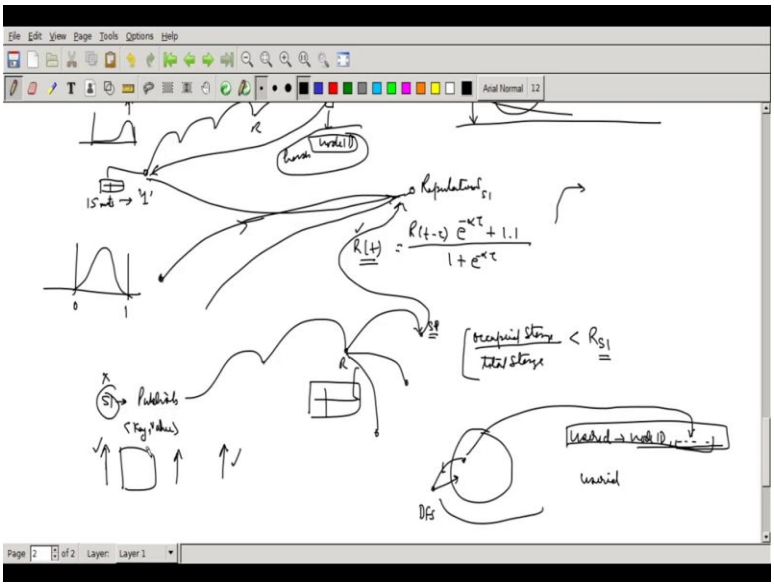
So, this will be my new value at e reputation. So, if many nodes are actually in for trying to access the file fragments and S1 is providing, so many 1s will be coming here to the same thing, and this R(t) will keep on replacing; this value will remain very high if it is providing service. If somebody is not contributing, it will be low. So, α essentially has to be chosen so that the value will be somewhere between 0 and 1 midway.

$$R(t) = \frac{R(t - ъ)e^{-\alpha ъ}}{1 + e^{-\alpha ъ}}$$

All values should not be skewed; the distribution of the values should not be something like this at any point in time. Ideally, the value should have a distribution, which is going to be something like this. So, this α has to be chosen in that manner, so this needs some study. So, through experimentation, we will figure out how much α will be required. So, it all depends on how much, at what rate people will be accessing the files. Typically caching will be there when you do not find a new cache, then you will get it.

Sometimes you may not be running the application you are not accessing; it is only part of the nodes from the whole network that will be accessing, accessing the file fragments from S1. But the question is, the guy who is contributing higher storage will be accessed more by more people. So, he will have a higher R(t) value at any point in time than anybody else who provides less storage. Once this is done, so storage will be for the S1 node will be updated.

(Refer Slide Time: 37:34)



Now S1 want to publish something, what is going to happen? So, how the advantage comes? So, S1 will now give a publish message; now this goes to some guy who is the root node. Now root node figures out, okay then I need to publish it; it does not have that much storage space. It can have somebody providing the storage thing to call SP, and the request goes here.

So, this SP will say I am supposed to publish for S1; there is a key-value and usually remember all fragments are of equal size; that is what we have been maintaining in our design. So, SP will now search, again make a query to the reputation server of S1. So, the query will go here, and S1 will provide it back the reputation value. Now SP will simply do, it will find out the occupancy as of now, occupied storage divided by the total available storage.

So, if this value is going to be smaller than the reputation of S1, then the value will be stored; otherwise, it will be declined, it will be simply sent back to R, and R has to find out somebody with a having more space which is going to have less fractional utilization. So, utilizing space, that fraction should be lower, so it has to go to only those nodes. So, R actually can keep various nodes and find out the fractional values they have.

But in that case, the problem is this, R has to have that spill over table has to be modified. Usually, it is not required, simply there should be a refusal that should go back, so S1 cannot store more than a specific value. So, it has to essentially now contribute more storage to make its RS very, very high. The higher the contribution it has, the chances are that anybody who is

having this value will store it. Intuitively, we can see that most likely, but this still needs verification. We are investigating this particular part that almost everybody's fractional utilization will generally be the same in such a scenario.

So, anybody who is going to have a lower reputation will simply be discarded. Only in the beginning might they be able to store, but after some time, when everybody storage will start building up, they will not store their files. In that case, they will get denied. So, some fragments will get stored; some will not get. But usually, this is going to; we use copy 1 copy 2, copy 3, so they will, it can find out only individual copies it can store not the others.

And this problem will be there, so one possibility is might it would use the storage, copies can try randomly, and when some random trials it will find out that storage is available, sometimes it will not be. But most likely, since it is a feedback system, so higher you contribute, the chances are that you will have a higher reputation, the chances are that other people will be able to store you. So, you get an award for actually sharing higher storage space, and you will get a higher chance of getting your files stored.

If you are sharing less, you have fewer chances. This automatically gives an incentive that you can contribute more towards the storage so that the other people will do it. Now one of the problems here is the people who are not participating in the storage DHT. They want to use this, have a DFS and use the storage; they cannot actually. Because they are not contributing anything, their reputation will be 0, as far as in the DHT layer, for the reputation is concerned.

So in such a scenario, probably there has to be, we have to build up some mechanisms through which it can buy the space. So, in that case, a node ID, if they get into the node, contract node ID provides okay I am going to contribute space on your behalf, so then actually for this user ID, I can now store, user ID to node ID publication when it has to be done. So, even this node ID has to be added there. So, this node ID will now allow this node ID to be added. And what happens is this user ID cannot arbitrarily put anybody's node ID here.

It has to be signed by this entry has to be signed by the node ID's private key and the user ID's private key, then only it will be feasible. So, we have to make a change here that every user ID, node ID mapping has to be signed both by the node ID's private key and the user ID's private key, user's private key both. So, there is an agreement that this user, node ID mapping is perfect

and now based on this because this guy is providing contributing storage. This guy gets a higher reputation, and its actual files can be stored in the storage DHT.

So, that is how you can now buy storage from the storage service providers. So that is the mechanism in peer to peer system which can be used. So, this is what we have been planning to use for the storage system. This is how the file system will get implemented and where the storage will also be balanced; we will also have some kind of rooting reputation system.

Of course, this will need a careful analysis, but as far as using this logic, the higher you contribute, the more chances are that your files will get acceptance in the storage. This particular mechanism virtually ensures that there is a tit for tat. If you are storing, contributing less, you will have a lower reputation nearly, and your files will also not be stored. So, the free-riding can be, is somehow taken care of in this system.

With that, we end today's this video, and we also end these whole discussions on storage systems. Now I will overlay multicasting, essentially do broadcasting of live sessions or live lectures in a peer to peer system.