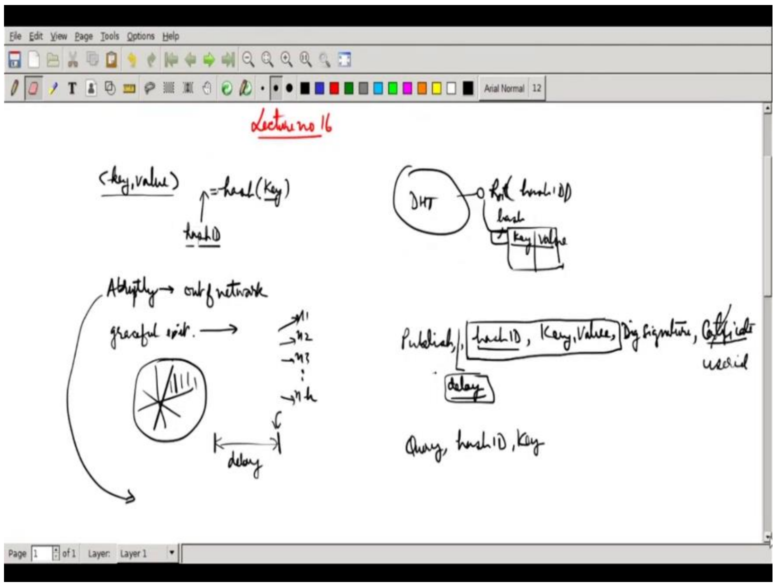
# Peer to Peer Networks
**Professor. Y. N. Singh**
**Department of Electrical Engineering**
**Indian Institute of Technology, Kanpur**
**Lecture No. 16**
**Abrupt and Graceful Exit of Root Node: Maintaining <Key, Value> Pairs Alive**

(Refer Slide Time: 00:14)



This is lecture number 16. In this lecture, we will be talking about how the key-value pairs are being managed. Key-value pairs are going to be managed if the nodes exit abruptly or go gracefully. In this scenario, how do we handle the key-value pairs they are not lost. Ideally speaking, the key-value pairs should always be there in the network irrespective of whether the nodes are joining in or are leaving abruptly or doing it very frequently or doing it after a long time.

So, it should be the key-value pair; their existence should be guaranteed in the network. We are essentially how this will be maintained what we will look for in this particular lecture. So, there are two possibilities. Now, this key-value pair how are they are being done. So, the way it is being stored, any key we have is computing the hash value.

This is the hash function. As a consequence, I will get a hash ID. And what we do is we try to find out with whatever mechanism the way we define the root node, root node for this hash ID in DHT network. So, we find out somebody who is going to be the root of this hash ID. And this node is going to be responsible for holding on to the key and corresponding value pair. So, it will be maintaining a database and, in this database, it will hold key value.

So, whenever want somebody wants to search for this key, he will just again compute the hash of this key, get the hash ID and send a query message to this, the root of the hash ID requesting for the key-value pair. Now there are two possible ways in which the root node can exit the system. One is an abruptly they essentially go off out of the network; they either have turned off or are disconnected.

So, they abruptly out of network. That is the way I should now term it. And the second possibility is they gracefully exit. They are planning to shut down, so they do a graceful exit. So, in the case of a graceful exit, I think things are simple. So, what they can do is any node that will cut down, he will simply collect all the key-value pairs he is storing. He will then will identify a few neighbours. So, if the neighbours will be, say, n1, n2, n3, and so on nk, he will now essentially divide all key-value pairs into k parts to create packages.

Each one is going to be packaged separately. He will send a publish message, but now in the publish message, usually in previous lectures the way it has been done, publish was the message that has to be sent, it has to go to the root of hash ID, so this would have been the destination thing. Then there is a key, and then we had sent a value. So, of course, the hash signed hash we also call it digital signatures. So, let me write as a digital signature set what we have been doing.

So, nobody can temper the value. So, digital signatures will consist of this particular part. So, this is what is going to be stored even in the database. So, here not only the key, but the hash is also being stored. So that keeps life simple; you do not have to compute it again. It is possible you can live without it and compute it dynamically based on the requirement. It is a choice, so it may be there may not be there; it does not matter.

So, even if it is not there actually stored, you can always compute the key's hash, put it here and then put the value and then compute the digital signature on top of it. But for verifying the digital signature, you require a certificate of the person who has created it. So, his public key will be available here. With his private key, only these digital signatures would have been created, and with the public key, I can verify this is correctly done. So, which means nothing else has been tempered in these 3 triplets.

So, the only thing which now we need to do is we now to insert a delay. So, once the delay is inserted, this node, which is doing a graceful exit, has communicated the various republish packed messages to various nodes. And these nodes now have to wait for this delay. This time, this node will be doing graceful exit; he will simply disconnect from the network.

There is a consequence, and he will also tell all his neighbours or all routing table entries that he is gracefully exiting, so please purge my entry. As, as they are purging their entry, they will also further communicate with all the nodes. There will be like in any node wherever routing entry changes are supposed to inform all the entries it contains. So, there is a triggered update that happens, and this node's entry will be moved out from everybody.

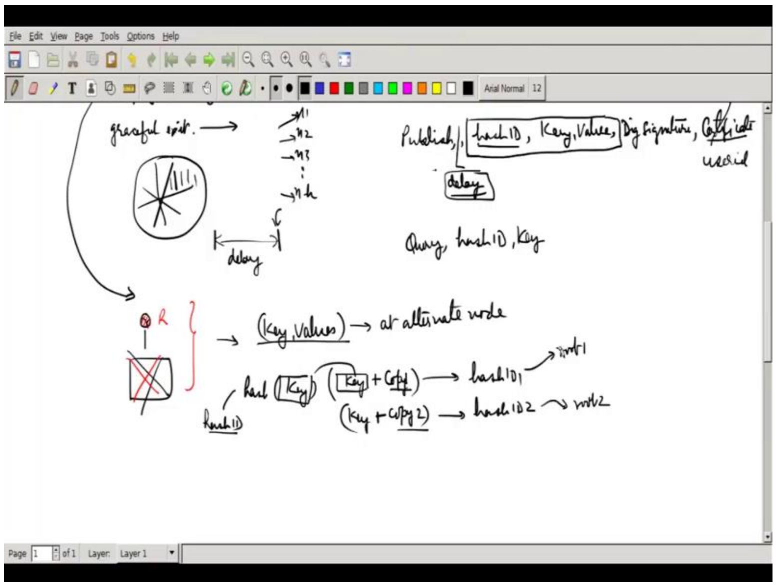
So, nobody will be maintaining the node ID of the gracefully exiting node. This will happen within a specific time, and we ensure this delay ensures that the routing table will not have the entry for the exiting node. After that, they will all do republishing all the key-value pairs which have been dispatched to them, and the ones they will republish will go to the new root nodes.

And they have found out they are new root nodes, anybody wants to search the same key again he will take the key and then compute the hash, find out the hash ID, send the message of the query, basically asking making a query in that a query message. The query message will contain the hash ID and key; that is the only two things it will contain, and this will go and terminate at the root node, which will then look into its database, which is now available because of the redistribution. The whole message hash ID key-value and digital signature and certificate will be sent back to the person who has made the query.

Of course, this certificate will be repeated a large number of times as a large number of key-value pairs will be there. We have to figure out a way even to sort this out. So, we will find out a way later where we will be just replacing it with a user ID, and the certificate can be found again from the DHT network. So, we will essentially add another kind of key-value pair into the system. We will come to that later.

Once this is done, so we can find out, we can now restore all the key-value pairs in case of a graceful exit. But what happens if there is an abrupt exit which is going to happen. So, an abrupt exit has to be handled somehow. So, let us see how this will be done.

(Refer Slide Time: 07:49)


graceful exit
delay
Publish, hash ID, Key, Value, Dig Signature, Certificate
delay
Query, hash ID, Key
(Key Values) → at alternate node
hash (Key)
hash ID
(Key + Copy) → hash ID 1
root 1
(Key + Copy 2) → hash ID 2 → root 2

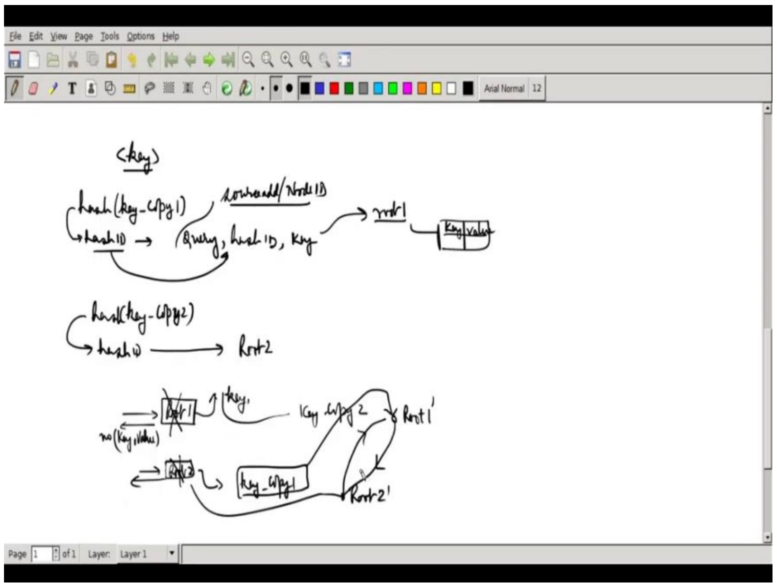
So, if there is an abrupt exit, the key value is lost because the node is no more there. So, all key values it was holding are inaccessible; they are not available to anyone. So, once they are lost, how can you even republish them? You cannot. We have to do something. It means that somebody else other than this root node has the same key-value pairs with them; the same key-value pairs are there at some alternate node.

They will not be a root node when this root node abruptly ends off. They need to monitor it. If it is not there, they will just republish these entries, and these entries will again end up with the correct root node and then be used in the network. That could be one possibility. Other possibilities we can think of the possibility of having more than one root node. Now, one actually will think about how that is even feasible.

I have only one key, and I am computing the hash of it. I have only one hash ID, and with this hash ID, I will always get the same root node, which there will possibility of having multiple root nodes. Now that is feasible if, in the key, I can append something. So, I can take a key, whatever is a string, and I can append something, say, for example, I can attend copy 1 and attend copy 2.

So, these are separate strings. Once with the same key and I compute the hash of this, I will get two different hashes. I will get hash ID 1; I will get hash ID 2. So, these will now go to a different root node, I can call it root 1, and this will go to a different root node, I can call it a root 2, and I can publish the entry for the same key value at both of these places.

(Refer Slide Time: 09:57)



Now, a node needs to search for a key either for publishing or even for research. So, in this case, I am assuming it to be for a search. So, what he will do is he has to choose one of the two options essentially. He can either put a copy 1 and then take this string and compute its hash value. You will get a hash ID, and with this hash ID essentially, he will now float a query. He will send a query message to this particular hash ID.

This hash ID will be sent, this will be the destination, and then you will say this is a key I am searching for. It will go to the root node 1, this will terminate there, and the key-value pair is stored there, and it will find, and key-value pair will be sent back. So, for sending back to whichever source, either actually can via DHT way or directly go to the endpoint point address given in the packet. So, there has to be a source address also.

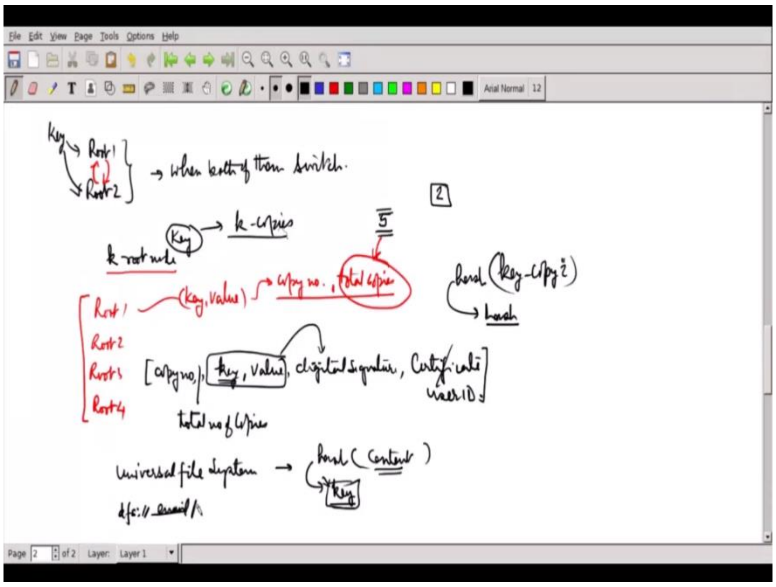
So, there will be a source address that can directly address the port number, or this can be a node ID either of the two. If it is node ID, it will be sent as a response. The query response message will be going with the key-value certificate and everything. So, alternatively, it is an option to choose, the node can decide I am not going to do this, I will choose a key copy 1, key copy 2 and then compute the hash of it. When searching for the same key, different nodes can randomly pick up copy 1 and copy 2.

So, they will essentially be going to two different nodes, and the load will get distributed among them. This will compute the other hash ID. Again, when the query is propagated with this one, this query will now terminate at root 2, which will respond that the same thing is being kept both the places to the same key-value pair. Now, I can use this to my advantage. If root 1 dies off, what is going to happen?

So, in that case, if a node makes a query with copy 1 it goes to root 1, the response comes there is no key-value pair with this key which is existing. It can then search for a copy by adding copy 2, and it will go to root 2, and from there, it will find out the key-value pair. So, we can now build up a mechanism that root 1 node this is supposed to do a republish of whatever is a key-value with the key if it is storing copy 1 it is going to put the hash ID to which it is going to republish will be computed with key copy 2. This guy will be doing a republish with a hash ID of key copy 1.

So, in this case, if this guy dies off, this guy will be making a publish. This will end up going to a different node, which will be a new root node. This will be a root 1 prime, and this guy will again do republishing, which will come here. So, if it dies off, this will go to the new root node. So, root nodes will always remain; they are essentially not trying to do a root. These roots nodes monitor each other, and they are trying to provide back up to each other. This way, entries are never lost; they are maintained in the system.

(Refer Slide Time: 13:53)


Key
Root 1
Root 2
→ when both of them switch.
2
Key → k-copies
k root nodes
5
Root 1 (key, value) → copy no., total copies
hash (key-copy i)
hash
Root 2
Root 3
Root 4
[copy no., (key, value), digital signature, Certificate]
userID
total no of copies
Universal file system → hash (Content)
Key
dfs://email/

Now the question arises when it will be possible that both pairs are holding key-value pairs, so one is the root 1 and one is root 2 or a specific key. These are the two root nodes. When key-value pair may be lost, so far, what we have done is we were assuming that both the node, nodes will not be dying together. But there is a finite possibility that even if they are far apart, they may end up turning off together.

So, there is a risk which is involved. So, when both die-off simultaneously, then key-value pairs will be lost. In this case, the one crucial thing is that for every key, how many keys will there be the same root nodes. So, when they die, probably one or two keys will only be lost, not more than that. Because you take, you change the key. The root node pairs will keep on changing. So, all keys which are there with him are pairing with different root nodes.

Only probably one or two will be there, which is pairing with this guy. So, if both die, only those particular key-value pairs will be lost. Another node will come back again, and they are restored on a disk, the key-value pairs will resurface again. So, that is the finite bleak chance of actually losing. So, can we remove that chance also? So, maybe one possible idea is that instead of using 1, 2, I can have k. So, I can have k root. So, I will now maintain k copies for this, k copies will be maintained. That is a replication factor. But how this will be done.

And then, of course, who is going to take the monitor each other? In this case, this guy was monitoring this person, this guy was monitoring this, and they were doing republishing. So, we created a kind of a circular backup of each other, in this case, how this will be happening. We also need to change something in our publication when it is done. For every node, for example, I have root 1, I have root 2, I have root 3, so everybody should know which particular copy number it is holding.

So, where I am only storing these nodes are only storing key-value pairs. We also need to store now which particular copy is being stored here. So, there is a copy number that needs to be maintained. I also need to maintain how many numbers of copies are being maintained actually. The total number of copies also has to be there. So, these two data also has to come in, now in each table.

So, the way to take this actually can be a parameter for the system. However, it is better to keep it programmable, so when the source is going to publish, he will say that how many total copies have to be maintained depending on the worth of the key-value pair. If he wants that, it should never be lost, maybe he can maintain 3 or 4, but even with 3, 4 this will be extremely reliable and extremely reliable. Chances it will be lost is going to be very, very small.

And because there is a contract replication, the next copy is automatically created if anyone dies off. In this case, the way the values will be stored will need to have now in the packet that we will send the key, and the value will be there as usual. We will also be having digital signatures, and digital signatures will still be only on this, and of course, now the hash ID does not make sense here, so hash ID is not included inside this the way it was told in the beginning.

So, hash ID now will depend on which particular copy you are trying to access, so this digital signature must be on this. And then, of course, there has to be a certificate. We will even modify this certificate thing, and we will convert it to user ID, as I mentioned earlier. We will have to add a copy number, which particular copy, and another total field number of copies.

So, this is what will be the total data structure that has to be communicated during publication. When you are doing the query, you have to nearly send it to the hash id, depending on which copy number you are trying to access. You do not have to worry about the copy number, the total number of copies there; just take the key, compute whatever copy you want, so you want key copy number i, we will put this i, compute the hash. You got your hash ID, root your query to this.

So, you can change any value of i. 1, 0, 1, 2, 3; you can keep on trying. If this number is an available parameter, if this number is a fixed parameter, then, of course, you know that you cannot go arbitrarily very large. However, you can start 0, 1, 2, 3, 4, 5 till you find it, and when you get the response back, you will know the total number of copies available with you. So, yes, that is one of the issues that which how much range which I should choose.

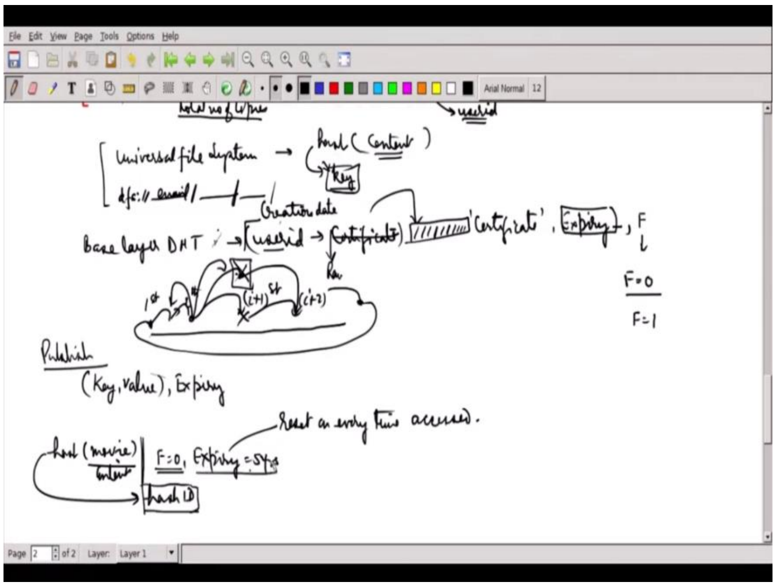
So, we usually can actually in our Brihaspati 4 design we have kept it at a fixed parameter value, we have kept it at 5. So, 5 is a replication factor. For when I am talking about a storage system, there we have kept it as 2. So, there will be a total of 2 copies which will be maintained. And one copy which the user itself because he will be maintaining his file systems copy. So, in case he dies off, he can always get the copies from here.

But we are using there; we are still discussing the mechanism which has to be used in that system. So, once this is there, now the question is what kind of key can exist. I have also to tell you that there will be and as I told you earlier, I think in the universal file system, for example, a movie. So, I cannot put a name or something; I probably will take the entire movie content or movie fragment, computing the hash of the content itself.

And this is what is going to be the key to the content. So, I will then attain copy 1, copy 2 this particular thing, this alphanumeric string and then, based on and doing the hash of that will give me the location was this content would be there. It will be indexed based on this key, which is the hash of the content so, that we will also be the indexing parameter. That could be one possibility; a key can also be a string; an alphanumeric is a string that is being decided by you.

For example, for a DFS system, we use it something like a DFS we also define an email ID, and in the virtual file system space where the inode is, I will talk about this file system, later on, this we have to come to.

(Refer Slide Time: 21:10)



Universal file system → hash(content)
key
dfs://email/—/—/
Creation date
Base layer DHT → [userid → certificate] ////// 'certificate', Expiry}, F
F=0
F=1
(i+1)st
(i+2)
1st
Publish
(Key, value), Expiry
hash (movie) content
F=0, Expiry=5yrs
hash ID
Reset an entry time accessed.

And the third possibility is we are now introducing ourselves and think this will make much sense. I do not want to store this certificate again and again; it is better to use only the user ID who has published it but to verify the signature, I need the certificate of this guy. So, in your DHT base layer, I can now put user ID to certificate mapping, which is also like a key-value pair. And this certificate will be signed by itself, and this user ID key pair will be signed by the private key of the user ID.

You can look at the public key here and then verify that this is appropriately signed. So, from here, you will be able to get this is another kind of a structure which will be present. I do not require a certificate again, actually, in this case. The value is also a certificate, and another certificate does not make sense. You need not do it because the certificate will contain the user ID, so you just search for user ID; you can keep this thing blank; there is no value.

Then, there is a hash signed hash that this certificate had created this particular entry. So, you can put some dummy thing here, some random thing. So, putting a certificate does make sense to me. The certificate is also a digitally signed stuff, user ID I can directly get the certificate and verify the digital signatures. So, that is what will be done.

Now, how the backup will happen, so I have now many copies of how these copies will do each other's back up. One possibility is that I will, the $i^{th}$ copy will always be publishing i plus, i plus first copy. So, it is forming a ring. This guy now does i plus second copy. So, since the i we have, we also have a total count also have been put here, the total count of copies. The last guy will be now publishing back to the first copy, copy 1, which will again do like this, so they are all, even if some node is lost.

It will now find out some other node which will then be connected to this, so this is the new root node i plus the first root node that will be existing. A larger number of this actually can be used with too high reliability depending on your application requirement. In Brihaspati, we are going with only 2 now, but you can also have a higher if you require you.

Now we can further innovate; there is a possibility that one of the root nodes is a rode mode, and you publish, and this guy keeps a garbage entry is not publishing any further. So, you can publish in both directions. You can also publish to i minus first that every node can do i minus first and i plus 2 in the cyclic fashion. So even if one node is ridden so from another side, you will be able to publish the whole thing, even if one node is ridden by chance, so when people try to find it, they will not get the entry, when the query is going to be sent terminating here.

So, usually, a good idea for making a query here is to make a query to at least 2 or 3 guys, then use the majority logic to get the key values pair, depending on the value of that particular data you are looking for. Now, so far, key-value pairs did not have expiry. But every key-value pair also need to have an expiry. So, this will be done when the publish will be done. This absolute expiry will be maintained, so I am putting my certificate and everything this will expire in 6 months.

So, in this case, I need to have an expiry entry, which has to make here. When I am going to make the entry at that time, I will put the expiry period, say it is 6 months, but then I also have to give another value when the entry was created, creating date. So, creation date and expiry period based on that, I can figure out when this entry has to be purged from the database.

Now a source has to put in this entry before this expiry happens, but this will be removed. Republishing is only done to always go to the correct node, but the source decides that data's life. Now there will be some sources, for example, like a movie, I am putting a movie, and the way it is done in our design is by computing the hash of the content and that hash ID. So, you will do a hash of the complete movie content, and this hash ID is the key to this.

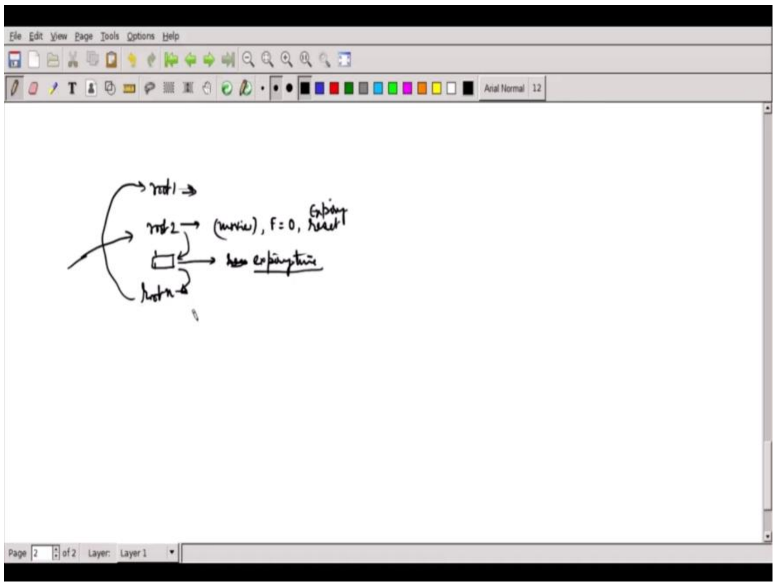
So, so far, somebody has this hash ID; this will be accessible; the moment this hash ID is lost, nobody has this hash ID, this content cannot be retrieved. So, I need to clean and purge all these extra what we call orphan data structures. So, one way to do this is I will also now set up a flag here.

So, I will create the entry, I will set up a flag, and this flag is F. F means if F is 0, this is not final, and when F is 1 this is final. So, when the user ID and certificate kind of entries will be there, I will make F is equal to 1 and set up the expiry. So, this expiry won't change. So, from the creation date, after that much period, this entry will be removed.

But if it is a movie, I will set F is equal to 0, it is not final, and I will set up a say expiry of 5 years. So, we have built a mechanism whereby when somebody accesses this particular content, this expiry is reset; this expiry is reset every time the content is accessed. So, this nearly gets a; if nobody will use it, nobody is accessed for the last 5 years, then we can purge this entry. For this kind of perpetual, this is a perpetual content; this is not final actually.

If something is in use, it will remain forever in the network, somebody will always be holding it, and so far, this hash ID is there, with somebody he will be able to access it. And if the hash ID itself is lost, nobody is holding it; this will any way will expire and be purged from the network. So, it will not be essentially creating a burden on the system.

(Refer Slide Time: 28:12)

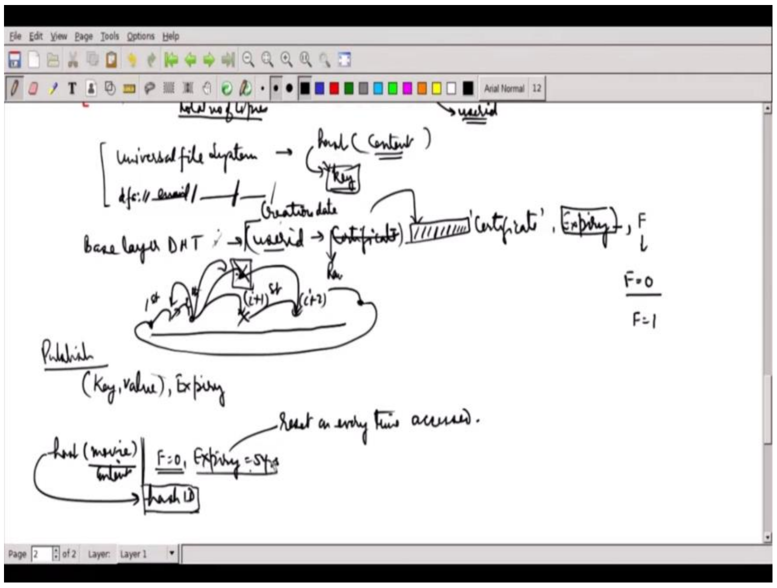

File Edit View Page Tools Options Help
Arial Normal 12
root1
root2
(movie), F = 0, Expiry reset
expiry time
rootn
Page 2 of 2 Layer: Layer 1

Now in case of expiry reset happening only I one of the root nodes, they are root 1, root 2, root 2, root 3, etc. So root n, for example, is there, the query came here, holding a universal file system. For example, in the movie, the case I am taking and the movie's f field is, f is equal to 0. In that case, a reset will be happening here for the expiry, but a reset will not be happening for the entries here.

So, the rule is whenever a publication is going to be happening, for example, it will publish to the root 2. When root 2 will be receiving the entry, it will figure out that the expiry value for the entry which is coming to me which is as a replacement because of a republish is actually better, than that case the earlier entry will essentially be is, is also reset timer will be, expiry timer will be reset in that case, even for this.

This will, have the expiry timer reset will happen, and during republish, wherever it goes there, the expiry timer reset will be done. It will be made equal to this guy will go this to the last person and ultimately to this place. So, within a time, even if one access is happening within no time, all the copies will have the same expiry reset timer. So, this is important so that the file, the consistency of the expiry timer has to clear across all copies of the key-value pairs.

(Refer Slide Time: 29:55)



Universal file system → hash(Content) → Key
dfs://email/ / /
Creation date
Base layer DHT → (userid → Certificate) 'Certificate', Expiry, F
F=0
F=1
1st
(i+1)st
(i+2)
Publish
(Key, value), Expiry
Reset on every time accessed.
hash(movie) content → hash ID
F=0, Expiry=5yrs

So, this how we can create resilience. No other methods can be used; we will be looking into those methods in the next lecture.