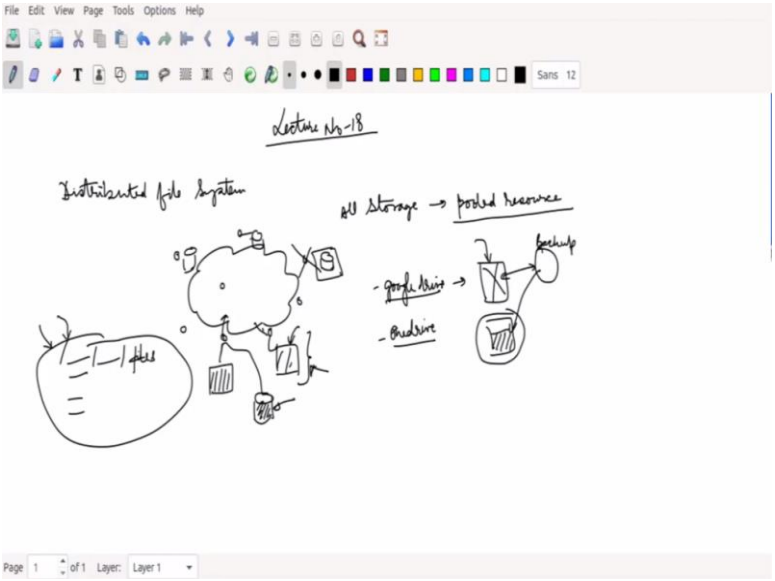
Peer to Peer Networks
Professor Y. N. Singh
Department of Electrical Engineering
Indian Institute of Technology, Kanpur
Lecture 18
A P2P Distributed File System

(Refer Slide Time: 00:13)



This is lecture number 18 for Peer to Peer NOC. In our earlier things, we discussed the resilience of key-value pair, how that could be done, and the various methods you can run through parallel forking or serial mechanism or a serial-parallel combination. But basically, the idea was that there has to be a key-value pair that has to be stored at multiple nodes and must be republished periodically.

The root nodes will always republish it whenever it has a local routing table in which happens because of which, maybe there can be better another node which should be now the root node and the other nodes which are going to act as a backup, which typically happens when I am looking at a serial combination. In that case, it has to monitor the root node and keep on republishing whenever it finds that there is going to be the root node dies off. Otherwise, the root node will publish it, so it is only just acting as a monitor.

So, in case of a node death, no entry should be lost. Now, we further extend this, and now, we come to another application earlier; we have talked about voice over IP, where we were trying to find out who is the other endpoint and then trying to set up the call. Now, we are looking at a Distributed file system. In a distributed file system, we are looking at nodes with a lot of storage.

And they all are willing to collaborate now. So, once they collaborate, this storage is now whatever storage they have a disk which they have, they nearly now cooperate. This becomes a pooled resource, so all, all storage will become a pooled resource. And, now another important thing if a node dies off, what will happen only this particular part of the storage is dead, not the others.

And since it is a pooled resource, the whole storage will not be removed or will not be non functional at any one point in time; only part of it can fail. But if somehow the content which we are storing is stored on the file system, for example, we have in every computer file, we have a root directory. We have many subdirectories inside this, and then there will be further subdirectories, and then there will be files.

So, this is all being stored on a disk. So, instead of storing on the disk, and we know that whoever is the owner of this machine, there is only a file system available. We are now going to create a file system over by a clip pooling in this pooled resource.

If the same DHT best mechanism can be used to store the files, somehow, then it will be possible that I can maintain two copies at any point in time, even if a node fails. Whatever files for which this guy is responsible, there must be a second copy. Somebody else will become the root mode and accountable, and another backup copy will be created. So everybody's file will essentially be all distributed all around.

So removal of one single node will not be causing the loss of files. So this will be very something like we have Google Drive service, we want to find out if you're going to use. So it is just like a cloud. I use it on my local machine, but this gets in sync with Google Drive. Even if my machine dies off, I can always get another machine, or whenever the machine gets repaired, it will again do the sync, and I will get the same files here.

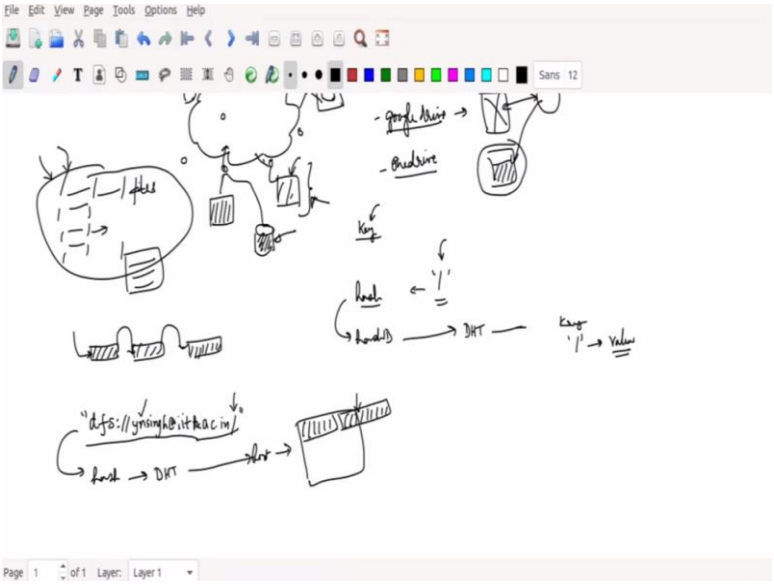
But their idea is that your primary storage is still is this. The drive is used as a backup. But in our case, even if you have a file, you do not maintain all your files here, here held in this cloud, which is formed by Peer to Peer system. So whenever you need something, you will retrieve it from here and write it back to it. Everybody does it.

So this collaborative use of disk storage but gives resilience. And this is not something like this where it is acting as a backup; though it is possible that you can form a backup, you can have an application that may be using some files. But these files are there on its local disk. But we can make a program where this program then keep on whatever changes which happens in this local space is being pushed into this particular the Cloud, storage Cloud which we are forming.

So, this will give the similar functionality that Google Drive or One Drive provides. Of course, we do have no applications on One drive and on Google Drive, where they use local disk only as a cache the way I have been telling this particular situation. And Google Drive also does work in this mode where it can create a backup on Google Drive.

So both modes are feasible. We will be first of all looking into a particular mode of operation, where you do not have everything on local disk; local disk only acts as a cache, but the part of the disk is part of this storage cloud, and your file system is on the storage cloud. How will this be done, as so many people contribute to each of them may have to have their file system?

(Refer Slide Time: 06:27)



So, how to handle that this situation? I cannot have this. For example, in this case, when I am creating a normal file system, I have a root directory, then within that, there will be subdirectories will be there. So, in any file system, and then we can move to subdirectories, I can move back and forth, and ultimately, there will be files which will be listed, and files can be of enormous size. But usually, when you are going on to disk, so disk has a very fixed capacity.

There are available sectors and the disk which are available. So it organizes on that part, small chunks. And these chunks are being used as a kind of a list. Like this is the head of the file, so you point to the next one. So this is how the long files are also written using multiple smaller chunks or basic storage units. But this becomes transparent to the user; he only sees it as one single file. I have to code something similar now.

But now there are multiple users, how to now create this. We do the root of a file; I will now be making a different kind of URL. We have to search for a key directly. And based on the key, I will find out the fragment, or multiple fragments, which I can put together and understand what is there in that file. So how will the root be figured out? So one possibility is I can take this root, compute the hash of this, and give me a hash ID and go to a node in a DHT network. I have still not talked about which particular DHT network.

It will be a separate layer. I will come back to this. So, in the DHT network, I will come to a root node. The root node will mention for this; there will be a corresponding value that is the key.

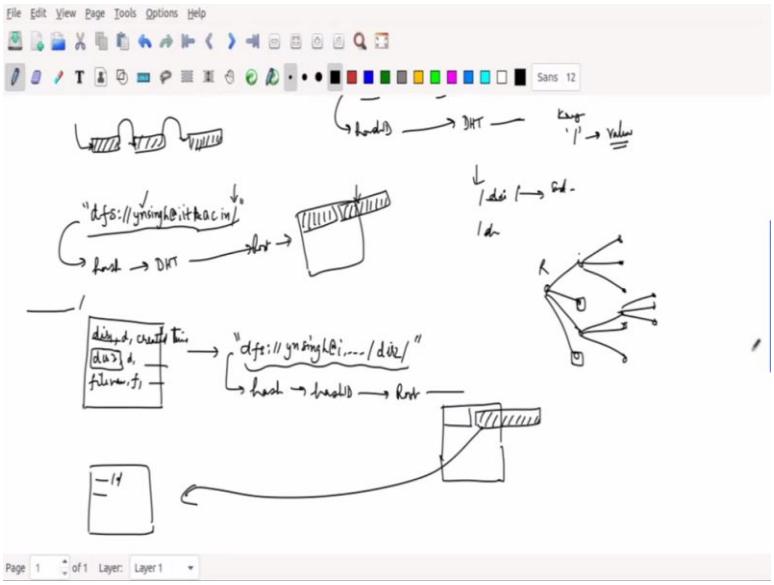
What is the value? Now the problem is for all different users; the resilience will be the same key; I need a unique key for every file that will be there. I need it should have a unique namespace for each user.

So what we do in Brihaspati 4 design we have created is what we call the protocol we are using; it is a distributed file system. Then we use email ID, for example, my email ID, which I have, for my file system, it will be there, which is the root indicator. Now, this is what is going to be my key string. I know to compute the hash of this, I will get the hash ID and search in the DHT network.

And I will go to the root node. The root node will have key-value pairs in its storage. This is what essentially they are going to maintain. This will be, the key listing will be there, which will say this whole thing. I know which I am looking at it, and there is going to be a corresponding value. And this value is going to be not returned to me; I have to ensure that this value, nobody should be able to read except this user.

Nobody else should be able to read this one. So that is what is security. You keep the things on your desk because nobody else can read them without your permission. But now you are using this, you are forming a cloud and using that. So that is why this has to be encrypted. But how that we will be done I will come back to that. So, now once this value comes back to the user, what is going to happen?

(Refer Slide Time: 10:10)



So, once this value comes, this value most likely will contain a text file. We generally call it an Inode file. So this is a directory I know, this is for the root directory, and of course, my user ID, which is going to be this is a unique identification. This has to be a unique identification of the user, each user. So each user's file name will now have a separate namespace for its file system.

And DFS only says it is a file system. So this will be now containing the subdirectory names. So it will most likely have directory 1; it will say it is a directory. It will also say that it was generated or it was created at what time. And similarly, maybe a directory 2 and it is a directory, then it can have your file also it is possible. It will say file, and they will have all creation times.

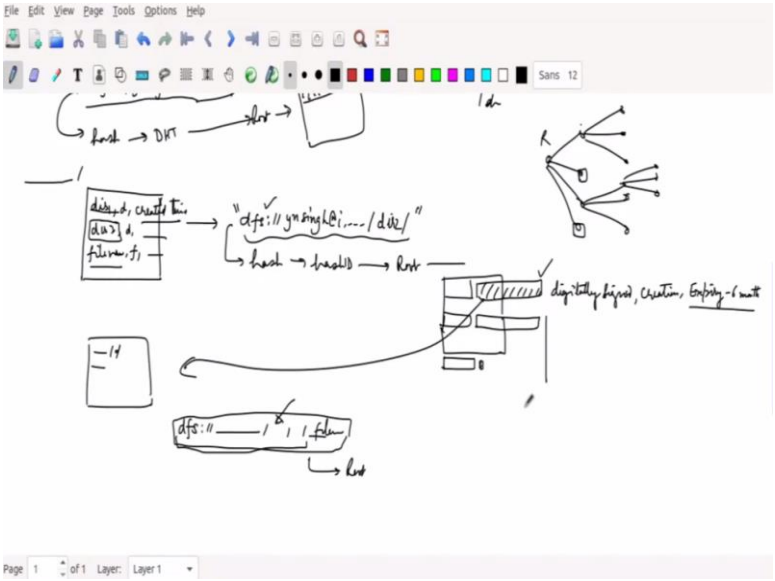
Once this thing comes back to me, I will figure out what is there in my root directory. Now I want to find out further what is there inside it. So what I will do is I will now form, I will take whatever query made, which I made earlier, I will say DFS, ://, I will put my email id again, / and then I will put directory 2. I will not say directory, so that I will put a /.

Now, the next string which I need to search. I will generate the hash of this, I will get the hash ID, and I will now go to its root. And that root node, wherever it is, will also be having its database where these key-value pairs are stored. And it will search for this particular key now. And again, for that key, I will have a value. Also, this has to be encrypted, I have come back to encryption, just now I just coming to that.

So this value will be returned; once it comes, it will also be again a file, which will say that all directories or files there inside it were created. And I can now figure out what is there inside. So I will be able to create a tree rooted at the user directly. Here is a directory name is in the root directory within that I then can create further (directory) subdirectories. And so on.

So basically, I am now creating a tree. So the file system is always being created as a trilogy. But this is the root; the root will contain either directory which may have further nodes, then it is a file it will not have further nodes, it is separate, that is a file, that is it. So this can further have subdirectories and so on. So I can recur, and I can go through the whole file system by doing it again and again, these queries and getting the values.

(Refer Slide Time: 13:31)



Now, the only thing is that how I will ensure that these are all not visible to somebody else remains. So whenever I am, I must have published this, so I have to see whether I have digitally signed this. So every entry which you put that will be digitally signed, you will have a creation date. And this was created. You also get the creation date in the previous I node, the same creation date, which is here.

So in fact, there can be for same for example, I write this particular file, and I want to search, I will now create, I will somehow find out what is the location, email id slash directory, that file name. And I will search for using them; I will compute the hash of this, using hash ID to go to

the root and find out the content. Now, I can read, I can make some editing changes, and I can submit the file. So what should happen?

Should I remove the earlier one or not? Maybe not the earlier one should not be removed. I should also put an expiry date here. Perhaps I am putting six months it has to be there. So but I have to ensure that I have to scan through all my file system and either refresh the expiry time or when I am overwriting I will create another entry with the same key value within six months I have to be on.

So when I am going to make a query I will get, I may get multiple values for the same key. For example, for this same key, I may get multiple values. I have to look at what is the creation time, and I will know the latest one. So that is why I have; now these are all different versions being created. Older versions will die out as time passes; the newest version will always die out much later because it has been made later. So if it is required, I can always backtrack and fetch the earlier files.

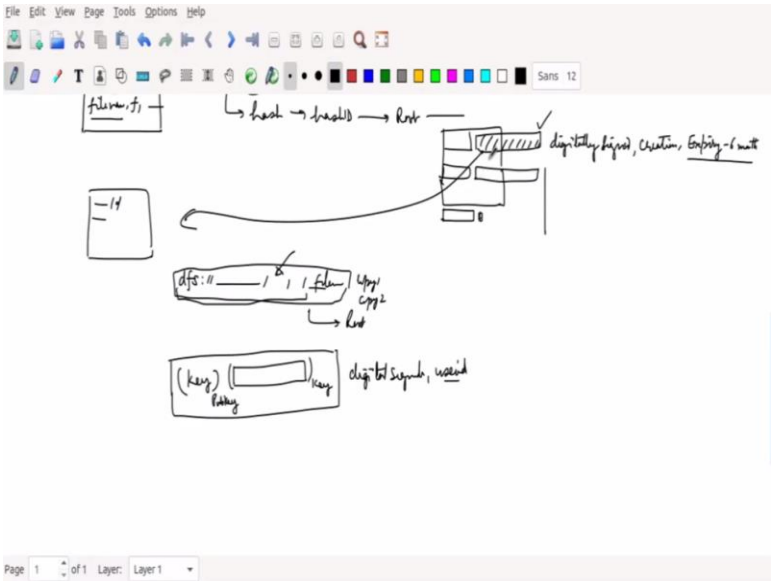
That is typically how we keep the things in the cloud; we do not delete the items. Even if I want to have a deletion of this file to be done, I will create an entry that this particular thing is there, but it is deleted. So I will also keep it in the I node and keep it here that is deleted, it is a null, I will just keep the status of this and then digitally sign the creation date and everything.

When all over the earlier versions are dead, this will also be the first no entry after some time. I have to keep track because nobody should be able to access; I should be able to find out. Now, when publishing this is also important; for example, I want to publish this file, just publishing this file at the root of the hash ID of this particular key is not good enough; I should also make an entry of this file the previous Inode.

So I have to go, I have to remove this file (now) filename, find out this whole stuff, and then retrieve that particular text file, and I have to republish it by making all the modifications that will become another version. So you will always be looking at the latest version of all the key-value pairs there. Some of them will be changing pretty fast, we will be creating more versions, but they will also be expiring very fast.

So, depending on some things, which will be changed very fast, we will be setting up the expiry time to be very large and some to be very small. So this has to be done by the user itself.

(Refer Slide Time: 17:07)



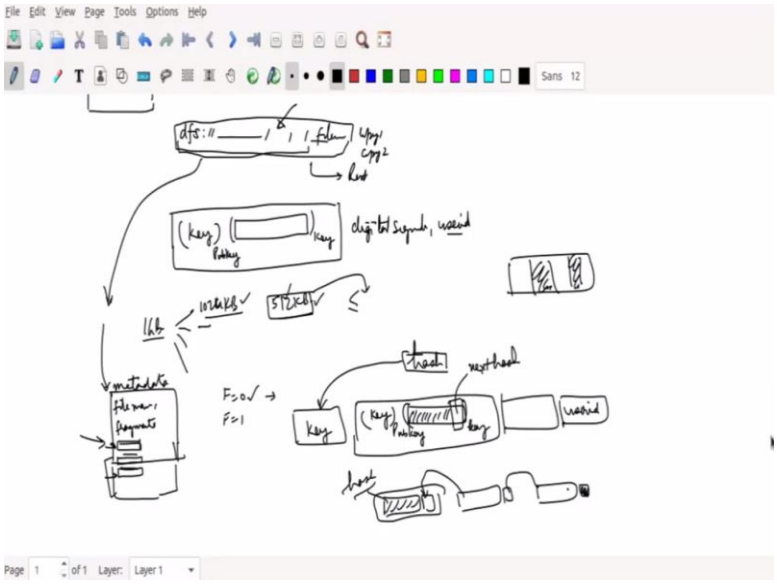
Now coming to the encryption part, how this will be encrypted, nobody else can read it. Usually, the content which will be put in, you will be having this content will typically be encrypted by a key. The key will be some random string; before you publish, you will generate that key. And that key is going to be further encrypted with your public key. Now, the guy who only holds the private key must be you; only you will be able to decrypt this whole stuff, but this will become your content.

And then you will be now putting digital signatures. And if there is a copy number of copies, all that thing for replication has to be done. So there may be at two different root nodes, the same thing might be published. So there is a copy one and copy two of the same thing. So I will essentially be taking this and will be appending copy one and copy two if the two copies have to be maintained.

And based on that, this same thing might be published at two (root) root nodes. So now there will be digital signatures here, there will be user ID. So user IDs certificate can always be retrieved from the base layer, as I have mentioned earlier. And this is how every content will be encrypted

and kept, so nobody else can read except you. But you are maintaining a file system in a distributed user cloud. Now there is one more thing we do, which I would like to introduce.

(Refer Slide Time: 18:36)



That when I am searching for this file name. But the way we have designed in Brihaspati 4, I do not get this file name because this file can be huge; I might have a movie file of say 1 GB. So with this key, I cannot hold 1 GB in one single place. Naturally, any file will be partitioned into fragments of multiple fragments. They maybe say 1024 kilobytes that is the 1 MB size or perhaps 1024 bytes or 1 kilobyte.

But it is your choice; it can be 512-kilo byte chunks, which can be there or 512 bytes. So you decide whatever it is. We have decided we are going by 512-kilobyte chunks, 512 kilobytes or loss, which is the maximum. It has to be less than or equal to this value. But that is a chunk which we are deciding. So when this file is, it means this file is being fragmented. So whenever you are going to access it, I will be getting a descriptor file.

I will not get the content. The descriptor file will be a text file, which will say that what is offered, it will contain metadata for this particular file. So this may contain the file name, maybe who authored it, what kind of file is there and as usual as we have discussed earlier, we will also have an F bit. The F can be 0, or F can be 1. So if it is 0 whenever you access this particular thing, this will be, the timer will be reset back.

So, when F is equal to 1, the timer will not be reset when it expires; it is just going to be purged out. So, the filename is there in that file name. Then it will say how many fragments are there in this case. It can now contain all the fragment list; what it will contain for each fragment, each fragment, will contain the hash value of the content. The hash can be this is just so that you can uniquely identify each fragment.

Normally no fragment, no two fragments will have the same hash value that will be almost impossible, unless the content itself is the same. It is being repeated in the file. There may be a big file, and then you chop it off, and you find this content, and this content is the same, if they are exactly the same, you do not have to bother to keep it twice, keep it only once but repeat the same hash twice so that you can retrieve the same thing in this places or two places.

So, one possibility is to have all the hash IDs of all the contents essentially. Each of these content, which is a content fragment, will be further encrypted in the same fashion. It can be through the same key, or it can be different does not matter. So that is independent, that is this layer, or this particular functionality should not bother about it. We will have a public key here. So this will be the encrypted content.

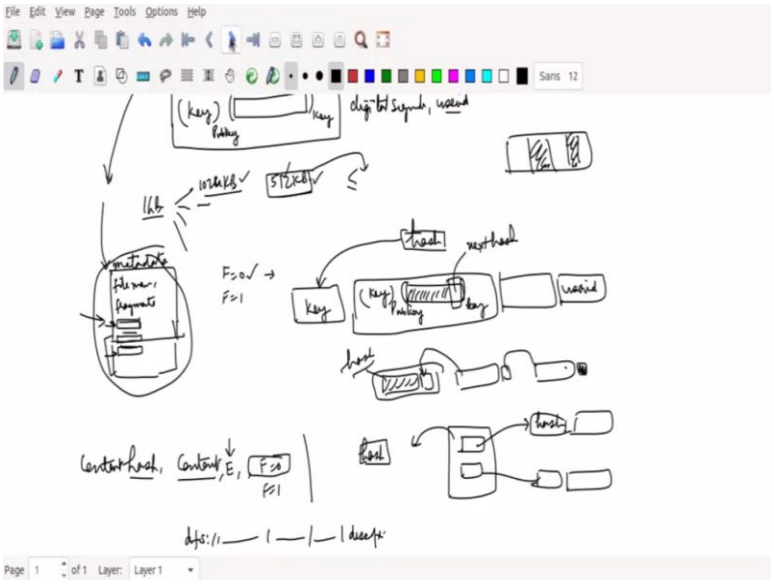
And then you will have digital signatures; as usual, the tempering can be taken care of. And of course, you will have a user ID, which will be there. And you will have the hash value, which is going to be the key to this. So you do not even have to keep the fragment number. So that fragment numbers are being assigned here. Now, there are two kinds of, two ways this can be done.

Now inside this content, I cannot key encryption; I can have a payload of the file the fragment size; I can also add another field inside this; this can be the next hash of the content. So, you break the content, you compute the hash of this, which will be its key. For the following content, you compute the hash, and you append it here, so this is what will be stored with this particular hash.

So, in that case, if you do it in this way, serial chaining. In that case, you do not require all the hash values, you only need the first one, and it will keep on doing it till it, it will fetch you the first content fragment from that the second one, the last one will not have this thing, this will be, this will be (proved) absent, so you have got the whole file. And you can recreate that. When you

have multiple hashes, you can parallel fetch all the fragments and then stitch the whole file together after decryption.

(Refer Slide Time: 23:21)



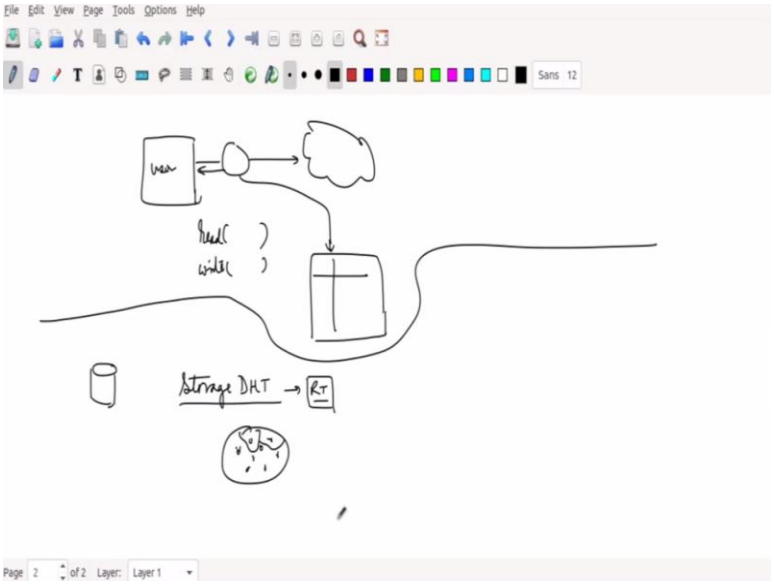
Normally, the actual content will always be stored with the contents hash, which will act as a key and the content. And content can be encrypted or not encrypted. And this can have an F is equal to 0 and F is equal to 1 bit. Normally, when F is equal to 0 is there, these are kind of public files. So these will not be encrypted. But you can have a; you can write down whether it is encrypted, not encrypted can also be put as a field, inside the key-value pair or the data secure, which is saved.

Now, there will be some, for example, you have a movie file, and you want to do it, so you create fragments of the movie, descriptor you make, the hash value of that segment is accessing each fragment. That is a key-value pair, and this goes in a descriptor file. And I can compute the hash of this. And that is being used to identify the file itself. For movie files, typically, you can do that because then, I will keep this in my file system as a link; it is basically a link you are creating.

So I can simply have a marker called DFS colon slash, slash something directory, and subdirectory, then I can have a descriptor file. So descriptor file will be something, will be something like this, where I can store metadata, this movie files, so and so, this is the hash through which you search. And using that hash, I will now go to this hash to find out this

descriptor file for either a parallel axis or a serial axis. So this way, I will be able to create a distributed file system. Now let us see how this is going to be implemented in the application.

(Refer Slide Time: 25:18)



In Brihaspati 4, the way we have done is, so we have an application that will access the file system. So it uses an API. So mostly, it is an essential read and writes options. So this read and write APIs will have arguments. What is the current directory that can be full length, or it can be concerning the directory's current position, and this goes to function implementation, which then accesses the DHT network to access this stuff?

So, we also maintain a cache here. So whenever a fragment or something is coming, that will be stored in the cache. And whenever read has to happen, it will first of all search in the cache, if you cannot find in the cache, it will search in the DHT if the failure comes from DHT it will tell this does not exist. If it exists in DHT, it will come back, and it will be first of all stored in the cache, and then it will be given back to the user or whatever application you have written.

And while writing again, it has to be written in the cache and the DHT also. Remember, it is a version number that will change automatically, and the version will be essentially decided by the date when the document has been saved, key-value pair has been saved. So every file essentially will be split into multiple parts and spread across the whole network.

Now one of the challenges is the basic Distributed file system. Now one of the challenges that I am assuming that every node has storage. So every node may not have the storage; that is one problem. When we create this system, so merge more storage, they have only have to come up and create a separate DHT layer for them.

So there is a separate storage DHT layer, which is used for providing this storage service. Other nodes normally will be the user of this; they can be leaf nodes to this DHT. But they may not; they are not participating. If DHT does allow in the storage, they can store their file system there without contributing anything back. But this leads to another column called Free Riding; we will try to look at that column later on. So, this storage DHT is, has to be created as a separate thing; only some of the nodes will be participating. So there will be a separate routing table for this.

These routing tables have to be exchanged across the storage DHT and maintained continuously. And this will again be the routing table; we are using the same thing, what we had used in the base tapestry code, hybrid only we have been using, a logarithmic partition-based system. Now, the only problem is that some people will have less storage, some will have higher storage.

And they just randomly get placed in the node ID space. Now the problem is node ID space. If they are just randomly placed, then there is a problem. So they will be responsible for some of the hash IDs, so almost everybody will be responsible for an equal number of hash IDs. And since it is a uniform distribution, they all should contribute an equal amount of the store now, which will not be visible; some will be doing less, some will be higher.

So how to solve this particular problem. So in the next video, we will be looking at this situation how this is handled. So unequal storage, if these are there, how this storage problem will be resolved in the storage DHT. So now, as of now, we know how the DFS is going to work. You have to solve the next problem of the storage DHT is non-uniformity. And then, of course, how if the people start doing Free Riding, they have storage, but they declare they do not have anything.

But they start using the storage from the guys who are providing generously. So this, this is cheating. This is, this should not be allowed. Can we make build up a system by which this can be discouraged? So we will try to look at one particular scheme, which we have planned for the

Brihaspati 4 system in one of the videos in future. So with that, I close the lecture for today, and we will soon meet with the next video on Non-uniformity of the storage in the storage DHT.