# Peer to Peer Networks

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

**Lecture-3**

## Building DHT Networks

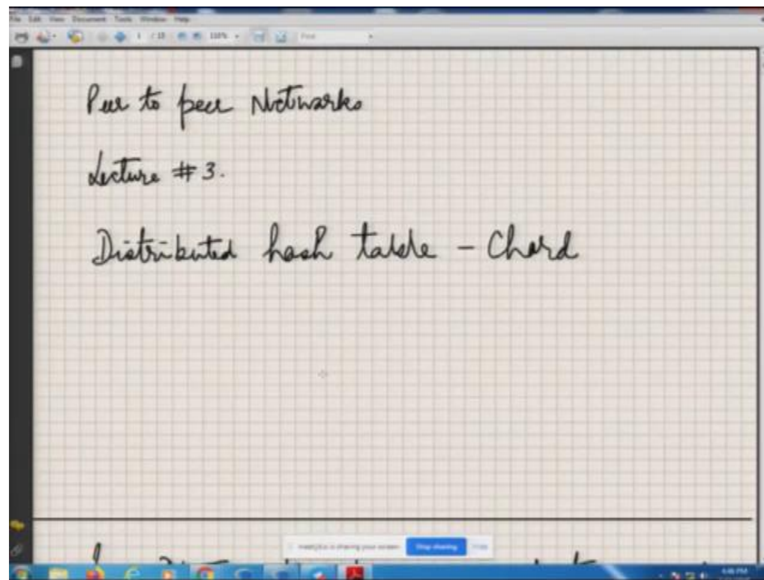
Welcome to lecture number 3 for this course on Peer to Peer Networks. So in the previous lecture, I had talked about Voice over IP systems and how we use the SIP server as a centralized entity. And we also then explored how we partition the whole Internet or whole world into domains, and each domain can have its SIPs, SIP server, and Session Initiation Protocol servers. And these SIP servers will also talk to each other. So they have kind of been acting as an entity on which everybody has faith.

And they are used to set up the connections between the two endpoints or two users who are communicating. There is no mutual authentication required because that has been taken care of by the SIP server. SIP server everybody authenticates a SIP server, and the SIP server has known about each other through something called a SIP record, which is equivalent to an MX record in DNS servers.

So, but then I also further mentioned that it is possible, we do not require SIP servers; we can make a system where all the nodes together can start storing the indexes. They can start doing indexing of endpoints, endpoint addresses, and phone numbers to the endpoint address mapping. Your endpoint address is the IP address, the port number, and the kind of transport acceptable to an endpoint. And once two endpoints know each other's address, they can communicate directly without having anybody as an intermediate.

The only thing that now has to be done is there is no indexing server; indexing has to be done in a distributed fashion. So, I will talk about an algorithm, which can be used; this is called a distributed hash table.

(Refer Slide Time: 2:27)




In the previous lecture, I have talked about hashing, which rendezvous was hashing, where everything gets uniformly distributed whenever a node dies off. All its responsible entries are being truly done, and we talked about consistent hashing, this is what we are going to use, and we also talked about inconsistent hashing. So, the distributed hash table, which is the chord which I will be talking about today, is an example of consistent hashing.

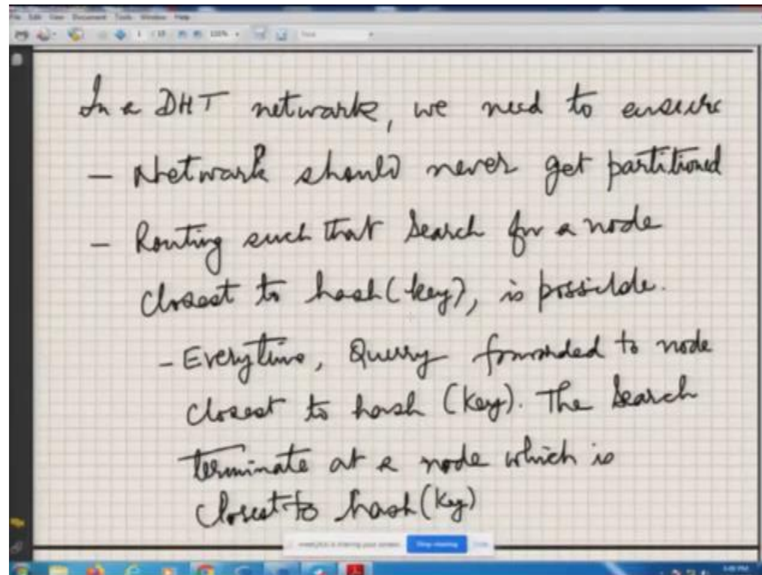
We do not use rendezvous hashing because normally, we assume that there will be many nodes that will be participating as an endpoint. They will also be participating in the indexing, distributed indexing. So, on average, it will not be a very large number of entries that the node will maintain. So, on average, it will be only one entry per node that will be maintained.

Some of them may not be maintained in that case, some others will be maintaining 2 or 3, but certain nodes can act as leaf nodes. So, leaf nodes do not participate in distributed indexing, but they have endpoint addresses to exist. I will elaborate on these leaf nodes and the core nodes or the nodes which participate. So I will be talking about the chord.

So, a chord is a kind of distributed hash table, which is very fundamental. This is an example that we will describe how multiple nodes almost all nodes can participate and can talk about, can store the phone number to the endpoint address mapping distributively. So, let us move

ahead and see how it happens. Now, what is the objective when we are going to build this DHT network?

(Refer Slide Time: 4:26)




We need to know one of the very important criteria, so chord does satisfy that all DHT network algorithms should satisfy this, that whatever happens if a node is participating in the network, it should never happen that one node dies off. Few other nodes are now separated from some of the other nodes.

So they cannot talk to each other because somebody who is an intermediate has died. So that network partitioning should never happen. So this has to be guaranteed. So this actually can be done by something a technique called logarithmic partitioning. So, that is what is used almost by all DHT algorithms. This also reduces the routing table size. So, this chord is an example of that. I will generalize this later.

Now, routing has to be such that you will search for a node closest to the hash of the key. Now, in our case, we are looking into Voice over IP system. So, the telephone number is my key so that I will compute the key's hash function. I have talked about hash in one of the previous lectures. So, the hash is a computation. When you push in the key, it will just do some randomization and create a fixed number of bits.

So, whatever be the input size, the output will always be a fixed number of bits, and this number of bits will be the same as that for that node IDs. So, the node that will be closest to this hash, we should be able to reach to this particular node, and this closeness is what will be the metric for the root; we call it a root node. The node which is closest to the hash of the key is the root node for the key. So that is the way we define it.

It can be, it is also possible in some network, in fact, in chord it happens that a distance from A to B, node A to node B or node A to hash ID B and from B to A, they are not same, they are different, it is the asymmetric distance which exists. When we define a root node, we have to specifically tell whether I am looking for the hash ID's root. So it is the distance from hash ID to the node ID, which has to be minimized or node ID to the hash ID, which has to be minimized. Depending on that, my routing tables have to be framed.

So we should be able to build up a routing mechanism; the node will only look at the hash ID, which has to be routed to its root. The routing table or our neighborhood information will find out the best match and send it forward to that guy, and this should keep on happening until the entry until the message reaches the root node. When the test is done at the root node, it will find out that the node itself is the only closest entry; no other entry in the routing table exists.

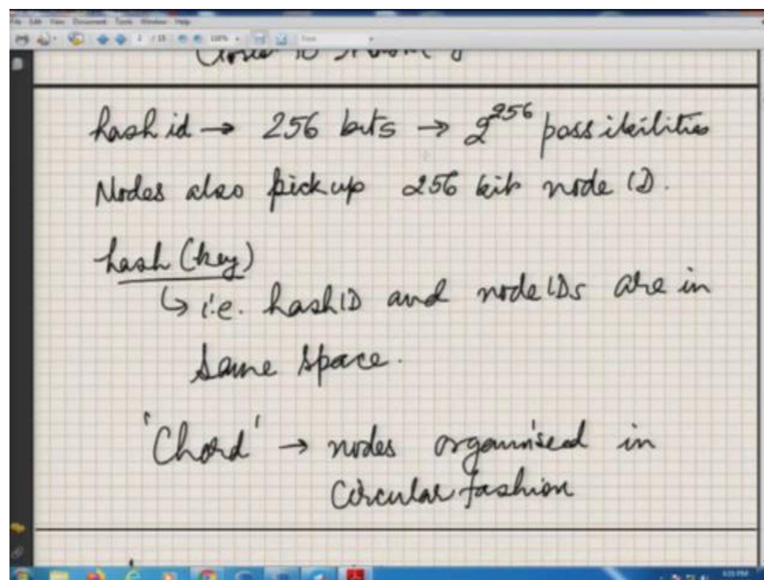
And that should guarantee that it is indeed the closest node; no other node should be the closest one to the hash ID. This indeed should be a root node that has to be guaranteed. So, this is the query forwarding mechanism.

So, whenever a node finds itself to the closest one, the search will terminate, it will find out it is a root node. It will now look into its database and find out if that key, that phone number is there in the database or not.

If it is there, then it will check what the corresponding endpoint address is. And now it knows this is an endpoint address, so whoever has sent the query, he will also send his node so that again you can use the same DHT routing to send a reply back. It can be done through a DHT routing, or if the source node has sent the IP address and port number, direct communication can also be done.

The source node will know the corresponding endpoint address for a phone number; it need not go to a server and find it. It is just finding out this phone number who must be the responsible guy holding this entry. Just find out send the message to him; it has to be routed, and the guy will respond. So the actual user who is holding the phone number will also publish in the same way. So when he is publishing, the entry is created by the root node. When somebody is querying, the entry is retrieved from the database and sent back to the querying node.

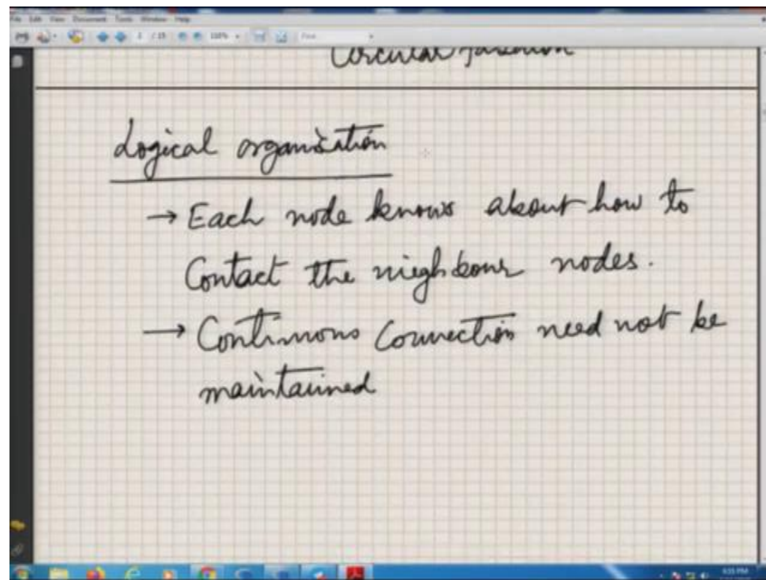
(Refer Slide Time: 8:56)




Now this hash ID size or the node ID size, how much it should be. So, this can be any bit. The example which I will be covering is based on 4 bits. But in real life things, we mostly choose a larger number of bits; generally, 256, 512 or 1024, which means a total $2^{256}$ possibilities for the node IDs exist, and each node has to pick up from one of these values. So and but each node ID will now be of 256 bits and so hash value.

So when I put a key into the hashing function, the hash ID, which I get, should also be 256 bit, and that is how the mapping. Then whichever is the closest node as per the definition, which I will define as I go along for the chord, what is a definition for the distance, whichever is closest to the root node. And that guy must be holding a phone number to the endpoint address mapping. So the chord is a circular organization; all nodes are placed in a circular fashion. So how this will be done, let us see.

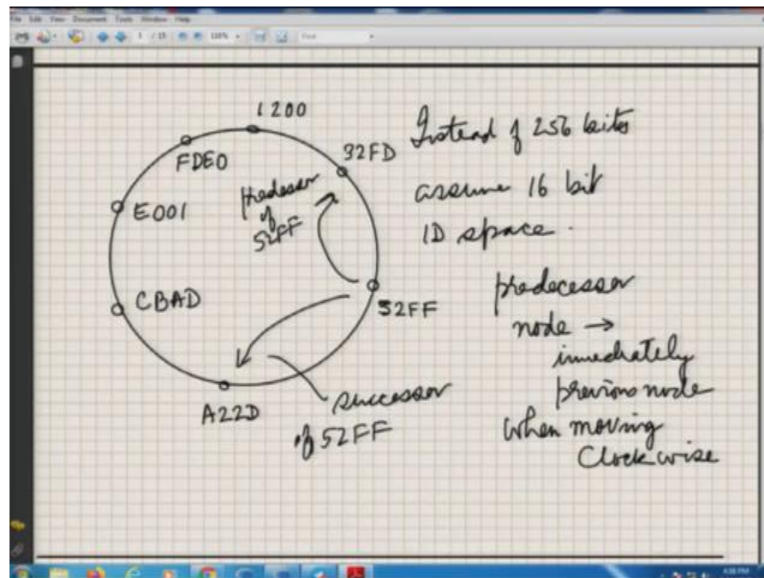
(Refer Slide Time: 10:14)




Now, remember, it is a logical organization. There is no need that if in the circular organization, somebody who is next to me or who is previous to me I am, I need not maintain a live connection with them. I just simply need to know what their corresponding endpoint address for communication are. So whenever some message comes to me, I have to give it to my next guy in the circle, we call it successor, the guy whose previous to me I call it predecessor.

So the guy who has a lower ID than me is a predecessor; the guy who has a higher ID is a successor. So I need to know the endpoint address of the successor. I can then set up a connection TCP or UDP or HTTP and transport the message to him. I need not maintain the connection all the time, so it is a logical organization, so there is no physical wire. It is a logical thing, and connection needs not to be maintained all the time it is based. It is done on a need basis.

(Refer Slide Time: 11:10)



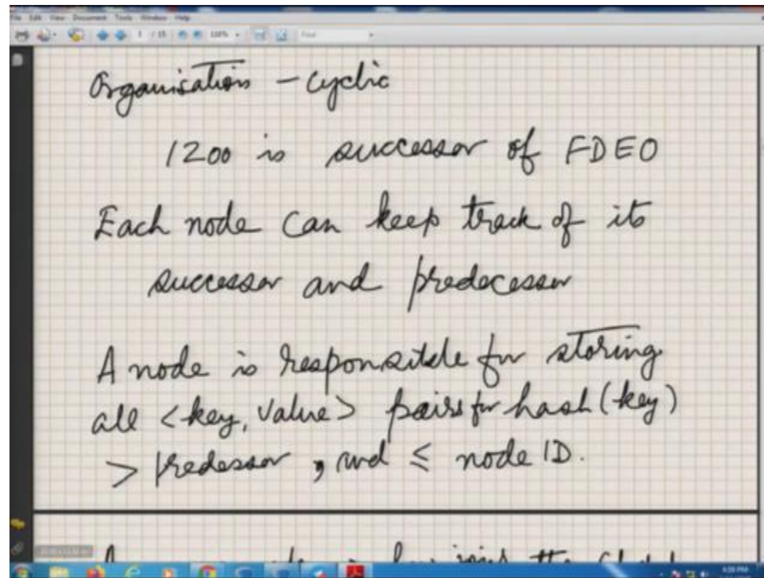

This is an example I am showing; this is a circular ordination you can see, and I have given a 16-bit number. So 16 bit means, each of the digits, there are 4 digits, each digit is represented by 4 bits that is why I can use a symbol from 0 to F, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 then A, B, C, D, E, F, so which is 15, 0 to 15 which is total 16 possibilities. And their 4 such digits so 4 into 4, 16-bit node IDs I have taken. And I have arranged in such a way you can see 32FD if you look at, so in 32FD, you will find out 1200 is the lower node ID.

So, between 1200 and 32FD, there is nobody else so, you can look at all possibilities there is nobody else indeed. And the guy who has the smallest node ID, but higher than 32FD days, the next successor of 32FD, which will be 52FF and so on, this goes in a circular fashion. So, when it goes to FDE0, it just completes the circle. So, you have to go from it again will go to FFFF. So there is no FFFF; it then goes to 0000. So FFFF is a predecessor of 000. If they exist, they do not exist here.

So 1200 is the successor of FDE0. So, we are not using 256 bit here in node ID space. Still, we are using 16 bits and immediately predecessor node; immediately previous node is the predecessor if I am going in a clockwise fashion. So when you are moving clockwise, immediately the previous one is predecessor, and immediately the next one is successor. And the very simple thing which we can do here I will modify this definition.

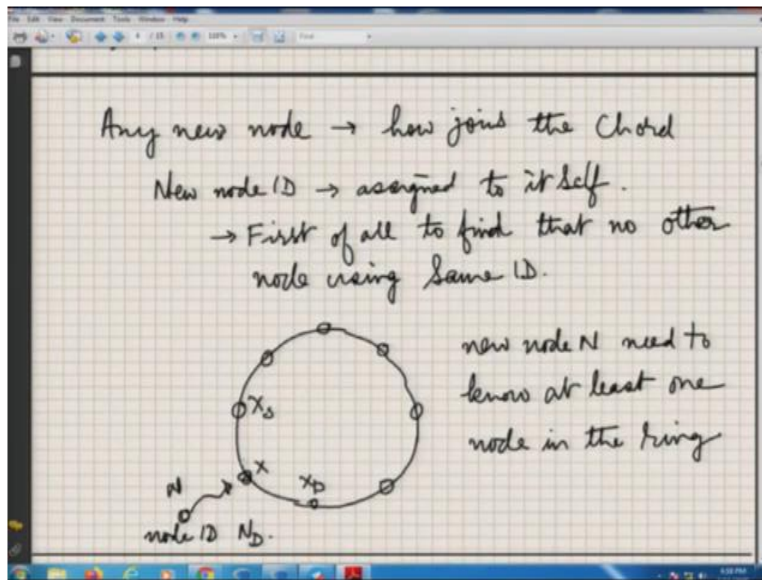
So all node IDs if you look back here, between 32FD so any node any hash ID which is higher than this, but lower than or equal to 52FF, all those hash IDs, if they are coming by hashing of the key all those keys will be stored in by 52FF.

(Refer Slide Time: 13:41)



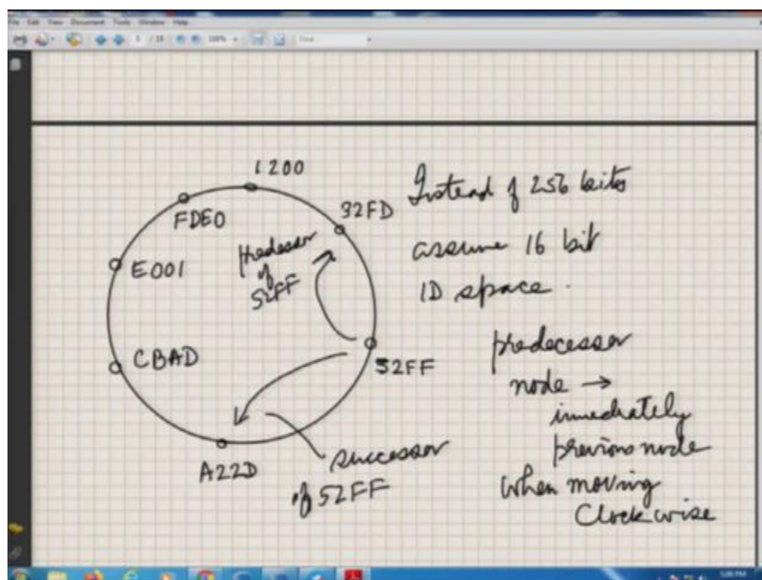

A node is responsible for storing all key-value pairs such that the hash of the key is greater than the predecessor, but it is less than or equal to the current node ID. So the current node is responsible for holding up all these key-value pairs. So key in our case Voice over IP system will be phone number and value will be the corresponding endpoint address.

(Refer Slide Time: 14:06)



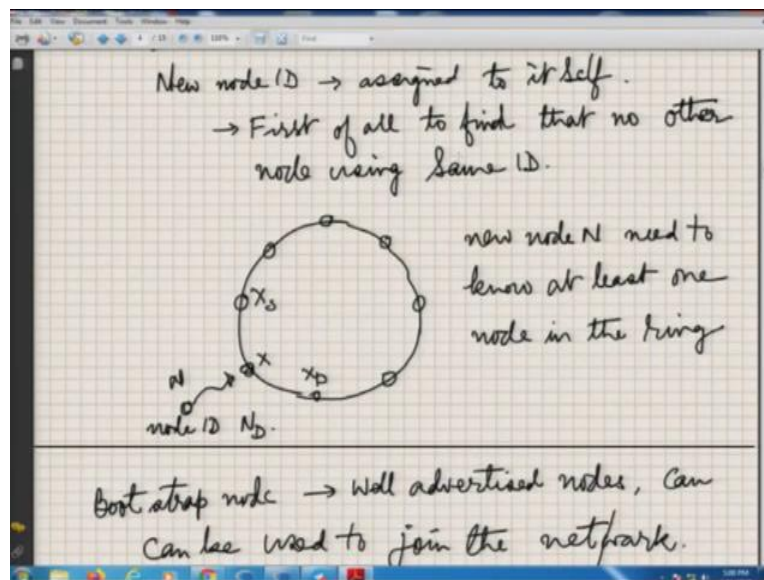

Now, the question is, if any new node comes in, how will he join the chord? How will it join this whole organization? It has to be placed at the right point.

(Refer Slide Time: 14:19)




If you are, for example, joining here is by say 4200. So, 4200 can only be placed here between 32FD and 52FF. So, wherever, whichever node it picks up, it has to should come here ultimately. There is no other option for him. So, that is the organization. So, how will this be happening?

(Refer Slide Time: 14:37)

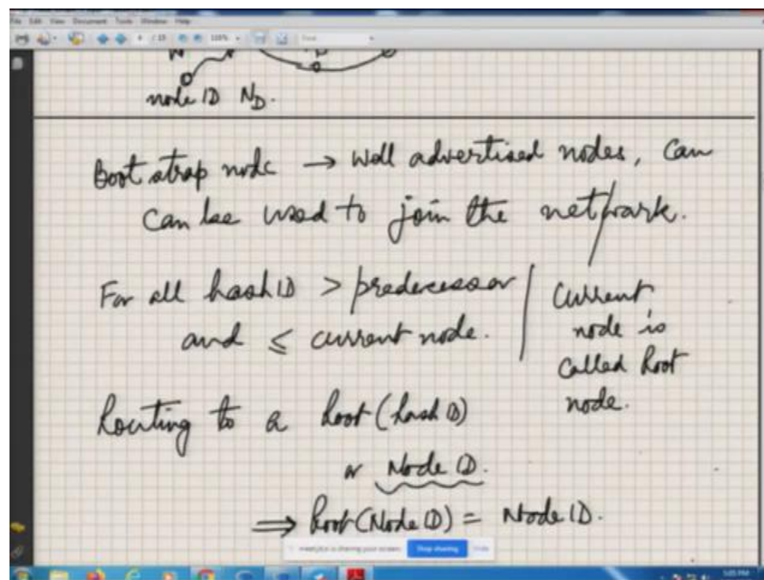



For example, in this case, we have this circular organization. So, in this chord organization, we have a node X that is already part of the chord. The new node, actually N, will come and let me have this node ID be $N_D$. Now, this $N_D$, we have to figure out it will be between which two nodes, and it has to be placed there.

But the first thing is that no other node should be using the same node ID that is very important. If it is already using, then there will be a clash. Every node should have a unique node ID that is one of the key characteristics of any DHT. In this case, when it generates a new node ID, $N_D$ randomly, it will then first find out somebody how this will be happening.

So normally, a few of the nodes are called bootstrapping nodes. So in any DHT network, they are already well-published ones. So, if you want to be a node, you will first get that list and become your neighbour. And when you want to attach, you will always be talking to them.
Node N will talk to X, and it will say I have picked up this node ID $N_D$; you, please go ahead and find out if a duplicate exists or does not exist. So then the node X will find this thing particular out.

(Refer Slide Time: 16:05)



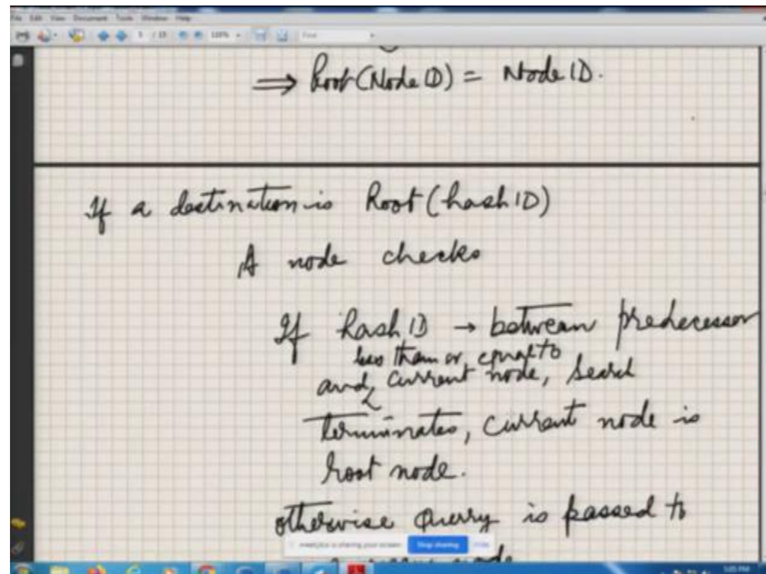

So, now this bootstrap node is a valid advertised node, as I mentioned, and it can be used to join the network. So that is a very fundamental thing; it always happens. I am now routing to a hash ID, R hash ID. So, this I will whatever is hash ID, I will the root node of this I need to find out. So whatever this new $N_D$ is there, I will put this $N_D$ here, and I will try to find out the corresponding root node. There is no hashing required in this case, so $N_D$ itself is acting like a hash ID.

So if there is somebody else who is having the same $N_D$, I will use this route search. I will keep on passing till a node figures out I am the root node. So if that node's node ID is equal to $N_D$ itself, then it is a duplicate. So that is the case which I am mentioning. If the root of the node ID is a node ID itself, it is a duplicate entry. And if it is some else, it means it lies between this current node and the predecessor, and no node ID is using this particular value $N_D$ as of now, and that the new node can use $N_D$.

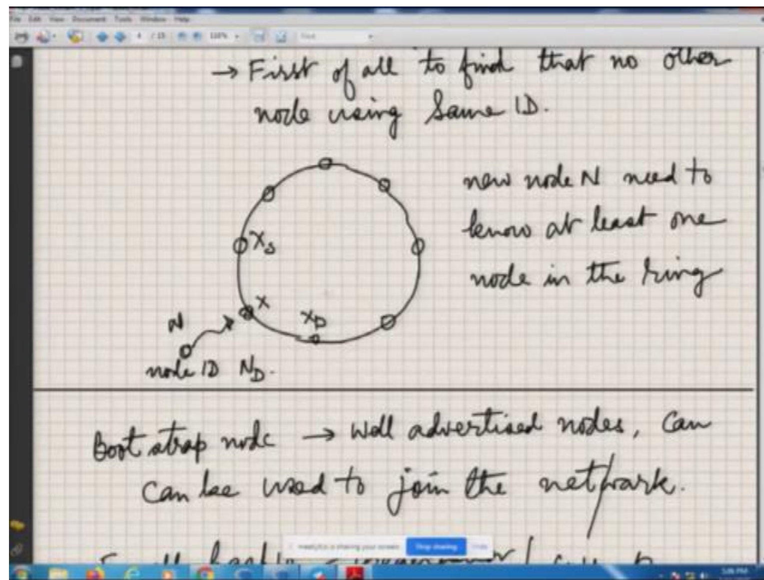
So, essentially the root of the value $N_D$ should be communicating back to node X and informing him that the root of $N_D$? If it turns out to be $N_D$ itself, it is a duplicate. So, in that case, X will tell node N that you cannot use $N_D$; please regenerate your node ID again. So it can randomly pick up. I will give you a method by which this has to be done because this has a flaw. As of now, there is a security loophole if you can attributively choose any node ID for yourself.

Because in this case, there is an attack called denial of service attack. So you need not understand it now. I will explain it later on, but you can search on the Internet about it if you want. So that attack actually can happen. But because of the conceptual reasons, I am currently assuming this guy gets a node ID, but how it gets it, I will explain it later in one of the later lectures.

(Refer Slide Time: 18:03)



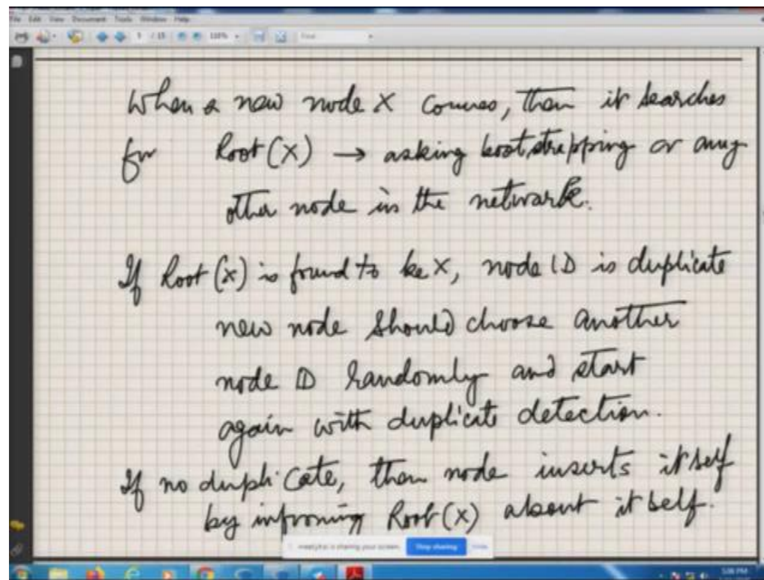
(Refer Slide Time: 18:37)




In this case, the root of the hash ID has to be found out. A node will check if the hash ID is between predecessor and less than or equal to the current node; the search will terminate the current node is the root node. So that is logic; otherwise, it is going to be passed to the successor node. You can keep on doing it. The problem here is if the $N_D$ is supposed to be happening here, this is the root node. So this will keep on moving in the circle and will come back here.

If you have thousands of nodes, so thousands of hops are required before the query can be resolved and replied to. We will solve this problem, but currently, let us only keep it as a circle. So it takes time.

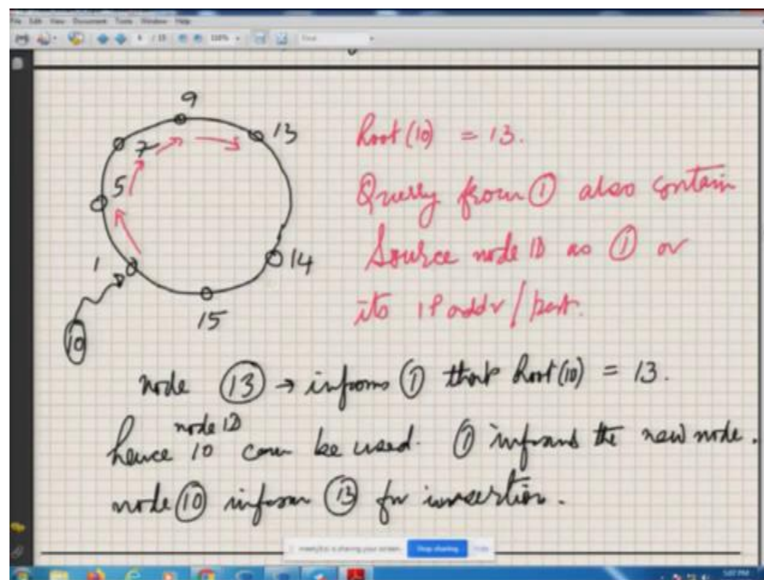
(Refer Slide Time: 19:01)




So, this is how it will be happening; again, I am explaining that new node X will search for the root of X, the new node has given this value, and it will ask the bootstrapping or any other node if it knows. The root of X is found to be X node ID is duplicate; otherwise, X can be used.

Now, in this case, the node which is starting the query, which is a bootstrapping node or who is a participant in the DHT, has to actually in the query itself has to mention its node ID so that the reply can come back through DHT or it has to give IP address and port number, endpoint address of itself on which the root node can respond back directly. And then, this guy will talk to the new node and tell him whether you can be inserted or not. Now, if it can be inserted, how it will be done.

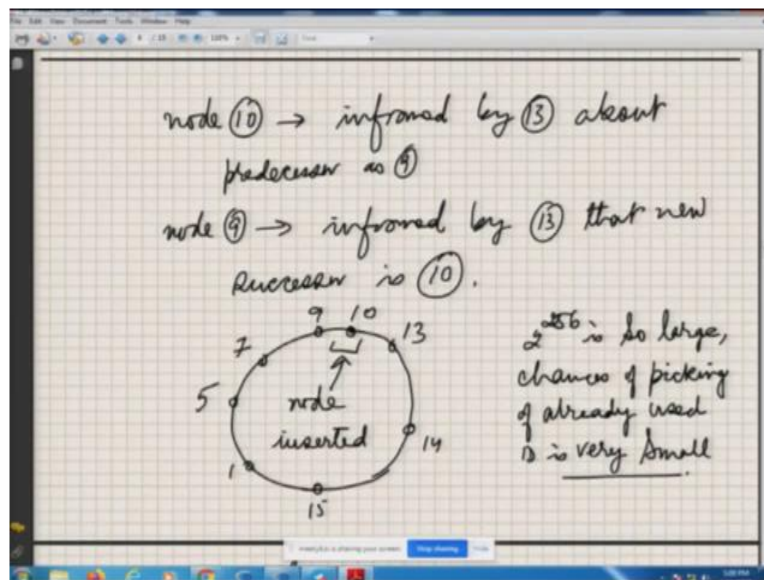
(Refer Slide Time: 19:53)




In this case, for example, I am showing a case here. So, this is again a small chord DHT. I have these node numbers 1, 5, 7, 9, 13, 14 and 15. Node 10 is the one that gets inserted, and when the root of 10 is searched, it will turn out to be 13. So, 13 will respond back, and it will say there is a unique one so that you can use it. So, node 13 is being informed by 1 that the root of 10 is 13. So node 10 can be used. So, 1 will now inform the new node and node 10 will inform 13 for insertion.

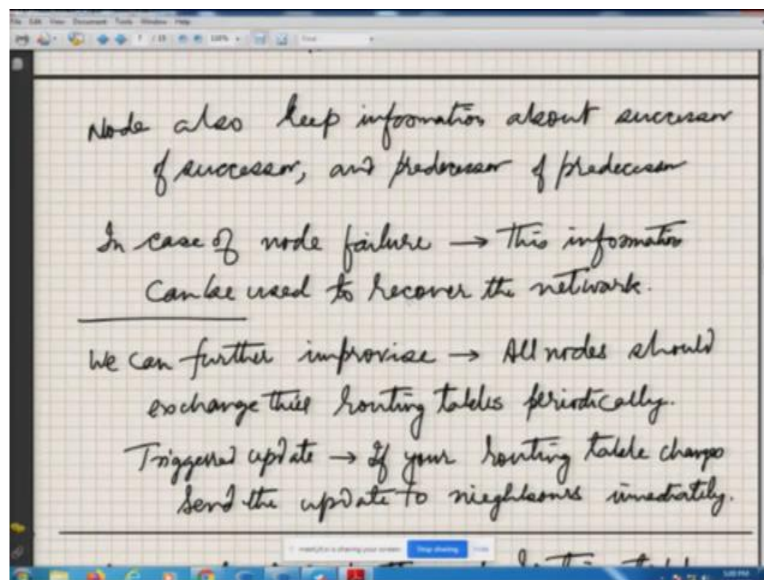
Now the 13's endpoint address is available to node 10, node 10 can directly tell him that you kindly insert me. So 13 has to tell about his predecessor to node 10, and it should tell node 9 about node 10. 9 will now insert 10 as a successor, 10 will insert 9 as a predecessor 13 as a successor, and 13 will add 10 as a predecessor, so 10 will get inserted into the chord ring. So that is how new nodes arrive. Every time nodes will keep on coming, and this will happen, nodes also can be removed actually. So normally, one of the best ways of handling this is that we can make a self-healing system that is done in actual implementation.

(Refer Slide Time: 21:26)




Now you can see node 10 is informed by 13 about predecessor is 9. Node 9 is informed by 13 that the new successor is 10 and 9, 10 and 13 make the entries, and you have got the ring ready.

(Refer Slide Time: 21:41)




Now nodes also need to keep the information about the successor of the successor, basically grand node, grandparent and grandchild. Now you also need to keep who is the successor of successor and predecessor of a predecessor. The first is the grandchild; the second one is the grandparent, and in case of a node fails, simply connect to your grandparent or grandchild.

So, depending on whether a parent or child has died, and this way, you can immediately heal if a node failure happens.

If you find your parent is dead anyway, you have to know just connect to your grandparent because that information is also there. Even if it is not there, you can still heal; the problem is to be done in a slightly different way now. The best algorithm used now is that all nodes will maintain these kinds of neighbor tables; they are also called routing tables in DHT.

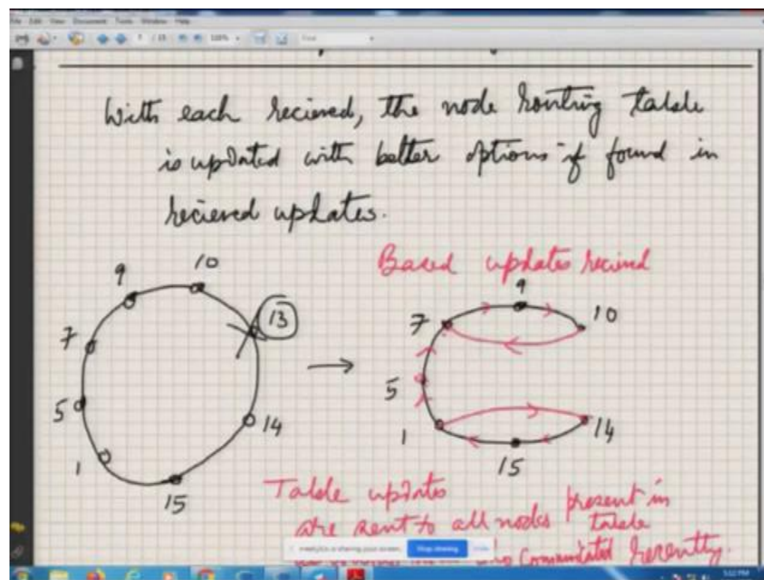
And periodically to all the neighbors who are listed, this neighbor table is being communicated. If my neighbor table changes, then also I will communicate with all my neighbors. So, as the neighbor tables are being received from my neighbors, I may get new information about new nodes, improving my current neighborhood table or routing table.
So this process will keep on happening. And as I get a better option for the nodes which can be put in the routing table, I will keep on updating them, and at some point, the routing table will stabilize.

So that is what will essentially be the chord; even if a disturbance happens for some time, it will stabilize back to another stable ring topology. And I think one of the things which we do is triggered update in all these.

Whenever my local my routing table changes, I should immediately send the copies to all my neighbors. If there is no change, I will do it after the period, after which it should be done. So, maybe every half an hour you are supposed to send an update, you will keep on doing it every half an hour. But if a local change happens, table change happens, then I had to do it immediately.

Consequently, wherever the table changes will impact, only those nodes or tables will be updated very fast if there is a change that is supposed to happen. It will not go into an area where table change will not happen because of the actual change, whatever is happening. So that triggered update kind of stabilizes routing tables very fast.

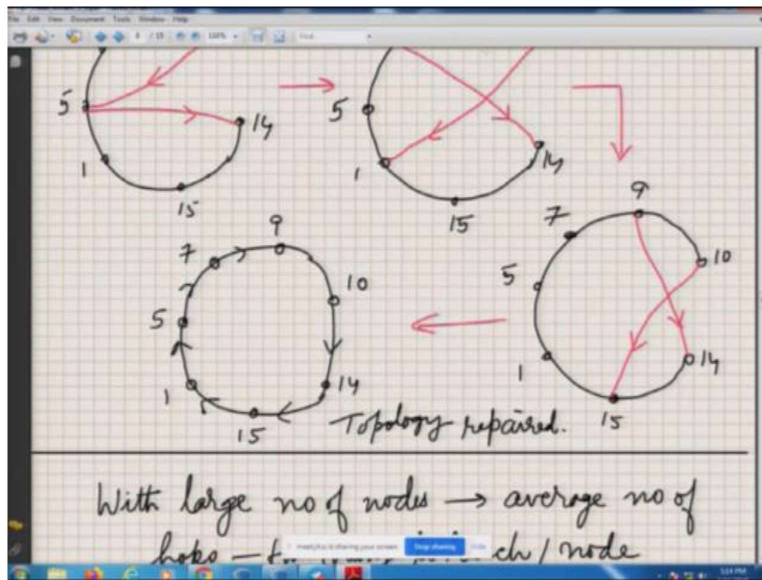
(Refer Slide Time: 24:25)




Now, what will happen when the nodes, this node dies off? For example, node 13 has died off here, and I am assuming that we do not; we are not maintaining grandparents and grandchildren. In that case, also it is going to work. For example, 10, we get from 9 the information about 7, his neighbor, and 9 telling the neighbor table update has been sent to 10. 10 will know I am not getting only entries coming from 9; for 14 it does not have entry 13's entry will expire because 13 is dead, is not sending the update, periodic update, it will make an entry for 7.

Similarly, 14 will make an entry for 1, so their kind of 2 subrings will get created. And of course, the rule is whenever somebody is informing me; I know whatever nodes I know which even not are there in my neighbor table should still send my routing table update to them. So, as a consequence 7 will send its update next time about 5 and 9 to 10 also as well as to 9 as well as to 5. So 10 will come to know of a new node 5, 14 will also come to know of a new node 5, and they will update, and that is a new update which will happen.

(Refer Slide Time: 25:32)




In the next step, 5 will send the information about 7. 1, which are routing tables, to 10 and 14. 10 and 14 are not in the routing table, so their information is not being distributed by 5. 10 will now update to 1, 14 will to 7 and so on. Ultimately ring will get formed 10 will get connected to 14, and topology repair will happen. And since it is done because of the triggered update, this will keep on happening pretty fast.

(Refer Slide Time: 26:06)

With large no of nodes → average no of hops — for Query to reach / node Root

⇒ N/2    N = no of nodes.

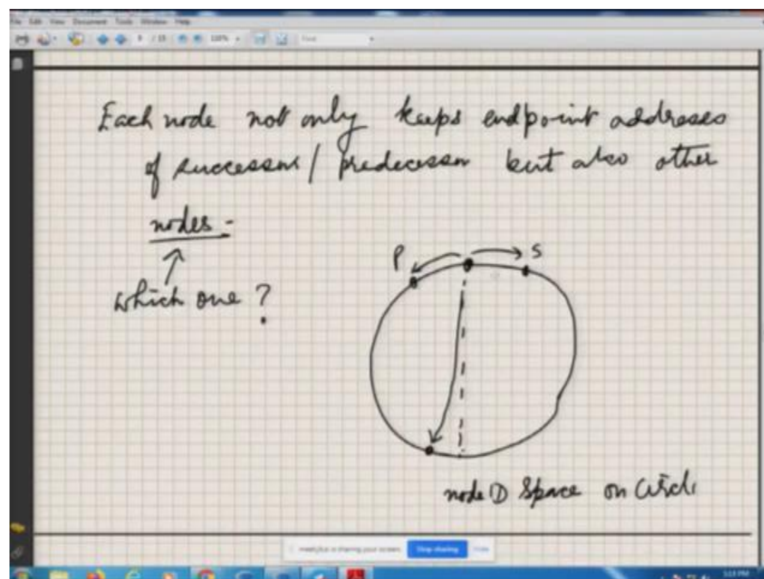
How to make it fast?

Each node can keep information about more nodes → Finger tables

If I create only a ring, there is a problem. If a large number of nodes average number of hops for a query to reach a root node will roughly be N by 2 half of the circle, on an average; otherwise, you have to do either a full circle with some probability, sometimes only with 1

hop on an average N by 2. If N is very large, this is going to be a slow process. So query discovery should be a fast process, not a slow process. So we can improvise on this. So the improvisation uses finger tables; we call them finger tables because they look like shortcuts being done; it is not a circle. In general, we can call them the routing table or neighbor tables. So each node will keep information about more nodes, but which nodes are a question that remains.

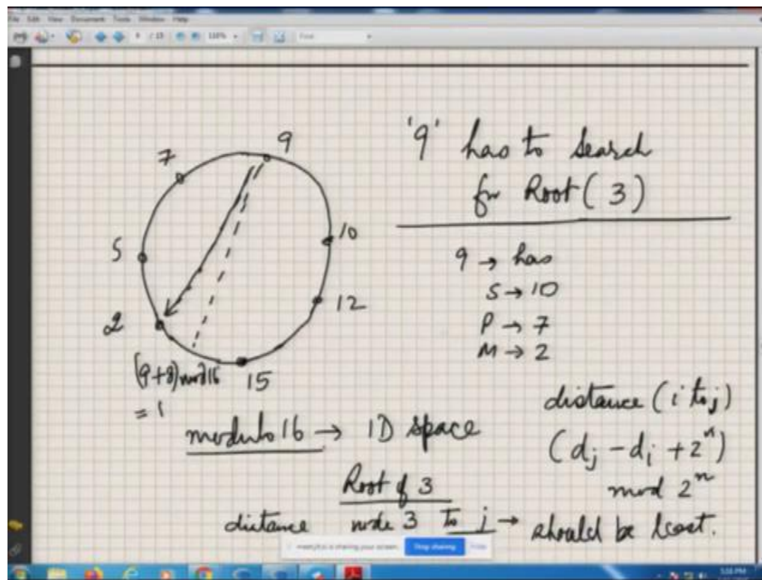
(Refer Slide Time: 26:48)




So should each node keep information about everybody? So if there are 1000 node, I have to make a thousand entries, every node has to maintain, that is also is not logical. It is like a full connected mesh, which can happen with low nodes, but not for a large number of them. So, maybe what we can do is create a successor; I can maintain a predecessor. I can also take a node, which is the root of the halfway across the entry. So, in my current node IDs, whatever it is, I go halfway across, find out what the corresponding root node is my middle entry.

So, I will find out if a query has to go between S and this. So it is closest to this one or closest to this one I can hand over correspondingly. Similarly, I can do hand over to P or this depending on it is closer to which of the entries. So, that could be one way. I can further actually expand on this.

(Refer Slide Time: 27:51)




So this is an example given that I have given node numbers 2, 5, 7, 9, 10, 12, and 15 in this chord, and I am looking for 9's entry. So 9 will have a successor as 10, predecessor as 7, but what would be the middle entry? There are 16 possibilities here; it is a 4 bit node ID space; I have taken that for simplicity. So 16 by 2 is 8, so 9 + 8 is 17, 17 modulo 16, 1 and root node for 1 is 2, so 2 will be the middle entry.

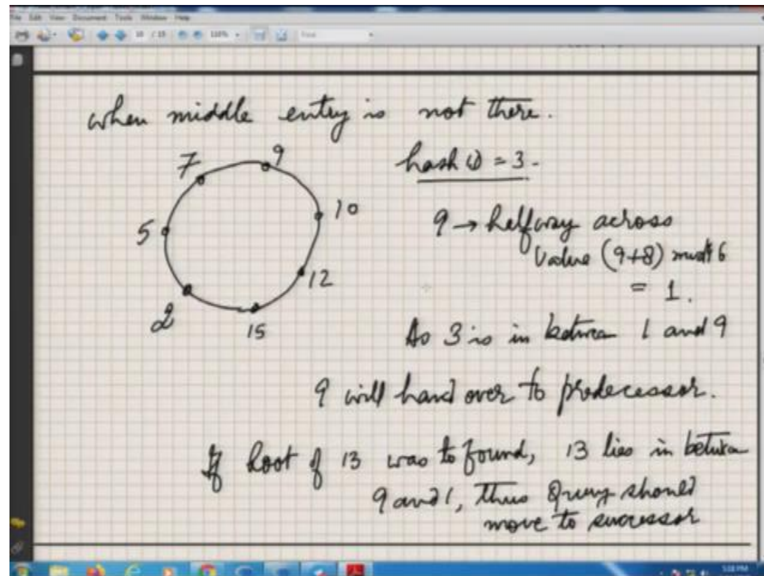
So, when I want to, I can also now have a distance from i to j, node i to node j can be $(d_i-d_j+2^n)$mod $2^n$ if a node is here between 12 and 15, that node to 15, the gap that will be my entry. If entry is between 14 and between 14 and 2, I want to find out space. When I go from 14 to 2, I will be the distance in a clockwise direction in a cyclic fashion. So this function gives exactly that value. So, even in fact from 14, 15 to 2, what is the distance? It should be actually after 15 you will get 0, 1 and 2, it should be 3.

So, I can say 2 - 15 I cannot have it will become a negative value, so I should add 16. So, 2 - 15 is 13, 13 - 16 will be 3, and 3 modulo $2^N$ is 3, so 3 will be the distance from 15 to 2, not from 2 to 15. So, 2 to 15 distance if you want that will be 13, 15 - 2. So, I to J is a directed graph kind of thing, so I to J distance is not equal to J to I distance so that one should note.

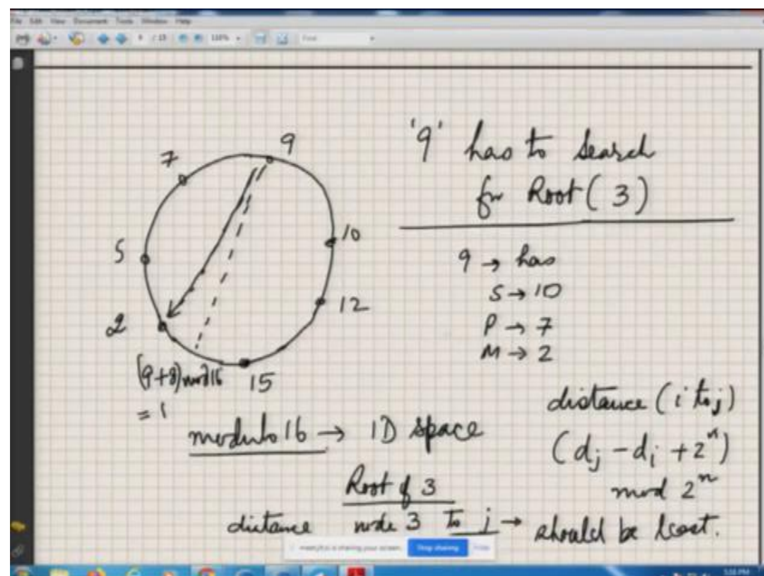
We can say 3 to J; the node J will be the root node if I am searching for a root node, hash ID3. So, 3 to J distance should be least, such that J will be the root node. So that is, for example, what I would like to search for. So, in that case, I will find out 3 will be in between

which value will be between 7 and 2. So I will just send it over the query to 2, and 2 will find out who is the next guy, who is the successor of 5, it will hand it over to 5. 5 will know that it is between my predecessor, which is 2 and myself which is 5. So 5 should be a root node; that is how it is going to happen.

(Refer Slide Time: 30:31)




(Refer Slide Time: 30:37)

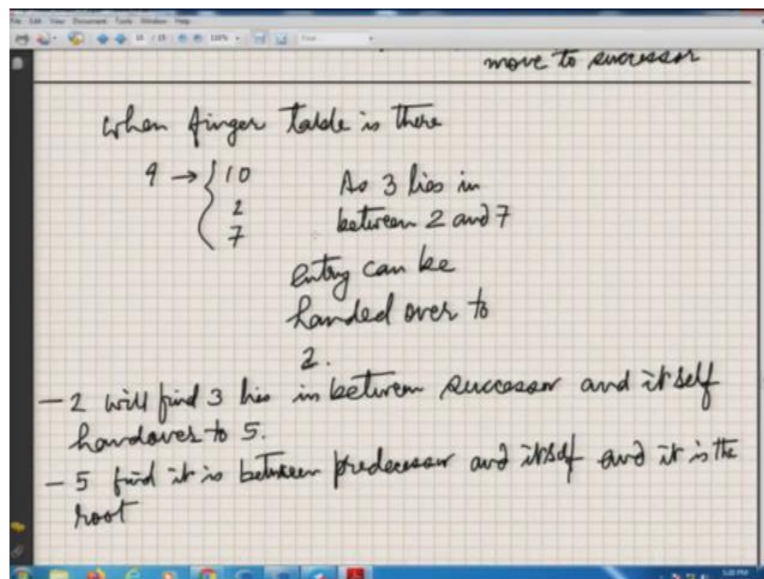



When hashing, when middle entries are not maintained, this extra entry, which is part of the finger table for making things fast, you can do it both ways. You can do it in a clockwise search as well as anti-clockwise. Ideally, if I implement an algorithm, I need not always go clockwise because I have both predecessors and successors. So for example, I want to search

for 3. So I will just say whether 3 lies on the left half of the right half. So diametrically opposite number for 9 is going to be 1.

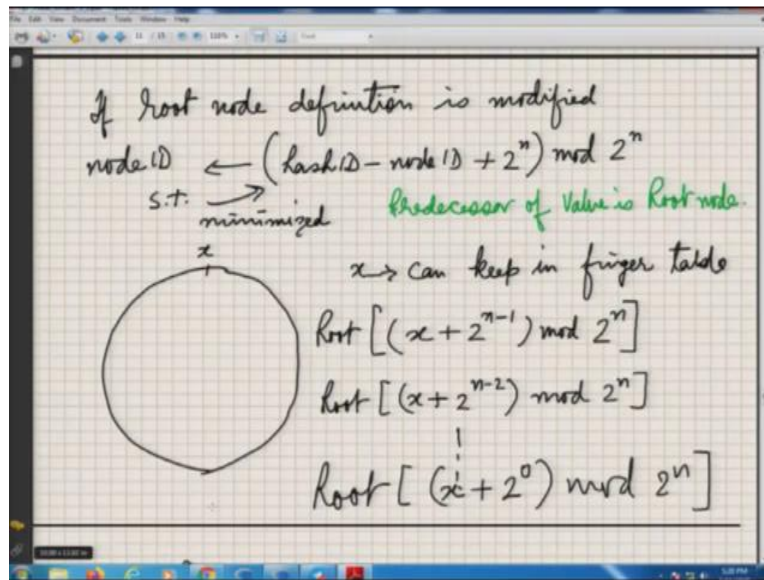
So 3 is more than 1, so it lies between on the left side, so it is good to send it via predecessor. Probably that will require a lesser number of hops assuming a uniform distribution of nodes. Otherwise, if it is between 9 and 1, then, in that case, give it to a successor. So this will, this actual entry for search will going from 7 to 5, and 5 will then figure out that it is a root node, and it will stop there. And if the root of 13 you have to find out, it will go through a successor root. If I am not using finger tables that is the way it will be happening. So I can do bidirectional also.

(Refer Slide Time: 31:42)




If the finger table is there, you will give it to between it, you can hand it over to 2 and 2 will then further give it to 5, and 5 will say that I am the root node.

(Refer Slide Time: 31:52)




So I can now expand on this finger table, so I cannot keep only one entry; I can keep many of them. So normally, it will be like root node definition can be modified. So I am modifying it slightly. I am not going from a distance from hash ID to node ID, and I am looking at a distance from node ID to hash ID. See, I have used hash ID first, hash ID - node ID here. So it is a distance from the node ID to hash ID, which is going to be minimum that node will be the root node. So I have now inverted the definition.

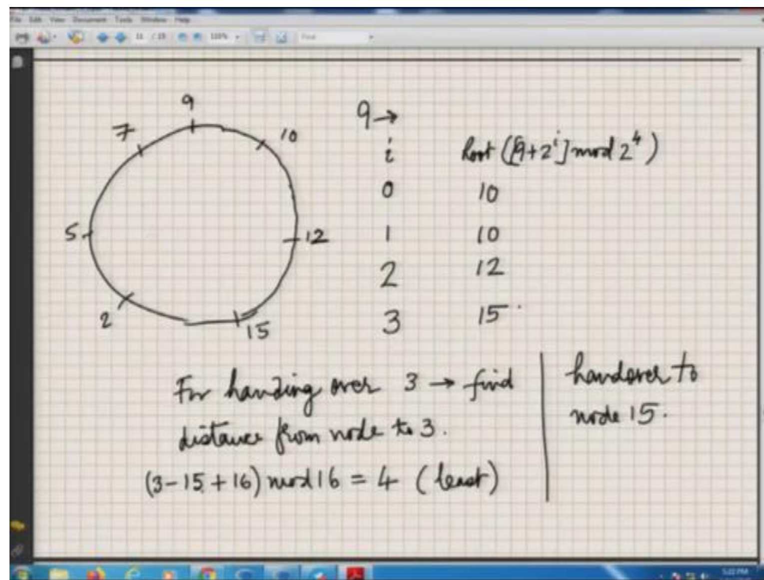
So, this needs to be minimized; find out the node for that. There is a reason for doing it because it makes routing much, much simpler and even testing to figure out whether I am the root node or not also simpler. Now, x is the current node at which I am, and now I want to find out the finger table entries. So I will now search for the halfway across the entry. So x and there is a diametrically opposite entry. So this will be $x + 2^{n-1}$.

So if it is $2^n$ is 16, it will be 8 in that case, modulo $2^n$, find out the root of that. The node that is closest to that but which is smaller is what essentially this distance means. So, that will be the first entry. Then I can do a quarter, $2^{n-2}$ and so on till I get $x + 2^0$ mod $2^n$. So, I can keep n entries in my routing table, where n is the number of bits, so there will be four routing table entries in this case.

But if n is 256, should I make 256 entries. I need not. I can make the lower side entries; I can make where it goes from this $2^i$. So it goes from 0, 1, 2, 3. And then on the higher side, I can

go from 255 to 254 or something and middle values I need not keep so I can actually keep a shorter table so I can do a jump around and still converges much faster than using only a pure circle.

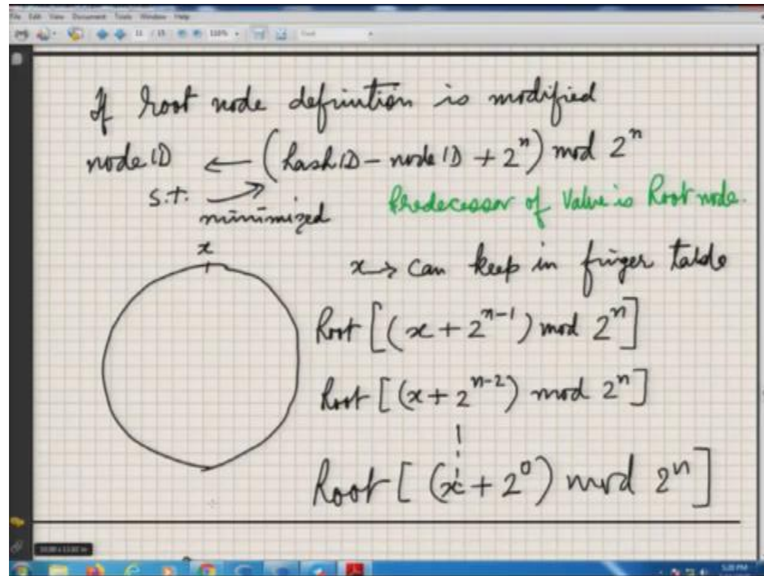
(Refer Slide Time: 34:01)




So, in this case, for example, look at node 9. So I am taking the same DHT, so it consists of 2, 5, 7, 9, 10, 12 and 15, and I am looking at node 9. So I take a value of I, so if it is 0, so these are first finger table entry, $9 + 2^i$, $2^0$ is 1, 9 + 1 is 10, modulo 16 it remains 10. For 10 the root node, the root node will be whatever is this value, this - node ID + $2^4$ total sums and then the module of $2^4$, that is what I should be using. So for this, the root node will be 10 only.

Then $2^1$, so 9 + 2 is 11, 11 modulo 16 is 11, for 11, the root node is 10 because 10 is the closest one, lower and closest as per the definition, which I have given here, the same definition.

(Refer Slide Time: 34:59)




When i is equal to 2, it becomes 9 + 4, 13 and 13 is more than 12. So 12 should be a root node here, which I have put in here, and then 9 + 8 is 17 modulo 16 will be 1. And if I look at the distance, 15 will be the closest one because it is lying between 15 and 2; this is the root node for 3. Now founding for ending over 3, I will find the distance from any one of these entries. I am looking at a node hash ID of 3 who is the root node of that.

I will just compare each of these entries with which I will hand it over to it. So distance from a node to 3. Remember this is as per the definition given here, which I have made a change from the first case, so this will become 3 - 15 + 16 modulo 16. This will become 4. So this is 3 - 15 will be - 12 + 16 is 4, 4 modulo 16 is 4. So 4 this 15 will give the least distance, so I will hand over the packet to the 15. And you can see for node 3 it is a 2, which will be the root node, not 15. So 15 has to do the same exercise again.

(Refer Slide Time: 36:22)

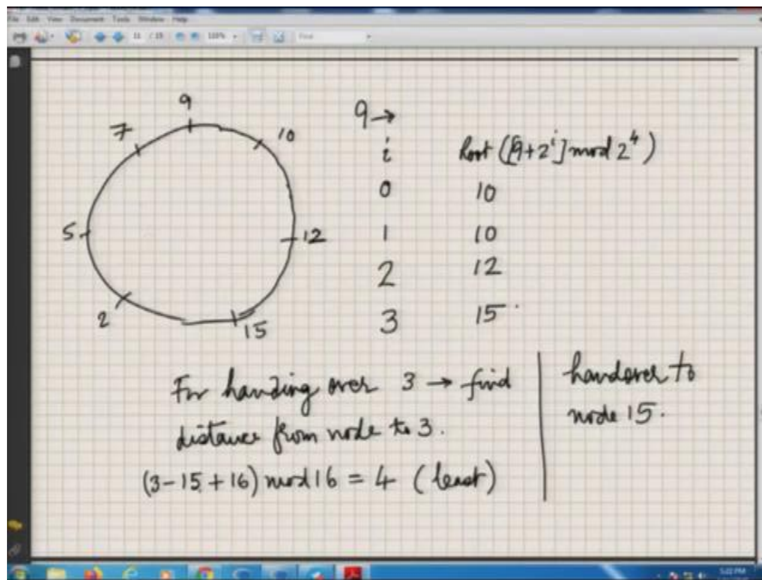
So node 15 will now also maintain its routing table. So, for 0, it will be 15 + 1, 16, and modulo 16 is 0. For 0, if you go back and see the figure for 0 and 1, the node route will be 15, so I will put 15. When I is 1, 15 + 2 is 17, which will give after modulo operation as 1, node ID 15 will be the root node. When i is equal to 2, it is 15 + 4, 19, modulo 16 it will be 3, the root node will be 2 and 3 will be 7. When we compare the node hash ID 3, we will figure out 2 will be the closest one muscle will hand it over to 2.

The second hop, 2 will now again do the same exercise; it will have its routing table. So 2 + 1 is 3 for routing, the root node for that is going to be 2. 2 + 2 is 4, the root node is 2, 2 + 4 is 6, the root node is 5, 2 + 8 is 10, the root node is going to be 10. These will be root nodes, and node 2 will figure out this remains the closest node to hash ID 3. So node 2 will declare that I am the root node, and it will now search for the corresponding query in the database.

(Refer Slide Time: 37:52)

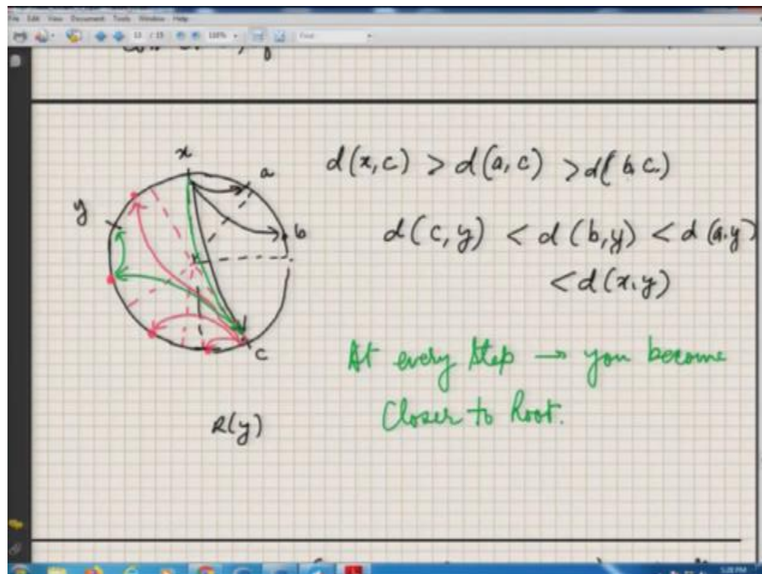
(Refer Slide Time: 38:11)

(Refer Slide Time: 38:15)




If we are going to use 256-bit node IDs, these are a pretty large number; I can have at best 256 entries in the finger table at each node. I am telling the middle entries need not be used. In fact, for some of the entries, you will get the same root node. You can see that I am getting for 2 entries 15. I am getting here again 2 entries 10 so; I can combine 0 to 1. It is 10 only in one single entity. I need not maintain multiples of them. So, I can compress the finger table.

(Refer Slide Time: 38:24)

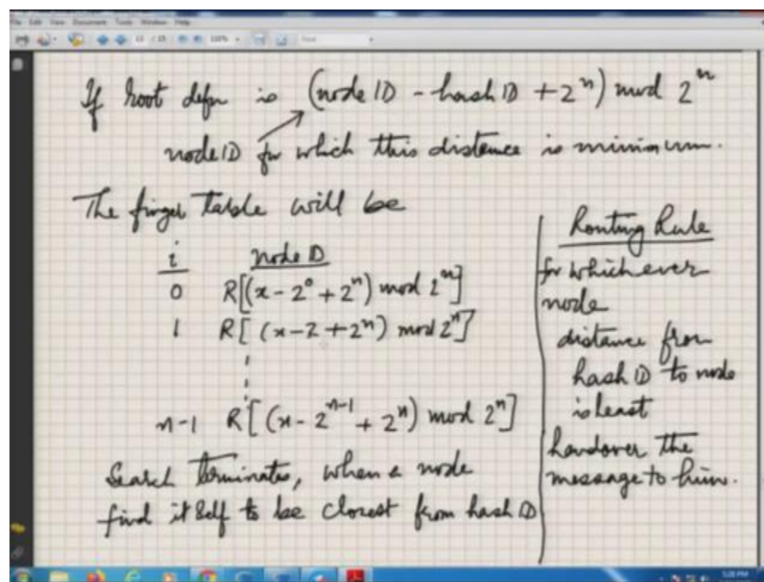



And of course, this is a property that will always be maintained. So, here, for example, the root of y has to be figured out; the root of y is this particular red node. So, I am starting from x, x will now we have a property we have d, x, a, b, c. So, d x, c from x to c the distance is

always going to be greater than d a, c from a to c this property is always going to maintain. This distance can never be larger than x to c distance. So the same is going to be true for d b, c and d a, c if I compare, and if I compare d, c, y similarly, and between d b, y, so every time you are going to the closest node so that x will have only these three entries.

So, d a, y, d b, y will be larger, but d c, y will be the least, so I will hand it over to d c, y. And every step, you always tend to go to the closer and closer to the hash IDs root hash ID essentially, and when you get the closest node, that will become the root node.

(Refer Slide Time: 39:32)




We can change the root node again, but my routing table will change. So somebody says, I am going to now the root node will be the guy next from the hash ID, so the distance we will minimize is node ID - hash ID + $2^n$ modulo of $2^n$.

(Refer Slide Time: 39:55)

Now, this is different than the definition which I have used earlier. Here node ID and hash ID have been swapped in the latter case. So I will not be using plus here. I was making finger tables on the successor side.

(Refer Slide Time: 40:05)

You can see here I am always making this halfway across on this side, quarter on this side, one eighth on this side and so on. Now I will start making a routing table on this side in the opposite direction because I have changed my distance metric.

(Refer Slide Time: 40:23)

Now, this is how it will look like. For 0, I will be doing $(x - 2^0 + 2^n) \bmod 2^n$ and so on, every time it is a minus sign this time. Again, these are n entries which I need to maintain and search will terminate when a node finds it is closest from the hash ID, now from the hash ID, it is not to, in the earlier case it was to the hash ID.

(Refer Slide Time: 40:48)

So take a case here when hash ID is 12, and the search will start from node 7. Finger table for node 7 will be 7 - 1, and as per the definition, it will be node ID - hash ID; that is what we are using, you can see here.

(Refer Slide Time: 41:11)

So based on this, the root node will be 7.

(Refer Slide Time: 41:15)

For i is equal to 1, 7 - 2, its root node will be 5, 7 - 4 again 5 and the last entry in the finger table will show 15. So when I am searching for 12, I will find out 15 will be the closest node. So 7 will hand over the query to 15.

(Refer Slide Time: 41:32)

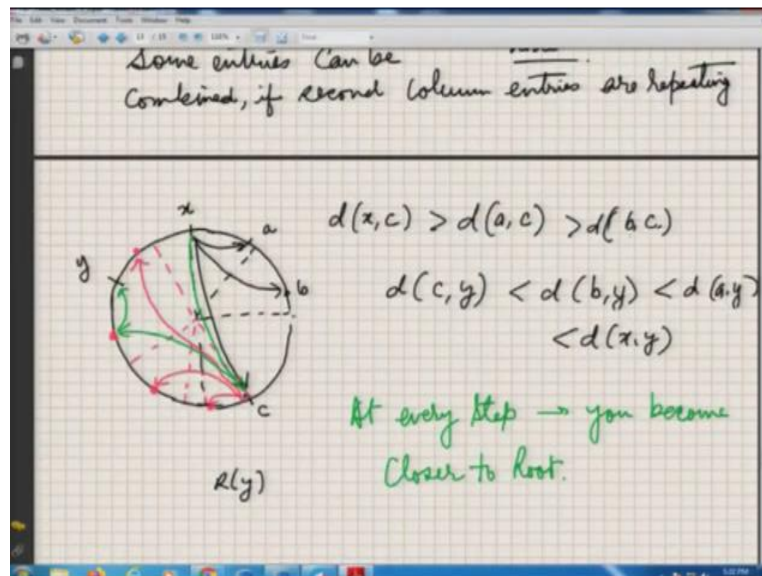
15 will again now build a finger table. For i is equal to 0, it will be 15, same logic 13, 11, then 7, and 13 you will find is closest from node 12. So it is from again hash ID to node ID. That is why I have written 12 is the hash ID, and it will hand over the message to 13. Again 13 will build up the finger table, and 13 will be the closest from node 12. So, the search will terminate at 13 and 13 will be the root node.

(Refer Slide Time: 42:17)

So 13 is so 12 lies somewhere here, so the next guy, so, this is the way it is working. So, it is very systematic; you always use one single distance metric. Based on that, you build your

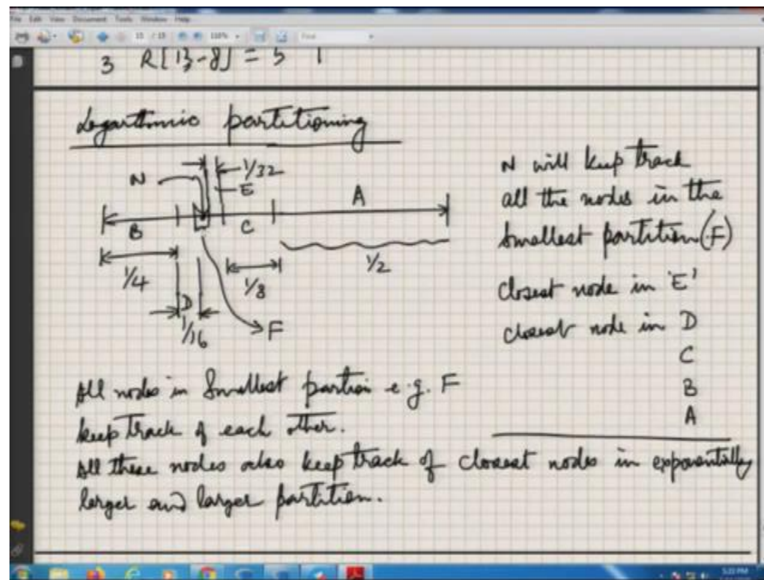
finger table and your routing strategy, and for the root node criterion, all 3 are using a similar metric.

(Refer Slide Time: 42:44)




Now, also one thing which you will do is that, notice that what the way I am doing partitioning let me go back to the figure. I am going halfway across one eighth, one fourth, one eighth, one-sixteenth. So, $1/2^i$ that is what I am using. So, I am logarithmically increasing, reducing the size of the partition, and I am trying to find the node. It is not, of course, the closest node here, but in general, when you build a DHT test to the closest node in a partition. So let us see what is this logarithmic partitioning is conceptual.

(Refer Slide Time: 43:14)

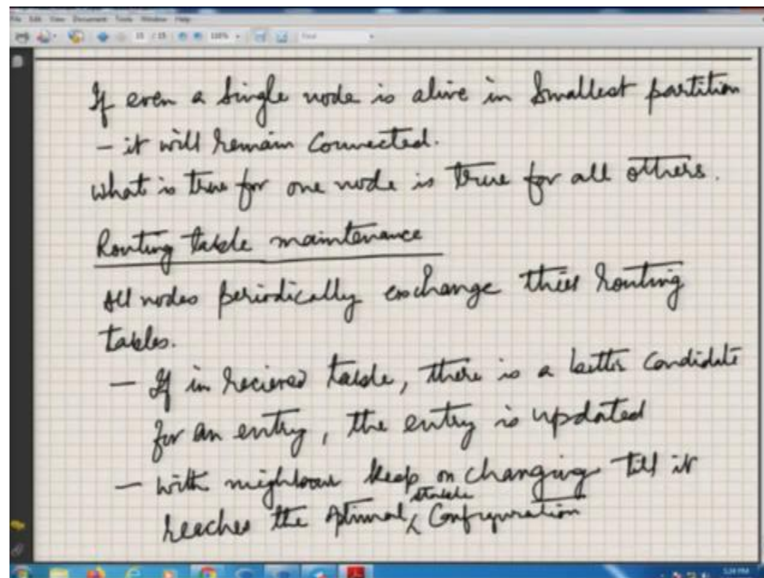



So, this is what we call logarithmic partition. Look at this node N, so you want to build up a routing table for this. So, the whole node ID space will be partitioned into two halves A and the remaining of A. So, the node which is in A closest to this node N will be maintained by N only one entry from this half. The remaining part may further divide into two halves; B is B; node N is not in part B. So, the closest note from B is also going to be maintained by N.

We will further divide it; this will now be 1 by 8, so C, the closest note from C, will also be maintained by 8. Then whatever the remaining part, I will further divide it into two parts, one comes 1 by 16 part, which is D. So, the closest node from this will also be again maintained by this node N. Further division 1 by 32 is again closest node will be maintained. All the nodes will be maintained by node N within its partition, which is a partition F.

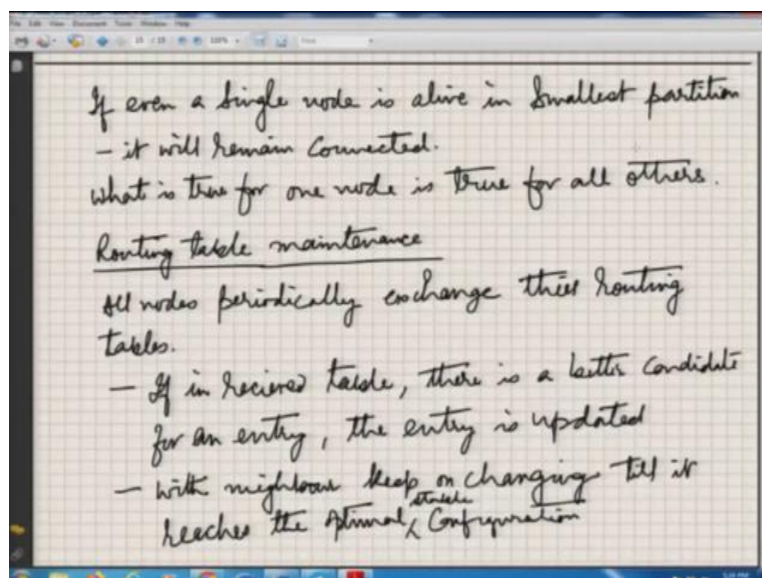
So, all nodes in the smallest partition, for example, F, will keep track of each other, and all these nodes also keep track of the closest nodes in exponentially larger and larger partitions. We are also doing almost something similar in the chord. So, the closest node in E is how the routing table will look like. So all nodes in the smallest partition, the closest node in E D C B A partitions.

(Refer Slide Time: 44:41)




Even if a single node is alive in the smallest partition, it will remain connected because it is connected to everybody else. Each one of them is connected to one node in every other partition. So this maintains a guarantee. And what is true for a node that is the smallest partition is true for every other node because everybody behaves in the same way. So, that is proof that logarithmic partitioning always maintains the connectedness property inside the DHT. That is the reason why it is used there.

(Refer Slide Time: 45:18)



Routing maintenance, as I mentioned earlier, that all nodes should periodically exchange their routing tables. You should, whatever nodes you are aware of, send your information to all of them even if they do not exist in your routing table. So, every node is getting a lot of updates from other nodes about the new nodes. So some of them may not be there in the routing table. So, you will now search for these nodes. For example, in partition A, if you are somebody's closest, that guy sends all its neighbor table to you next time.

You find some other node is also in that same partition, but it is closer than compared to what was earlier value, you will replace it. So this will be happening all the time for all partitions, and you will be maintaining ultimately optimal and stable routing table configuration. So this is the way that you maintain a DHT thing. We will further look into the next lecture on how to study more about these DHT routing tables. And we will generalize this concept in a more elaborate fashion.