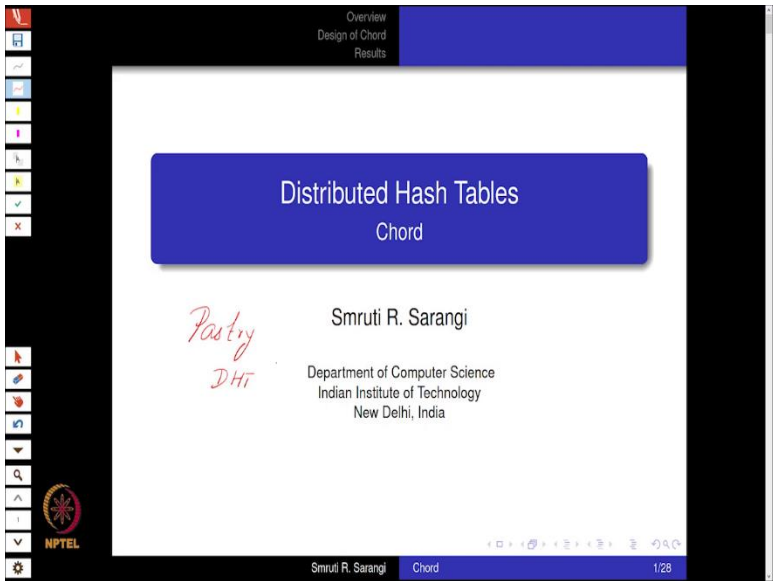
Advanced Distributed Systems
Professor Smruti R. Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology Delhi
Lecture 5
Chord: A scalable DHT

(Refer Slide Time: 00:17)



Distributed Hash Tables
Chord

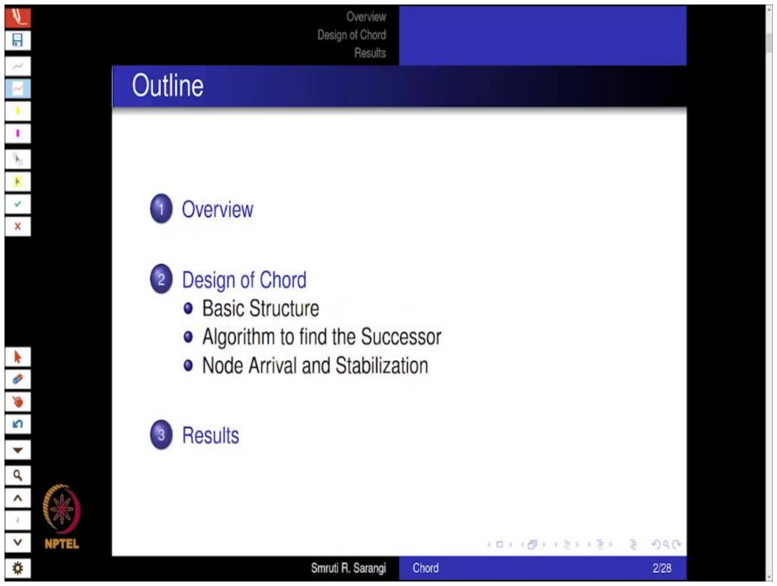
Smruti R. Sarangi

Department of Computer Science
Indian Institute of Technology
New Delhi, India

Pastry
DHT

So, we will discuss the second distributed hash table algorithm called chord. So, a prerequisite of this lecture is the Pastry DHT, the Pastry distributed hash table. So, this is a prerequisite of this current lecture. So, first we will discuss Pastry. So, that has been done in a previous lecture. So, now we will discuss chord.

(Refer Slide Time: 00:44)



Overview
Design of Chord
Results

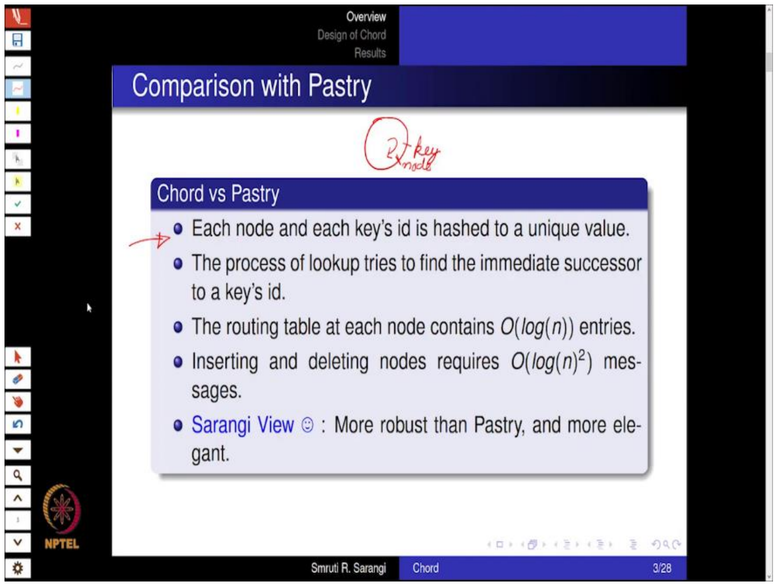
Outline

1. Overview
2. Design of Chord
   - Basic Structure
   - Algorithm to find the Successor
   - Node Arrival and Stabilization
3. Results

Smruti R. Sarangi | Chord | 2/28

So, the lecture will go like this overview, design of chord, the basic structure, algorithm to find the successor, node arrival and stabilization and finally the results.

(Refer Slide Time: 00:57)



Overview
Design of Chord
Results

Comparison with Pastry

Chord vs Pastry

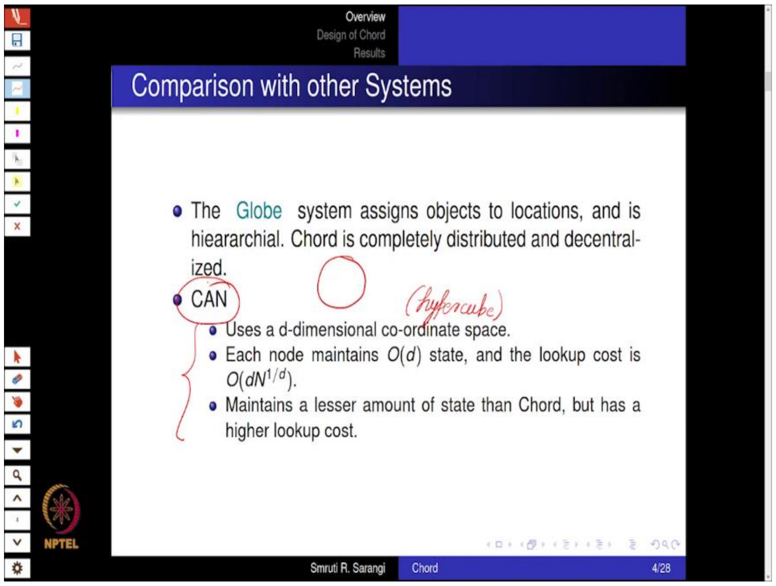
- Each node and each key's id is hashed to a unique value.
- The process of lookup tries to find the immediate successor to a key's id.
- The routing table at each node contains $O(log(n))$ entries.
- Inserting and deleting nodes requires $O(log(n)^2)$ messages.
- Sarangi View ☺ : More robust than Pastry, and more elegant.

Smruti R. Sarangi | Chord | 3/28

So, chord versus Pastry. So, the key idea is the same that each node and each key's id is hashed to a unique value. So, this is same, but the process of lookup is different. So, recall that in Pastry, the process of lookup tried to find the node that was closest to the key. But in this case, we do not do that, what we do is instead that in the circle, we try to find the node that is the immediate successor of the key.

So, we take the key's id and we try to find the node that is the immediate successor, when we are traversing the circle clockwise. The routing table at each node will contain roughly on an average O(log(n)). And inserting and deleting nodes will require O($\log(n)^2$) messages. It is just my view that this is more robust and Pastry and far more elegant as a solution. But that is just my view, and you need not subscribe to it. So, you make your own views by the time we reach the end of this lecture.

(Refer Slide Time: 02:08)



Comparison with other Systems

- The Globe system assigns objects to locations, and is hieararchial. Chord is completely distributed and decentralized.
- CAN
  - Uses a d-dimensional co-ordinate space.
  - Each node maintains $O(d)$ state, and the lookup cost is $O(dN^{1/d})$.
  - Maintains a lesser amount of state than Chord, but has a higher lookup cost.

(hypercube)

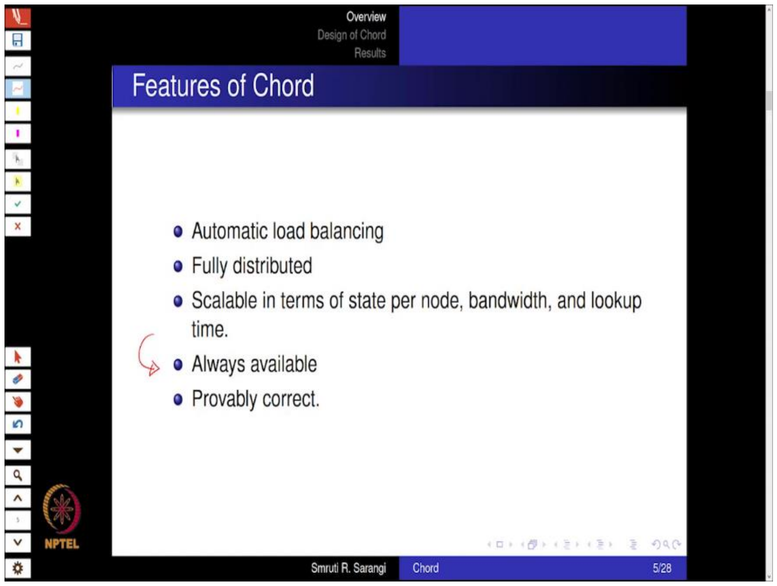
So, comparison a few of the other systems. So, by the time that chord actually came, there were many more DHTs in the research market. So, there was a globe system, which used to assign objects to locations, it was hierarchical. As compared to that chord is completely distributed and decentralized. So, of course, chord also similar to Pastry uses a ring-based overlay.

If you do not want to use it, you can use CAN. So, CAN users, or d dimensional coordinate space. So, instead of a circle, it actually uses a d dimensional coordinate space that is basically a hypercube. So, a square, a cube and hypercube. So, it is a d dimensional coordinate space. You do not know what is a hypercube?

This is the right time for you to take a look, just Wikipedia for it. So, each node maintains O(d) state, which is basically the d neighbor in the d dimensional coordinate space. And the lookup causes O(d$N^{1/d}$). So, it does maintain a lesser amount of space than chord but has a higher lookup network.

So, CAN is one of the immediate competitors of chord. And the interesting part is that as opposed to a circular overlay, which Pastry also uses, chord also uses CAN has more like this d dimensional hyperspace. So, this is a worthy competitor, but we will still find the elegance of court to be something that will really endear it to us.

(Refer Slide Time: 03:51)



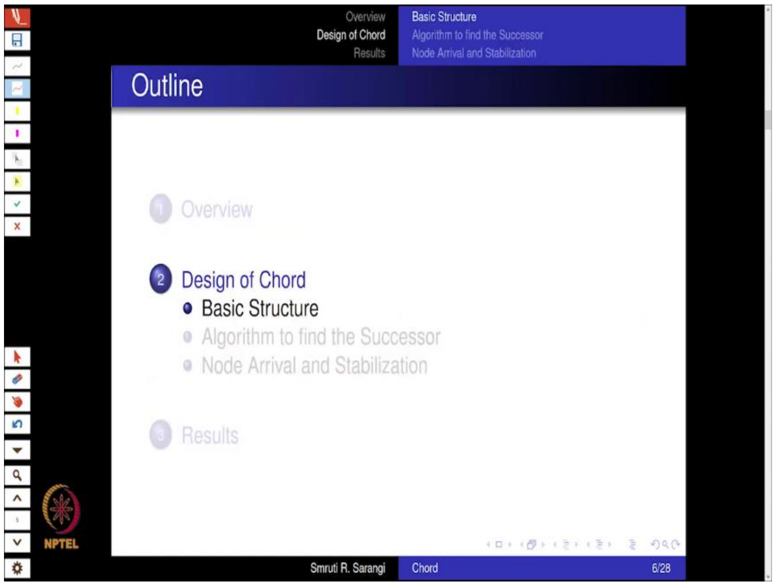
Features of Chord

- Automatic load balancing
- Fully distributed
- Scalable in terms of state per node, bandwidth, and lookup time.
- Always available
- Provably correct.

So, what are the features of chord? Well, you have automatic load balancing. It is fully distributed. In terms of state per node, bandwidth and lookup time it is fully scalable. Furthermore, there is greater availability, it is always available in the sense that there is a built in amount of redundancy into the chord protocol.

So, that is the reason that availability is high and second it is provably correct but the same was with Pastry also, that these are simple protocols is easy to prove that they actually work. I mean, all corner cases are taken care of. And they work in a robust, provable fashion.

(Refer Slide Time: 04:32)



Overview | Basic Structure
Design of Chord | Algorithm to find the Successor
Results | Node Arrival and Stabilization

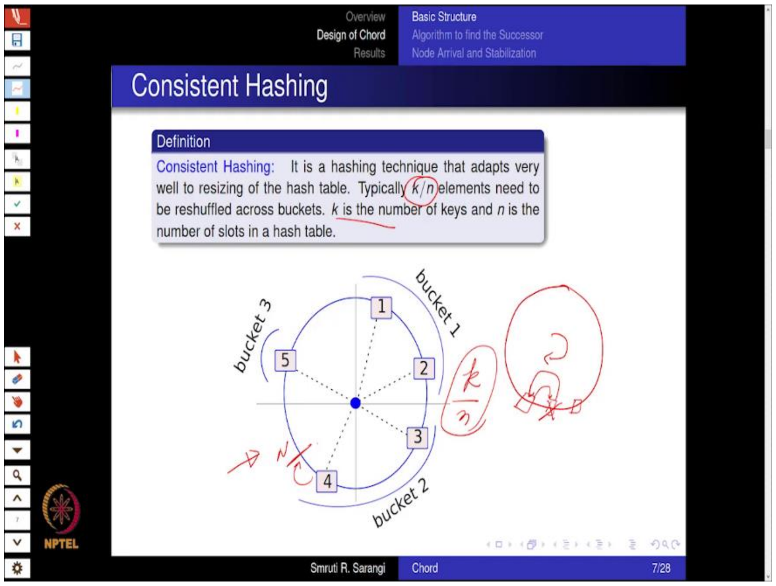
Outline

1. Overview
2. Design of Chord
   - Basic Structure
   - Algorithm to find the Successor
   - Node Arrival and Stabilization
3. Results

Smruti R. Sarangi | Chord | 6/28

Design of chord, we will take a look at the basic structure.

(Refer Slide Time: 04:37)



Overview | Basic Structure
Design of Chord | Algorithm to find the Successor
Results | Node Arrival and Stabilization

Consistent Hashing

Definition

Consistent Hashing: It is a hashing technique that adapts very well to resizing of the hash table. Typically k/n elements need to be reshuffled across buckets. k is the number of keys and n is the number of slots in a hash table.

Smruti R. Sarangi | Chord | 7/28

So, the key idea of chord is called consistent hashing. So, it is a hashing technique that adapts very well to actually resizing the hash table. So, what is the key idea? It is that typically $\frac{k}{n}$ elements need to be reshuffled across buckets. So, what happens is look?

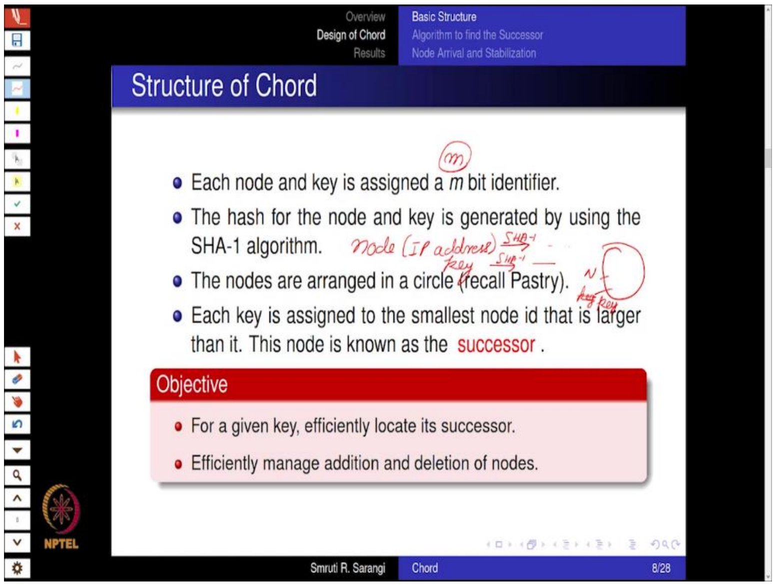
When you have a large space, and you have these nodes that are also called buckets, and you add a node in the middle, then you need to move some keys from here to here. So, that needs to be done. So, in Pastry, it was a two way move, but in chord, it is a one way move. So, this is again one more advantage. That is because we are storing the immediate successor immediate clockwise successor that is.

So, on an average, the number of nodes that need to be moved. And the same is for an add in for a delete as well, if later on, we delete this, then some nodes have to be moved from here to here back. So, the average movement is limited to $\frac{k}{n}$ elements across the buckets with k is the number of keys and n is the number of slots in a hash table.

Number of slots in a hash table basically refers to the number of nodes in this case. So, $\frac{k}{n}$ is the typical movement, which is anyway, on expected lines, it is not unexpected. And of what we are basically seen. Over here, is that we will see how consistent hashing plays a role in chord. So, here, of course, in this figure, we are using the term bucket, so we will get to it in some time.

But essentially, what this figure over here is showing is that on big circular ring, different keys are hashed to different positions. And then we group them into buckets. This basically means that a set of keys in one bucket are assigned to the same physical node. So, I am not showing the node over here. But if there is a node, then there is entire bucket gets assigned over here, because we assign them to the immediate clockwise successor.

(Refer Slide Time: 06:59)


Overview
Design of Chord
Results
Basic Structure
Algorithm to find the Successor
Node Arrival and Stabilization

Structure of Chord

- Each node and key is assigned a $\overset{\boxed{m}}{m}$ bit identifier.
- The hash for the node and key is generated by using the SHA-1 algorithm. node (IP address) $\xrightarrow{SHA-1}$ key $\xrightarrow{SHA-1}$
- The nodes are arranged in a circle (recall Pastry).
- Each key is assigned to the smallest node id that is larger than it. This node is known as the successor .

Objective

- For a given key, efficiently locate its successor.
- Efficiently manage addition and deletion of nodes.

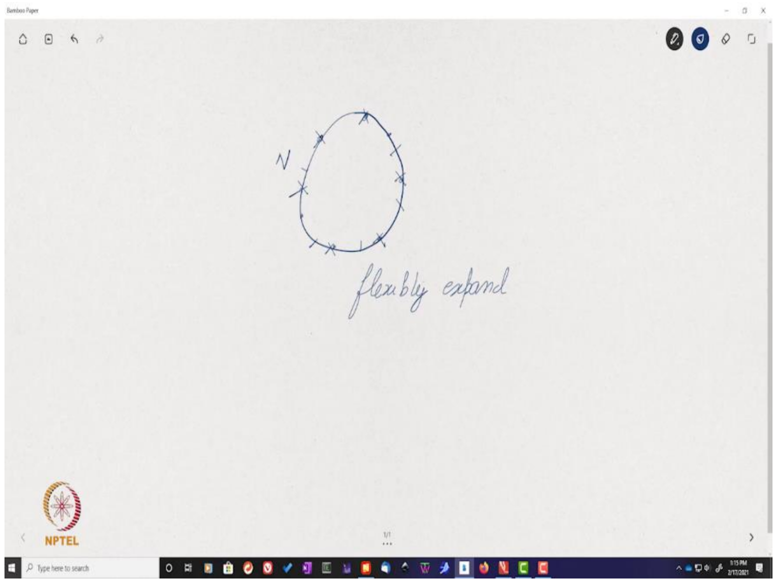
So, the structure of chord is like this slightly different from Pastry. So, each node and a key is assigned a m bit identifier. So, it is m bits, will discuss what is the value of m. So, the hash for the node and the key is generated using the SHA 1 algorithm. So, it is a very simple hashing algorithm that takes the value of a node, the value of a node basically is essentially the IP address of the node.

So, given the node, we take its IP address. We pass it through a hashing algorithm. And this gives us the hash. And similarly, we take the key, we pass it through a hashing algorithm and that gives us the hash. The nodes are arranged in a circle, similar to Pastry. The major difference with respect to Pastry is, that each key is assigned to the smallest node id that is larger than it or call it a clockwise successor.

So, that is the most important thing that each key is assigned to the node that is just larger than it. Basically, it is clockwise successor. So, the objective is that for a given key efficiently locate its successor's efficiently located successor on this ring, and efficiently manage the addition and deletion of nodes. So, those are two aims to manage to ensure that if you want flexibility, can you provide it.

And in this case, flexibility basically means that we can add as many nodes as we want, we can remove as many nodes as we want. So, what is an advantage? I will just quickly show it over here.

(Refer Slide Time: 08:55)

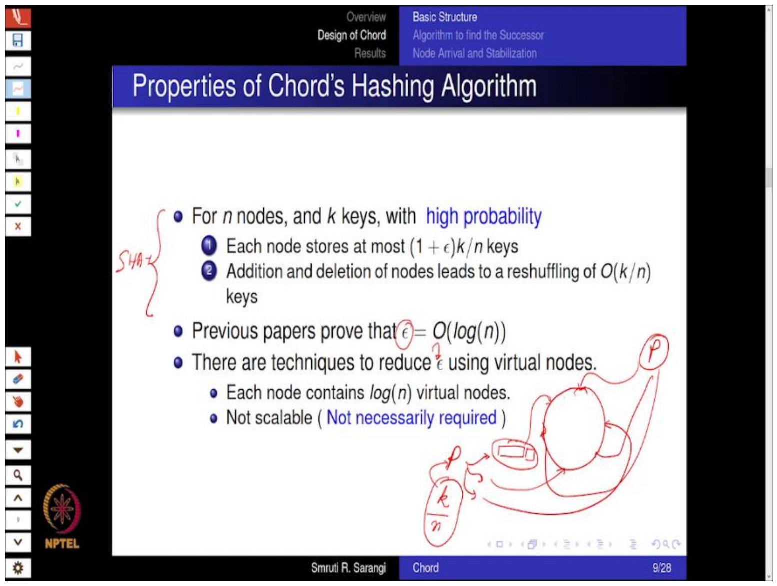


So, on advantage of this scheme is that, let us say we have the circular ring, and we have some nodes. And let us say that, it is the time of a festival. Let us say it is Christmas in the US, Diwali in India that is when most of the shopping happens. And your favorite shopping site uses a DHT. Then what you can essentially do is that you can add more servers.

So, the new servers will basically be positions along this ring and they can take away a large part of the load of the existing nodes. So, the network will rebalance. And as we have seen, roughly $\frac{k}{n}$ nodes need to be transferred between for adding every new server. So, this allows us this gives us a method to kind of flexibly expand and grow the network.

So, let us say that when there is a festival you expect a lot of people to be searching things, buying and selling things. So, all of those things can be incorporated by adding extra servers in the network to share the load. And then when the festival is over many of these new nodes that were added, they can get deleted. So, this is what gives DHTs the flexibility.

(Refer Slide Time: 10:34)



Overview | Basic Structure
Design of Chord | Algorithm to find the Successor
Results | Node Arrival and Stabilization

Properties of Chord's Hashing Algorithm

- For n nodes, and k keys, with high probability
  1. Each node stores at most $(1+\epsilon)k/n$ keys
  2. Addition and deletion of nodes leads to a reshuffling of $O(k/n)$ keys
- Previous papers prove that $\epsilon = O(log(n))$
- There are techniques to reduce ϵ using virtual nodes.
  - Each node contains $log(n)$ virtual nodes.
  - Not scalable (Not necessarily required)

Smruti R. Sarangi | Chord | 9/28

So, now let us see, what are the properties of cords hashing algorithm? Because pretty much the uniformity of the hashing algorithm to a large extent plays a rather important role in the design and operation of chord. So, for n nodes and k keys as we have said with high probability. Each node stores are the most $(1 + \epsilon)^{k/n}$ keys.

Addition and deletion of nodes leads to reshuffling of order of $\frac{k}{n}$ keys. This is a property of the hashing algorithm of chord, which is basically SHA 1. Previous papers have proven that epsilon is limited to O(log(n)). And there are techniques to reduce epsilon, if we are really interested about the uniformity of distributing keys to nodes, using what are called virtual nodes.

So, what we can do is that we can make each physical node contain log n virtual nodes. So, then if each physical node contains log n virtual nodes, so a virtual node by the way, the full-fledged node in so far as the hashing is concerned. So, let us say that we have a consistent we are varying like this. And let us say we have a physical node p, then the physical node p.

If let us say we are creating log n virtual nodes out of it, and then each nodes id will basically be the id the IP address of p with the virtual node number. So, this will hash to let us say one point on the ring, the other virtual node will hash to another point on the ring. This one will hash to this point on the ring.

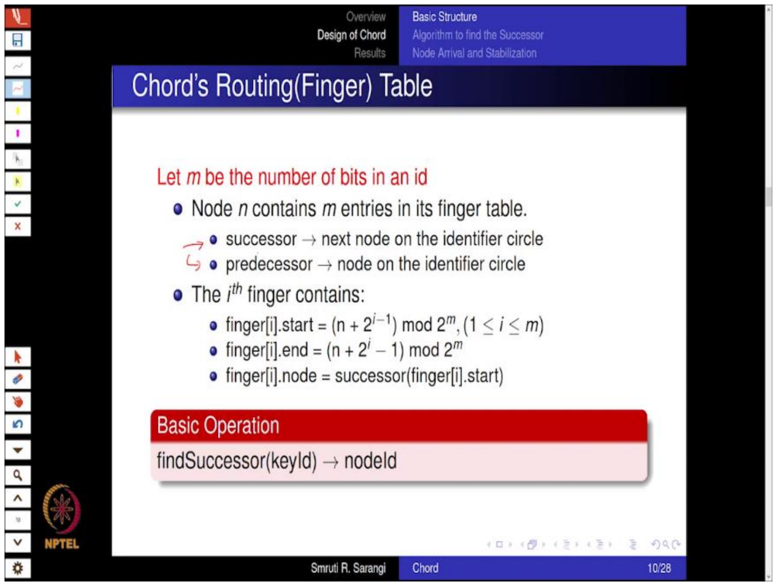
Similarly, if there is another physical node p with let us say, three virtual nodes, one virtual node might hash over here, one might hash over here. So, what this would do is that this would further increase the spread and the randomness. Think about it that way, that this will further the number one, it will increase the number of nodes, so it will, it is not a one-way street.

So, this will increase the load on the network. And it will just in a sense, create a larger network. But also, it will increase the amount of randomness and the spread. So, the benefit will be that the number of keys that are actually stored on the physical node. It is expected to be substantially more uniform and more homogeneous in comparison with other nodes, which is what we wanted.

Of course, there are scalability issues, because we are increasing the number of nodes substantially log n times, so that is an issue. But if we want uniformity, this is a good strategy, because as you can see from the figure, if we have let us had three virtual nodes, they can hash all over the place. If this has three virtual nodes, it can hash all over the place.

So, given the law of large numbers, we can say that look, the probability that the number of keys that will be there with each node is a very, very high likelihood that will be $\frac{k}{n}$. Where k is the total number of keys and n is the number of physical nodes. Let us say, then the number of keys will be $\frac{k}{n}$, or very close to $\frac{k}{n}$ with p if we follow this virtual node-based scheme. And the reason should be clear from this diagram.

(Refer Slide Time: 14:14)



Overview
Design of Chord
Results
Basic Structure
Algorithm to find the Successor
Node Arrival and Stabilization

Chord's Routing(Finger) Table

Let m be the number of bits in an id

- Node n contains m entries in its finger table.
  - successor $\rightarrow$ next node on the identifier circle
  - predecessor $\rightarrow$ node on the identifier circle
- The i^{th} finger contains:
  - finger[i].start = $(n + 2^{i-1}) \bmod 2^m, (1 \leq i \leq m)$
  - finger[i].end = $(n + 2^i - 1) \bmod 2^m$
  - finger[i].node = successor(finger[i].start)

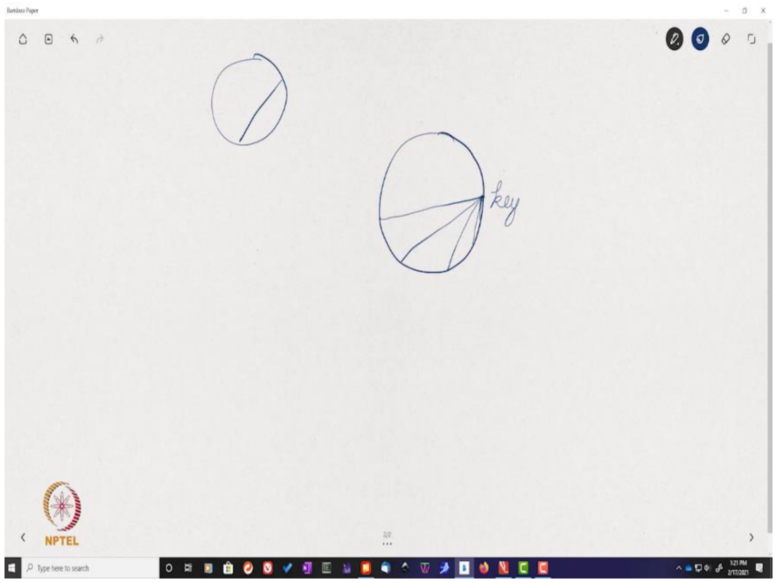
Basic Operation

findSuccessor(keyId) $\rightarrow$ nodeId

So, now let us see, so let m be the number of bits in an id. So, m is the number of bits and let us see both the node id and the key id which is produced by the SHA 1 process. And for a node whether it is a physical node or virtual node as far as the lookup is concerned, it does not matter. It is only when we are looking at the uniform distribution of keys to a node does it actually matter. So, we will now define two terms the successor and the predecessor.

The successor is a next node on the identifier circle. The predecessor is the previous node on the identifier circle. So, here is the fun part that we are going to do, we are going to define the notion of fingers. So, if you see what is actually a chord?

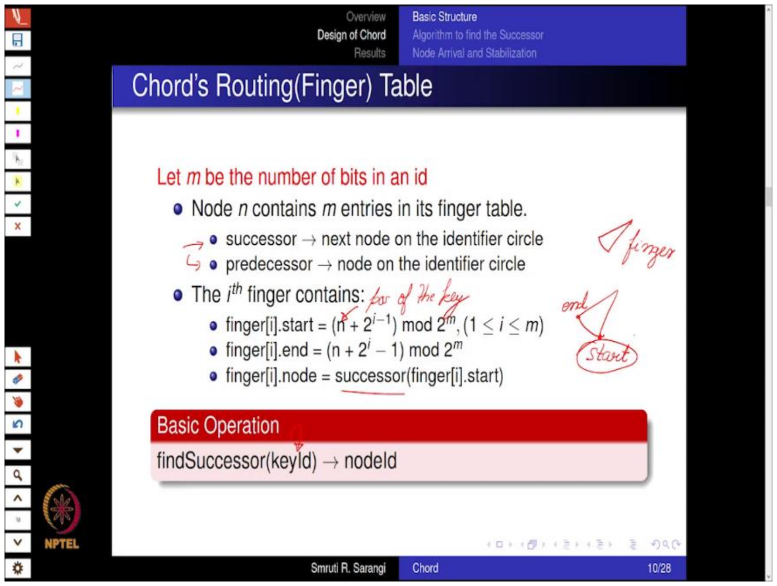
(Refer Slide Time: 15:12)



Say chord basically is that we take a circle, and we cut it like this. So, this is a chord. So, now it is very clear why we call this system chord. So, you take any point, whether it is a node or a key that is immaterial. So, maybe let us start with a key. So, let us for every key will have a position on the ring, then what we can do is that we can, so the biggest chord, of course, will be the diameter.

And then we can start defining smaller chords, where we kind of half tangle every time, something like this. So, as you can see, these are smaller and smaller and smaller portions of the circle, which they gradually get, they kind of gradually get bigger, bigger, bigger, bigger and bigger. So, this is actually the way that we divide the entire space of keys by drawing a set of straight lines, and as you can see their chords. And then the final one is, of course, half the circle, it is kind of the semicircle. So, how do we do it?

(Refer Slide Time: 16:21)



Overview | Design of Chord | Results — Basic Structure | Algorithm to find the Successor | Node Arrival and Stabilization

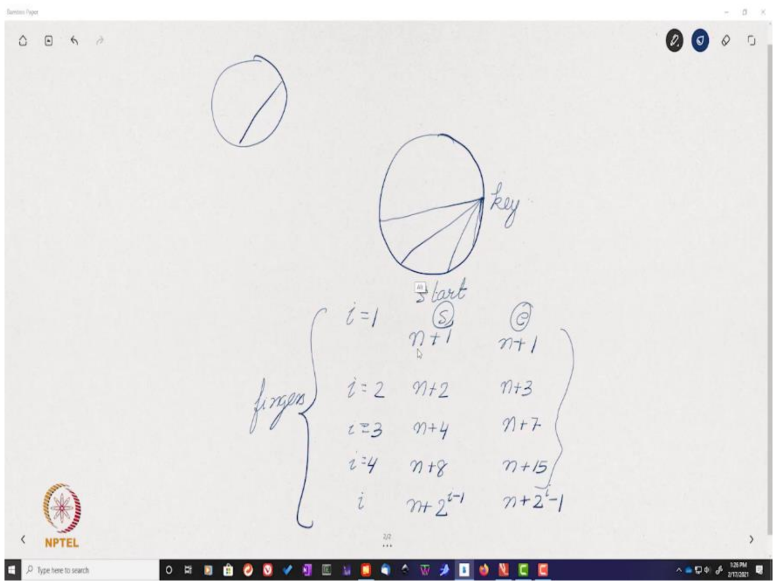
Chord's Routing(Finger) Table

Let m be the number of bits in an id

- Node n contains m entries in its finger table.
  - successor → next node on the identifier circle
  - predecessor → node on the identifier circle
- The i^{th} finger contains: for of the key
  - finger[i].start = $(n + 2^{i-1}) \bmod 2^m, (1 \leq i \leq m)$
  - finger[i].end = $(n + 2^i - 1) \bmod 2^m$
  - finger[i].node = successor(finger[i].start)

Basic Operation

findSuccessor(keyId) → nodeId



	start	end
$i=1$	$n+1$	$n+1$
$i=2$	$n+2$	$n+3$
$i=3$	$n+4$	$n+7$
$i=4$	$n+8$	$n+15$
i	$n+2^{i-1}$	$n+2^i-1$

Well, let us first start by mathematically defining it. So, for each of these, we call it a finger. So, let us say these individual chords. So, let us say something like this, we actually define this because as a finger. Well, it looks more like a pizza slice, but let us calls this a finger. It also looks like a finger a finger nail, if you look at it from a certain angle, but I would have called it a pizza slice.

So, the i'th finger basically contains, so every finger has a start and an end. So, it has a starting point. And it has an ending point. So, the start for the i'th fingered its position is like this, (n + 2^{i-1}) mod 2^m. So, why mod that is because you want to wrap around the circle, you want to wrap around and come to the other side. So, let us and then the finger ends at (n + 2^{i-1}).

So, it is essentially you are increasing the size of the finger by a power of 2, which is something that you saw. So, let me maybe show it to you in a table graphically. So, that will give you an idea of what exactly I am referring to. So, let us say that, let us say you consider (i = 1). So, then what is the start? So, let us say the id over here as we have seen over here, let this be n. So, n is the position.

So, I am slightly changing the definition of n here. The previously n was referring to the total number of nodes, but now, let this be the position of the key on the ring. So, then, this will be (n + 2^{i-1}), which is $(2^0 + 1)$. This is the start and the end will be (n + 2^{i-1}), which is (2^{1-1}), which is (n + 1). So, then if (i = 2), then this will start at (n + 2). (2^{2-1}) to (n + 3).

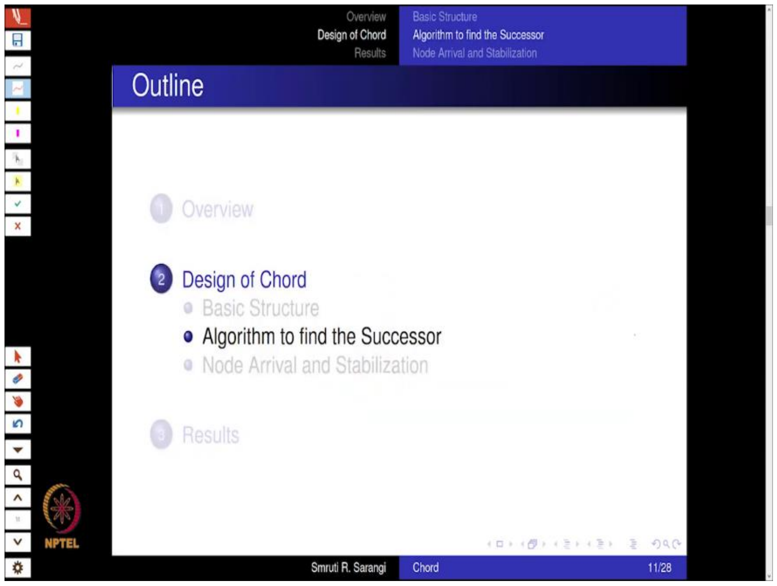
If (i = 3) this will be (n + 4) to (n + 7). Similarly, (i = 4), this will be (n + 8) to (n + 15). So, as you can see, we are gradually doubling the number of nodes in each finger. So, they are getting bigger, bigger, bigger and bigger. Ultimately till it encompasses half. So, as you can see this diagram is embodied in this math. So, then for the general i, this will be $(n + 2^{i-1})$ to $(n + 2^{i-1})$.

So, the first finger is small let slightly bigger, slightly bigger, big, big, big, big, big, and big till it reaches the semicircle. So, the first question is, why do we define fingers like this? Well, we will see there is a massive advantage. But we will gradually appreciate it. But I hope that the idea of these fingers is clear.

The way that we define these fingers, and each one of them, which are essentially drawing a line. So, that is also why the scheme is called chord. Because there is a circle at any time two points in a circle are joined by a line, it becomes a chord. So, finger i dot node. So, for every finger, we have a start and an end, which is only two positions with finger i dot node is the name of a node Id, which is the successor, the clockwise successor of the finger i start.

So, it is a clockwise successor or start. So, what is the basic operation? The basic operation is that for a given key, you find the successor you find the node Id that is its successor, which means that from the key, we start walking the circle clockwise, until we find the nearest node and that nearest node is the successor.

(Refer Slide Time: 21:23)



Overview | Basic Structure
Design of Chord | Algorithm to find the Successor
Results | Node Arrival and Stabilization

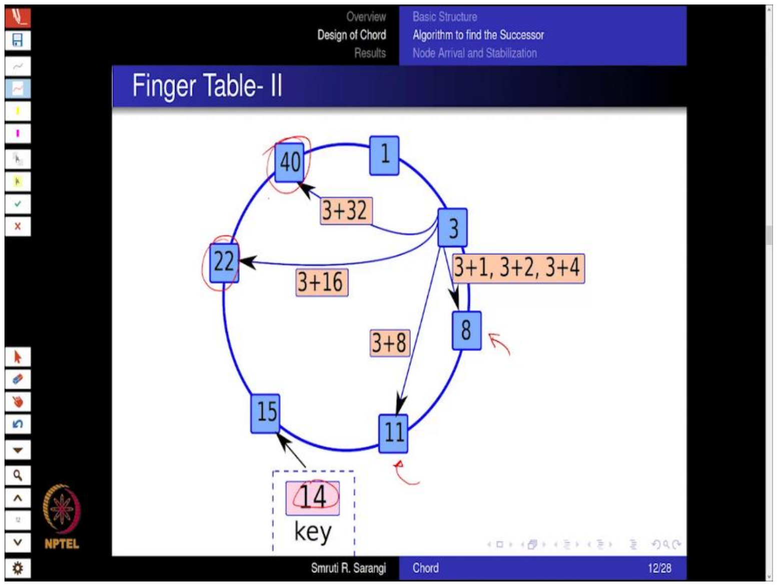
Outline

1. Overview
2. Design of Chord
   - Basic Structure
   - Algorithm to find the Successor
   - Node Arrival and Stabilization
3. Results

Smruti R. Sarangi | Chord | 11/28

So, what is the algorithm to find the successor? This is by far the most important algorithm in chord. So, we should take a look at this pretty seriously.

(Refer Slide Time: 21:35)



Overview | Basic Structure
Design of Chord | Algorithm to find the Successor
Results | Node Arrival and Stabilization

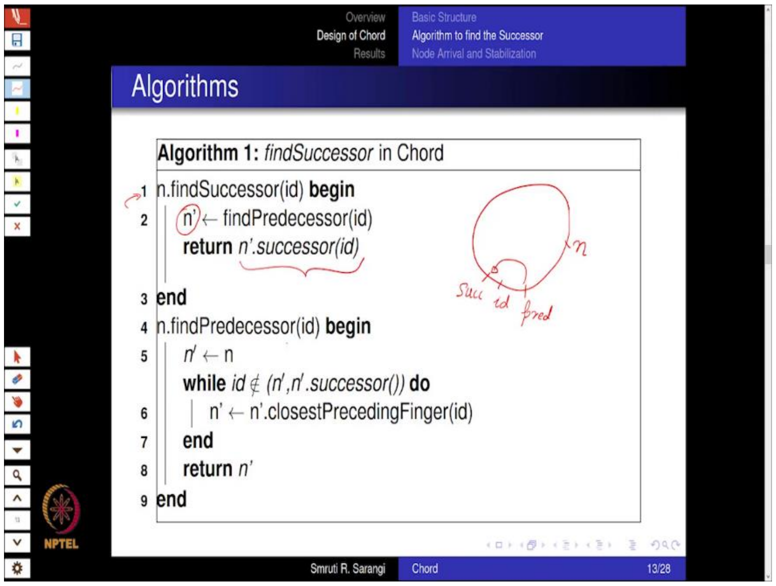
Finger Table- II

Smruti R. Sarangi | Chord | 12/28

So, again, just to come back, if I were to look at a circle, and give actual values, so then what we would see is for the node 3, and for the 3 fingers who start is (3 + 1), (3 + 2) and (3 + 4), their successor or the node or the fingers is not 8. But for (3 + 8), which is a next finger, there is already a node over there, so its successor is 11. And if let us say there is a key that arrives, the successor of key whose id is 14 → 15.

And (3 + 16) is the next finger, its successor would be node 22, and (3 + 32) next successors will be node 40. So, of course, this diagram has not been drawn to scale it is meant to show you certain things. But of course, it is not fully drawn to scale. So, do not expect a semicircle when you are not finding one. It is just an illustrative example.

(Refer Slide Time: 22:46)



Algorithms

Algorithm 1: *findSuccessor* in Chord

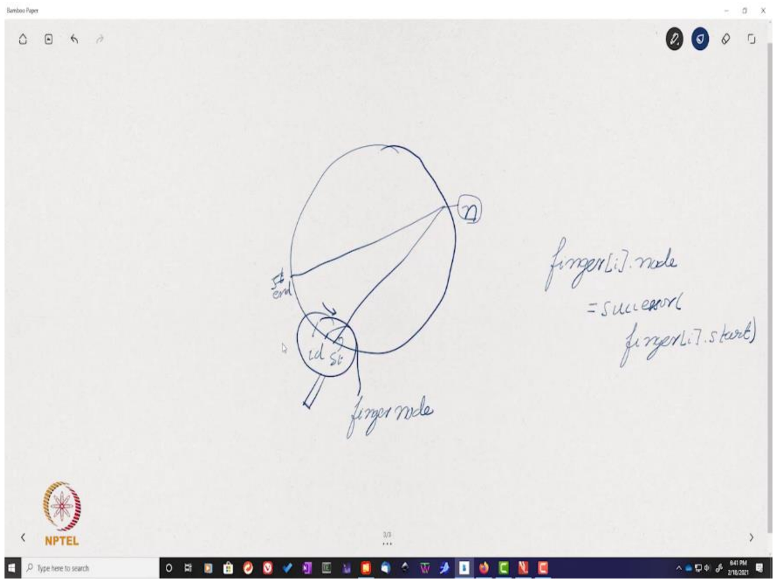
```
n.findSuccessor(id) begin
    n' ← findPredecessor(id)
    return n'.successor(id)
end
n.findPredecessor(id) begin
    n' ← n
    while id ∉ (n', n'.successor()) do
        n' ← n'.closestPrecedingFinger(id)
    end
    return n'
end
```

Let us now discuss the algorithm to find the successor. So, the key idea over here is that if I have the ring, and I have an id, over here. I want to find the successor node of this id, the one that this id is mapped. This CAN be the id of a key or it can be the id of a node. So, it does not matter at this particular point of time.

What it actually is? So, what we do is that we contact one of the nodes that we already know which is node n, which is a part of the ring and then we ask you to find the successor of id. So, we call the function over here n dot find successor id. So, what we do is that it turns out that to find the successor over here, it is actually easier to find the predecessor and since every node has a pointer to its successor, we arrive at the predecessor first and then from there we arrive at the successor and this node will also be the successor of id.

So, this is exactly what we do we initialize node n' which is the predecessor of id from n' we find the successor of n' which is what we need to return. So, to find the predecessor what we do is we follow a recursive procedure. So, I will show this in a slightly bigger diagram.

(Refer Slide Time: 24:22)



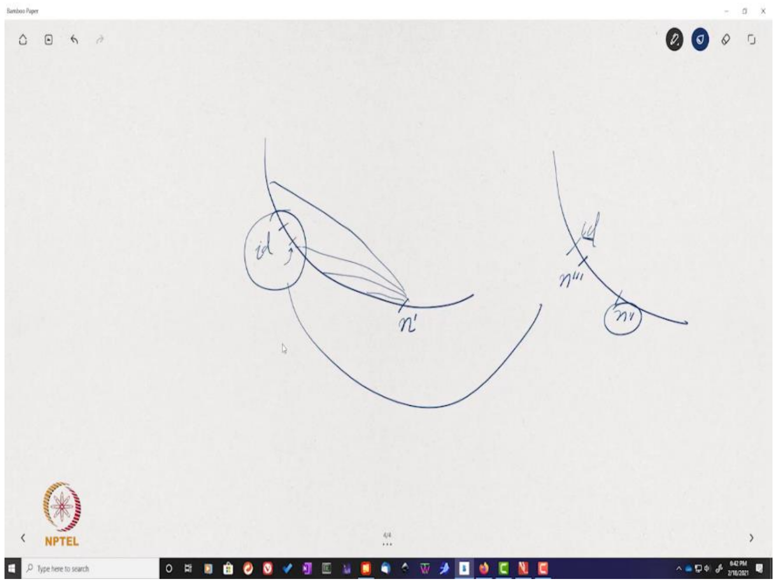
So, consider this to be the ring and consider this to be the starting position and this to be id. So, what we do is we find that finger of n which I am calling the closest preceding finger whose finger node. So, recall that the finger, fingers[i]. node = successor(finger[i]. start. So, finger [i]. node is the successor finger[i]. start.

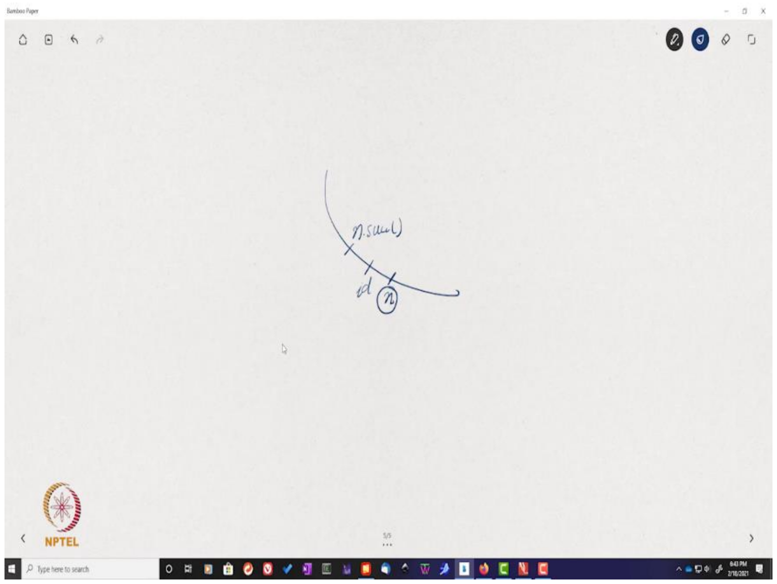
So, we find that finger of n, whose finger node is the closest preceding in the sense that if let us say start here and a walk anti clockwise, so I encountered many finger nodes of node n, but I want the closest one, the first one that I encountered while walking anti clockwise. So, it is possible that there could be a finger of this type.

So, this is the start of the finger. This is the end of the finger. And maybe the node of the finger is over here. So, this is the fingers node. So, let us call it the finger node. So, from here, I can immediately reach here, which is the best information that I have about id.

So, I can look through all my finger nodes and find the closest preceding one, which means that out of all my finger nodes, this is the one that is closest to it and precedes it. So, this I can do very easily, I just need to search within my finger table, node and needs to do that. Let us now blow this thing up. Let us have a magnifying glass. And let us blow it up. So, let us go to the next slide.

(Refer Slide Time: 26:28)





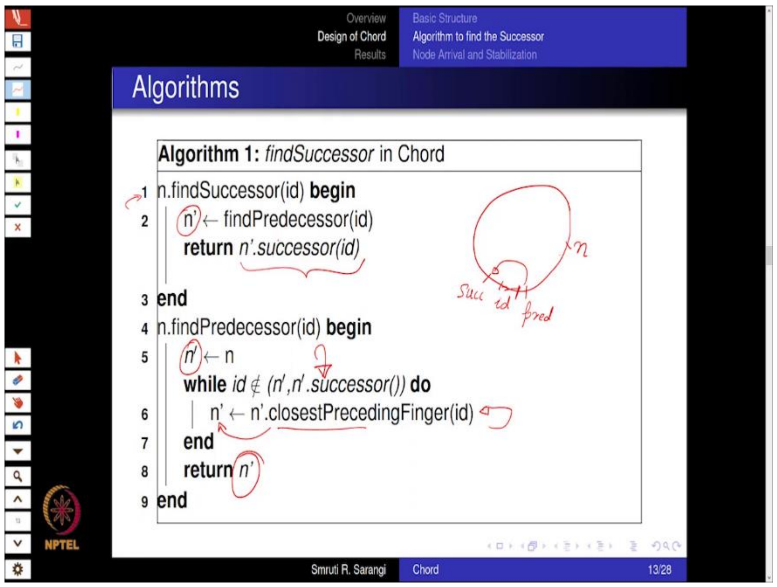
See if I blow it up, and then I look at it. So, it is possible that id is over here. And then the closest preceding finger which will again call node n, let me call it node n’ is here. So, again, this will have a bunch of fingers. So, one among those will be the closest preceding fingers, finger node. So, let us say that n’ will again do the same, it will again find this point, which is the finger node of the closest preceding finger.

So, the way that we have defined it, again, what we can do is we can blow this up. And again, we can look at this again, we will have an id over here again, we will have a point and dash over here, we can again look at its fingers and again come to a node which is even closer. And this is the closest preceding singer of id as per n’s information.

So, if this is n’, let us call it n’’. Again, I can start from this point which is n’’’ and keep on keep on narrowing myself narrowing and narrowing and narrowing until, until I reach a point when id is between whatever nodes I am considering, and the successor of the node. So, until id is between these two points, I will keep doing.

And I am ultimately, given the fact that I am continuously reducing the distance to id, an ultimately, guaranteed to find such a point. So, at this point, I actually stopped. So, this point tells me that I have found my position. So, at this point, I actually stop.

(Refer Slide Time: 28:20)



Algorithms

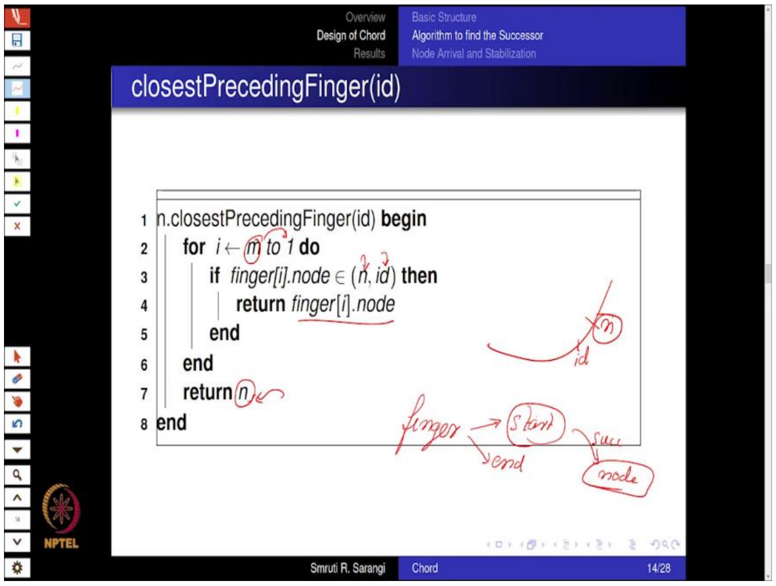
Algorithm 1: *findSuccessor* in Chord

```
n.findSuccessor(id) begin
    n' ← findPredecessor(id)
    return n'.successor(id)
end
n.findPredecessor(id) begin
    n' ← n
    while id ∉ (n',n'.successor()) do
        n' ← n'.closestPrecedingFinger(id)
    end
    return n'
end
```

So, exactly what we see in chord is something very similar. What we do is that we initialize n’ to n, as long as our condition is not met, which one the one that I showed just now, that id is not between n’ and n’s successor, I keep on finding the closest preceding finger, keep on setting that to n’ and I keep on repeating in a recursive fashion.

So, I just keep getting closer and closer. So, maybe this is as closest proceeding finger again, I get closer, closer, closer and closer. Ultimately, this condition will hold and that is when I know that I am done. So, I will return the value of n’, which is the predecessor and the predecessor successor is what I am after. So, now what I need to do is? I need to figure out how the closest preceding finger functions is written.

(Refer Slide Time: 29:13)

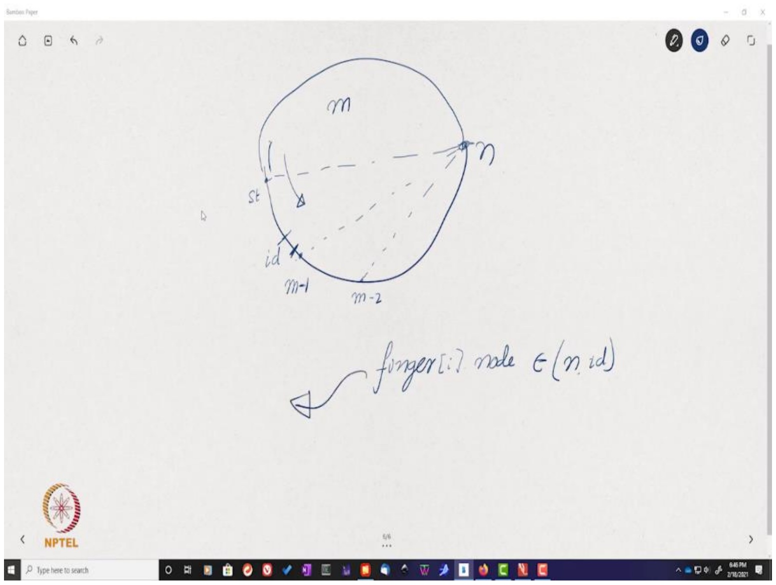


Overview | Basic Structure
Design of Chord | Algorithm to find the Successor
Results | Node Arrival and Stabilization

closestPrecedingFinger(id)

```
n.closestPrecedingFinger(id) begin
    for i ← m to 1 do
        if finger[i].node ∈ (n, id) then
            return finger[i].node
        end
    end
    return n
end
```

Smruti R. Sarangi | Chord | 14/28



So, the way that this is written is again rather interesting. So, again, I need to draw a big fat diagram to explain. So, if I were to consider this as the relevant part of the ring, or maybe I can show you the entire thing. Of course, this is not drawn to scale. So, this point is node n. And this point is id. So, what I do is, I start working backwards from the biggest finger to the smallest finger. So, this could be the biggest finger. So, from here I start walking backwards.

So, let us say that, if let us say this is the m’th finger, so then of course this is the starting point of the fingers. So, then the finger node will be after that, so I am not after this. So, may be let us say that this is the m - 1’th finger. And so, then I will start at this is the starting point, again, I will see if there is a finger node, the node of this finger is before Id if it is, then I will return that, otherwise, I will look at the starting point of the m - 2 with finger and also look at the finger node.

So, say the finger node is between n and id, and then I will return that. So, that is the way that I will proceed. The reason that I proceed this way anti clockwise and not clockwise is basically because I can kind of eliminate most of the circle by considering the largest finger, then I can eliminate most of it by considering the second largest, so on and so forth.

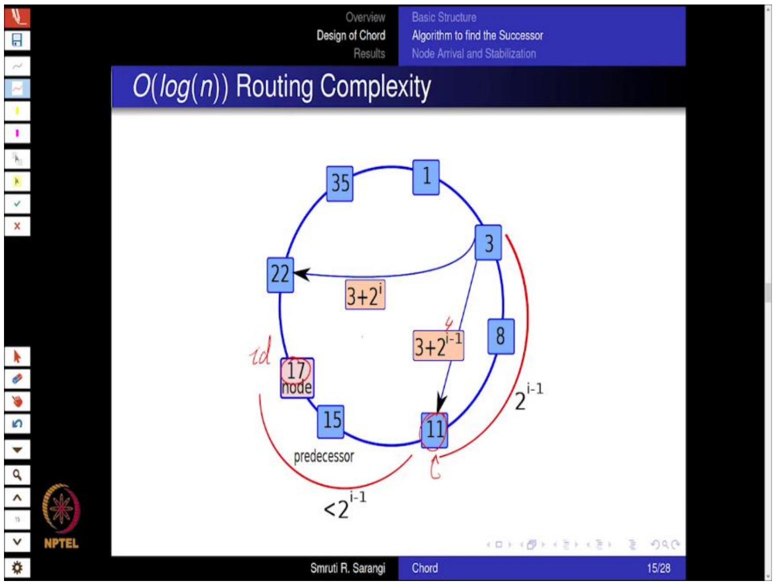
And furthermore, the first time I find a match, which means the what is the match? the matches that if let us say the matches for the i'th finger, it means that the node of the i'th finger is in between and then Id, if it is in between them, the first time I find such a match between the first time I find that look, this node is over here in between and an id, I can simply return that as the result of my function.

So, this is what I am doing in the chord that I am starting from the largest finger and counting down to the smallest. Any time that my function holds, which means that finger i. node, the node or finger i is in between n and id, the first time that it holds. I return finger i. node, I am done. So, there is a possibility that I will not find a finger i dot node that satisfies this condition.

When would that happen? If let us say n and Id are close by, so none of its fingers, the successor of the start or actually between this range, then it is easy, then it means that the current node n, n is the closest preceding fingers. So, that is like the trivial case, and n can simply be returned. So, this is a very important operation, I would request you to kindly go over this part of the video several times until you understand all of it, and particularly the special case, which indicates that the finger nodes, all the finger nodes are after id.

So, every finger recall is a very important concept. Do not forget, has the start has an end. So, that is the range of the fingers. But the most important thing that we are concerned about is the successor of start, which is node. And if you are unlucky the finger node can be after. And also, because there might not be any nodes within this range. This is an important thing that we need to bear in mind.

(Refer Slide Time: 33:01)


Overview
Design of Chord
Results
Basic Structure
Algorithm to find the Successor
Node Arrival and Stabilization

$O(log(n))$ Routing Complexity

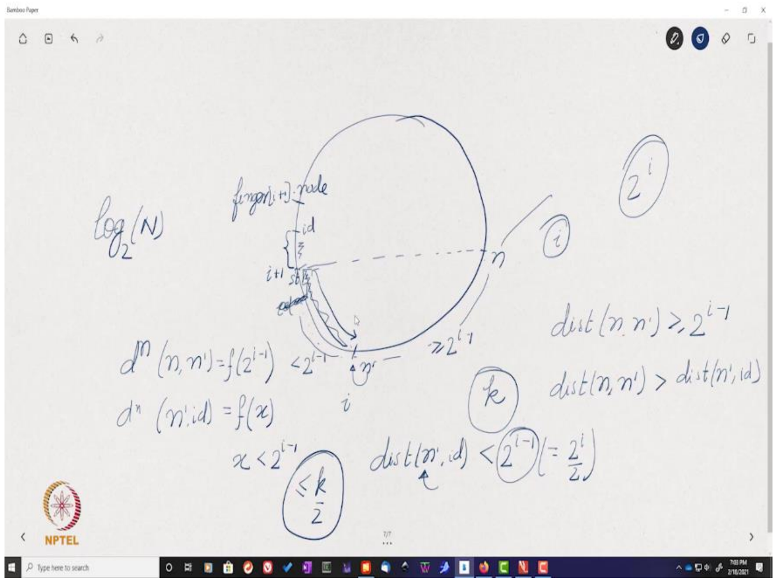
Now, here is the fun part. So, the fun part is that what is the complexity. So, when you are talking about the complexity of this algorithm, we really do not care about the number of operations we perform within each node. Because that is assumed to be extremely fast in any distributed algorithm. The only thing that we care about is how many messages are sent in that two sequentially. So, that is the only thing.

So, we do not care about what a node is doing, that is assumed to be infinitely fast. See if that is the case, if that is the case over here, we need to understand how many messages are actually sent. So, let us look at this concrete example over here. So, consider this example of a set of nodes. And we are interested in id number 17. So, this can be the idea of a key, idea of a node, it does not matter. It is not that important.

But here is the important part. The important part over here is that this is lying in the i'th finger. So, why do I say that because $(3 + 2^{i-1})$ is basically 11. There is a node over there. So, this is its finger node. And $(3 + 2^{i-1})$, so in this case, what is the value of I? well, it is i = 4. So, (3 + 16) = 19.

And clearly the finger node is the next finger, i + 1’th finger = 22. So, as far as 3 is concerned, the closest proceeding finger of 17 is node number 11. So, this is how we are going to proceed. This is an example. So, given that that is the case, can we derive some sort of a routing complexity.

(Refer Slide Time: 35:03)



$\log_2(N)$
finger[i+1].node
id
$i+1$
n
n'
i
2^i
$d^n(n, n') = f(2^{i-1}) < 2^{i-1}$
$d^n(n', id) = f(x)$
$x < 2^{i-1}$
$\leq \frac{k}{2}$
$\geq 2^{i-1}$
k
$dist(n, n') \geq 2^{i-1}$
$dist(n, n') > dist(n', id)$
$dist(n', id) < 2^{i-1} (= \frac{2^i}{2})$
NPTEL

For that, I will again take you back to my big board. So, where we will draw a big circle. So, I drew a circle as big as I can. So, here is a point. So, let us say I am starting at n. And this is where id is. So, in my estimate, on an average, so we expect that our nodes are uniformly distributed, homogeneously distributed across the ring. It is a massive ring, and I am using consistent hashing with a very random and uniform hashing function.

So, I can say that, with a good certainty that over all my nodes are uniformly distributed. So, now, let us say that between n and id, I expect k nodes to be there. Then I find the closest preceding finger and let us say the closest preceding finger is at this point, and this lies within the i'th finger. So, now the key point is that, after this, I will start my search from this point. How many nodes are expected to be between this new point?

Let us call it n' and id. How is this related to K? that is the most important question. So, the distance between n and n', if I were to measure it in just an id space, not in terms of the number of real nodes, the distance between n and n' will at least be 2^{i-1}, given the fact that it is within the i'th fingers. Furthermore, between n' and id, so it is not, it is in the i'th finger, it is not an i + 1'th.

So, here, there is a little bit of a complication. But let us consider the simple case first, so let us consider that i + 1'th finger will start from here, then you can clearly see that this distance is much more than this distance. So, I can say that the distance n to n' is more than the distance from n' to id. So, this is just the key distance, this is just the distance in the id space.

So, you can clearly see because if that was not the case, then the closest preceding finger would not have been i'th finger, it would have been i + 1'th finger. Of course, there is a little bit of a complication over here, which I will discuss later. But here again, I am considering the same simple case where within the i'th finger, we are finding it.

So, if that is the case, we can clearly see that the distance from n dash to id measured again in the key space is going to be less than the full distance, which is the distance from n to the next finger, which is 2^{i}. So, what is this, so the main idea is that the distance from n to n' = ≥ 2^{i-1}, and the distance from n dash to the beginning of the next finger, and assuming that id is before that, this is bound to be < 2^{i-1}.

So, I can write it over here. I mean, assuming that things are in control, so I will discuss when they will get out of control, not now. So, if that is the case, if this is the simple case, then what I can say is that even the i'th finger, if let us say I was considering a span of 2^{i} entries with n, then when i start from n', I will actually be considering a span off. Because n to id what was the maximum distance, it was 2^{i-1} + 2^{i-1}, which is 2^{i}.

Now, when I start from n', I will be considering a smaller span in the key space, which is 2^{i-1}, which in other words is $2^{i/1}$. So, with every closest finger operation, my span the area in the key space with that I am looking at that is reducing that is dropping by a factor of 2. But again, this is the key space.

So, now that is a little bit of a problem, it is very well possible that instead of this point. So, let us say that this is where i + 1'th finger begins, and my id is over here. Why is id over here? Because it is possible that the finger node of i + 1 lies over here, i + 1. node lies over here. So, then what will happen is this logic will not hold. But we also realize that there is no node in this region.

Had there been a? No, node in this region, finger i + 1. node would not have been here at the top, it would have been between the start of finger i + 1 and id, that has not happened. Given the fact that this has not happened, we are in a far better shape. So, what we can say is that look, this part is empty. So, as far as we are concerned, the distance from n' to id if I were to just count the nodes, I can ignore this part because I am sure that there is no node over here because of the logic that I just gave.

So, I can still say that if I were to break this an expected number of nodes. So, if let us say that let me define a new distance metric, which is the number of nodes that lie between n and n' expected number. So, let us call this let us say dn, between n and n dash will be some linear function of 2^{i-1}.

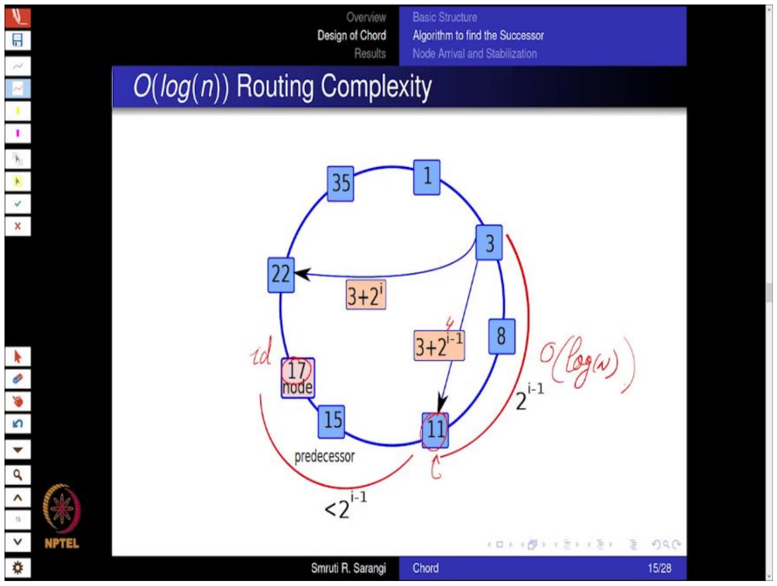
We can also say that the distance in terms of the number of nodes between n dash and id is also expected to be some linear function of a number of some number, which is basically let us have some number x, which will not really be the distance between n' and id in the key space, because we know that a large part of it is devoid of nodes. So, it will actually be some function of x where x is basically this distance, which is $< 2^{i-1}$.

So, using the same logic, regardless of where it is placed, we can say that even if we were to measure the distance in terms of number of nodes, and the number of nodes in a given region is proportional to the size of the region and the key space. Using that logic, we can say that look if there were k nodes between n and Id, between n dash and id we expect that there will be $\leq \frac{k}{2}$ nodes.

And the reason that I say that is because we are roughly halving the distance if it is lying within the i'th finger, even if it is lying outside the i'th finger, regardless of where it is lying from the end of the i'th fingers till wherever id is, there are no nodes. So, as far as we are concerned, we are primarily still, the unknown region is limited to this starting of the next finger and n'.

So, which is less than the other region. So, we can say that this is $\leq \frac{k}{2}$. So, every iteration of clauses preceding fingered, we are reducing the search space of nodes by a factor of 2. So, this will immediately tell us that look, if we just keep on dividing it by 2, ultimately $log_2(N)$, searches have to be done in terms of closest preceding fingers. And this is the number of messages we will have to send over the network till we ultimately find the predecessor and consequently the successor.

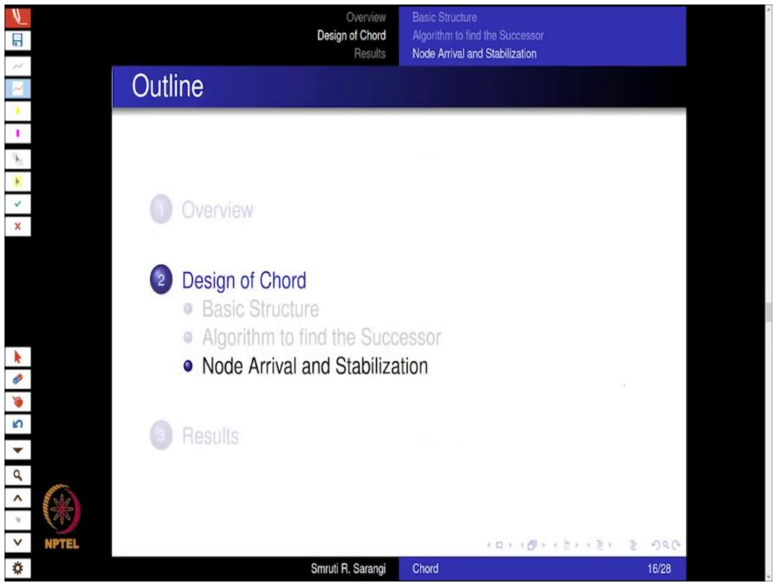
(Refer Slide Time: 43:46)


Overview
Design of Chord
Results
Basic Structure
Algorithm to find the Successor
Node Arrival and Stabilization

$O(log(n))$ Routing Complexity

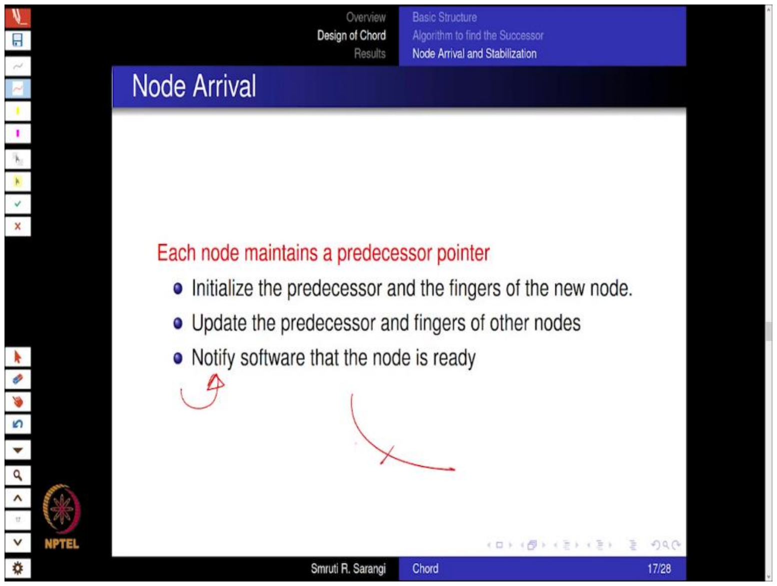
So, coming back to our presentation over here, the O(log(N)) logic is clear to us. So, it is not given that clearly in the paper, but you will have to consider all the sub cases and corner cases that I discussed with that it is very easy to see that it will be ordered log n and we also see this coming out experimentally as well.

(Refer Slide Time: 44:09)



Outline

1. Overview
2. Design of Chord
   - Basic Structure
   - Algorithm to find the Successor
   - Node Arrival and Stabilization
3. Results

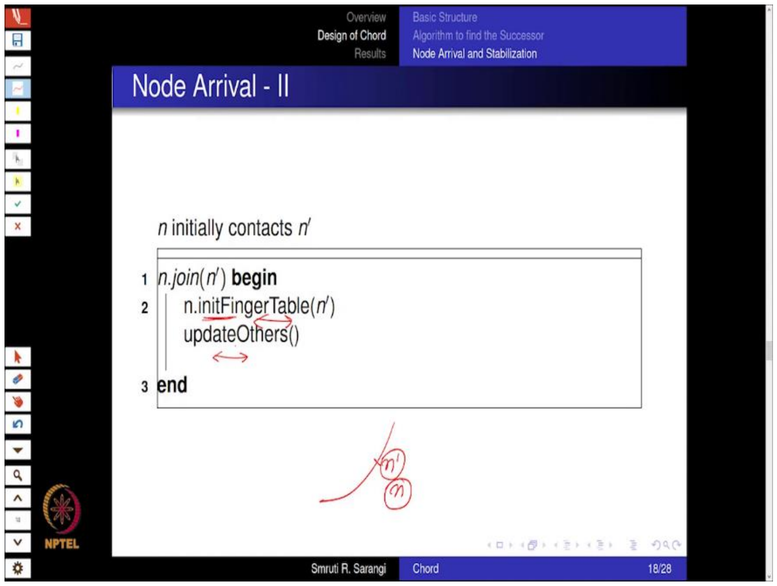


Node Arrival

Each node maintains a predecessor pointer
- Initialize the predecessor and the fingers of the new node.
- Update the predecessor and fingers of other nodes
- Notify software that the node is ready

Now, node arrival and stabilization, so, each node will maintain a predecessor pointer. It will initialize the predecessor and the fingers of the new node. It will update the predecessor and fingers of other nodes and then it will notify the software that the node is ready. So, in most implementations of Chord, we have a successor pointer as well as a predecessor pointer. So, we know which node is there on both sides, the successive node and the immediately preceding node on the ring.

(Refer Slide Time: 44:50)



Overview | Basic Structure
Design of Chord | Algorithm to find the Successor
Results | Node Arrival and Stabilization

Node Arrival - II

n initially contacts *n*′

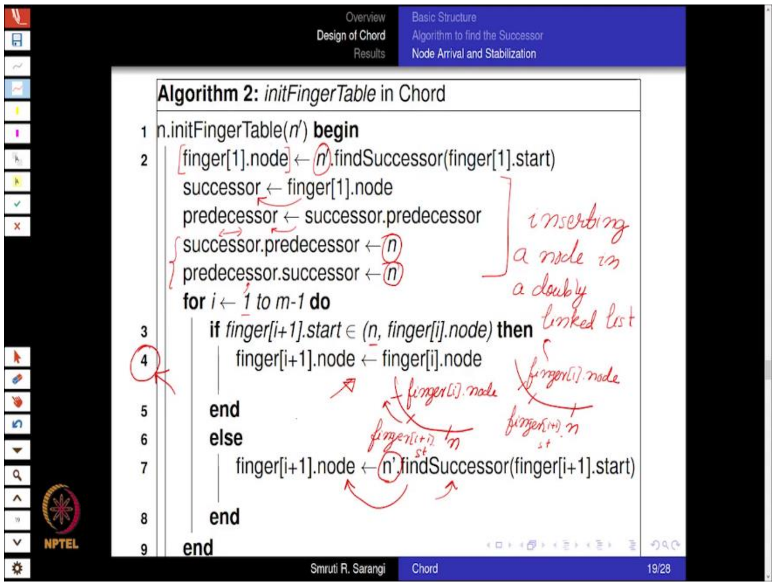
```
1 n.join(n′) begin
2     n.initFingerTable(n′)
      updateOthers()
3 end
```

Smruti R. Sarangi | Chord | 18/28

So, how does a node arrive? So, let us say that this is the node n and it wants to join the ring. So, initially, it will contact one of the nodes that is already a part of the ring. So, this will be let us say node n’, which we know already it will, that it is a part of the ring. So, then it will join. So, it will ask n’ to initialize its finger table.

So, there are two basic operations. The first is that for the node that is joining, its finger table needs to be initialized. Second, others have to be updated. So, we need to update others, it means the finger tables have other nodes saying that they need to point to me.

(Refer Slide Time: 45:42)



Algorithm 2: *initFingerTable* in Chord

```
n.initFingerTable(n') begin
    finger[1].node ← n'.findSuccessor(finger[1].start)
    successor ← finger[1].node
    predecessor ← successor.predecessor
    successor.predecessor ← n
    predecessor.successor ← n
    for i ← 1 to m-1 do
        if finger[i+1].start ∈ (n, finger[i].node) then
            finger[i+1].node ← finger[i].node
        end
        else
            finger[i+1].node ← n'.findSuccessor(finger[i+1].start)
        end
end
```

So, the init finger table this is how it works. So, n. init finger table n'. So, let us consider the node for the first finger, finger[1]. node. So, we will ask the node n' to find the successor of finger[1]. start. So, that is pretty easy. So, we will just ask node n' that you kindly initiate an algorithm and find the successor of my start. And that is done. So, that becomes my successor, so finger[1]. node as I said every node maintains a pointer to its successor.

So, once the node n' has done this basic service, and it has found the successor for the first finger, that is also the successor of the current node. And that can be stored in the successor's pointer. Furthermore, the predecessor is actually the successor's predecessor. So, what we can do is we can ask this successor node look who's your predecessor, and we can initialize that. Then we need to do a little bit more of math.

So, we need to also record that the successor's predecessor is the current node n. And this will require a message we need to tell the successor that look, I am your predecessor, and we need to tell the predecessor node that look, I am your successor, the predecessor dot successor the current node n. So, this is no rocket science. It is similar to inserting a node in a doubly linked list, so this is no rocket science.

Then what we need to do is we need to look at the fingers. So, for i ← 1 to m - 1, here is what we do for all the fingers. So, we look at the i + 1'th finger, say finger[i + 1]. start, so if let us say for the current node whose node is being initialized, if the start of its I + 1'th finger is between n and finger[i]. node. So, what does this mean? That I am node n, this is my finger[i]. node, let us just call it maybe finger[i]. node.

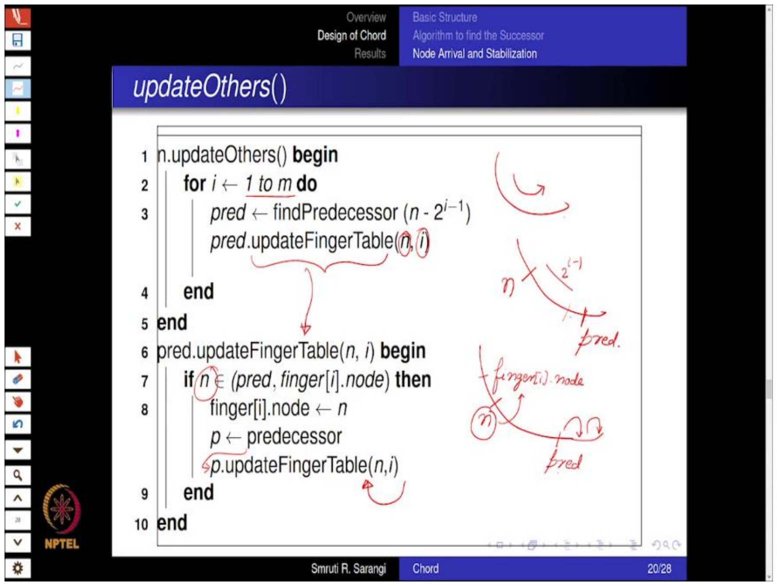
And let us say finger[i + 1]. start is over here. It is starting over here, then, it is very easy to see that finger i + 1'th node will also be the same node because from here, if we start walking down the ring clockwise, the first node we will encounter is finger[i]. node which is its successor. So, we can say that finger[i + 1]. node is finger[i]. node. So, this should be obvious, if this is not obvious, it means that the way that these fingers and finger nodes are created, that is not clear to you.

So, if let us say you are not able to understand line number 4, the logic behind line number 4, the way that I explained, let me draw it again to make it slightly clearer. So, this is finger[i]. node. So, as I said, if this is not clear to you, kindly do not proceed. So, at this point, this should be absolutely clear to you, why I am doing, what I am doing. So, I am saying if this is finger[i + 1]. start.

From here, I will just traverse the ring clockwise. And I should arrive at finger[i]. node and that should be the first node that should be my successor. Because from finger[i]. start when I walk, this was the first node that I saw. So, if this case is holding, this will also be the first node that I will see. But as I said, it is very important to understand this line before you actually proceed.

So, then, if this is not the case, if this case is not happening, then we will again have to request node n' to find the successor finger[i + 1]. start for us. So, it needs to do the same again, it needs to again find the successor of this for us, and then we will use it to initialize. So, we just keep on doing it, we just keep on doing it for the rest of the (m – 1) finger. So, this will initialize the finger table completely.

(Refer Slide Time: 50:26)



updateOthers()

```
n.updateOthers() begin
    for $i \leftarrow 1$ to $m$ do
        $pred \leftarrow$ findPredecessor $(n - 2^{i-1})$
        pred.updateFingerTable(n, i)
    end
end
pred.updateFingerTable(n, i) begin
    if $n \in$ (pred, finger[i].node) then
        finger[i].node $\leftarrow n$
        $p \leftarrow$ predecessor
        p.updateFingerTable(n,i)
    end
end
```

So, now once my own finger table has been fully initialized, what I need to do is I need to update the finger table of others. So, what I do is I take a look at all the m fingers, for the i'th finger, we find the predecessor of (n - 2^{i-1}), which means that if this is my current id, I just subtract 2^{i-1} which means that I will lie in i'th finger or somewhere. So, from this point, I find the predecessor which is maybe this node.

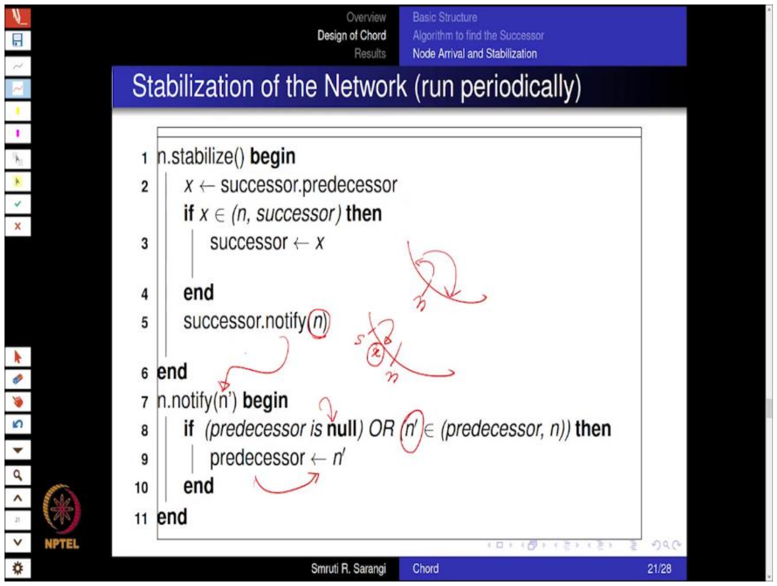
So, for this node, I am going to be the i’th finger. So, I go to this node pred and I will send a message and will pretty much say that look, I am now your i’th finger’s node. So, you kindly update your finger table and say that look node n is a node for i. So, what I would do over here is that I will call the update finger table function over here, which you see here at the bottom. So, I will do it for all my fingers.

And each time I will call the update finger table function. So, when I am doing that, here is a check that I am going to do, it is very important check. If let us say for node between pred. So, for its i’th finger it will have node, it will have finger[i]. node. And let us say the current node n lies somewhere here. If it does not lie, I am not interested.

But let us say it lies somewhere there. Then the current node n will be set as the i’th finger node for pred. That is point number 1. Furthermore, there might be a need to cascade this information. So, it will look at its predecessor, it will then send a message to its predecessor that look, I have gotten information that node n has been added, node n sent me a message, I found the message to have merit in it so added node n, which was lying between me and my previous finger[i]. node.

So, I added it as the finger node for my i'th finger, it might be a finger node for you as well. So, why do not you also kindly check. And if that is the case, then you also update yourself. So, then it will keep on cascading this to its predecessors until all of them update. So, this procedure is done for all the fingers. So, by this we basically ensure, so mind you, we only walk in a certain direction, which is anti-clockwise, because that is the way that we created our finger table. So, once the rest of the nodes have updated their finger table, the process of node addition is done.

(Refer Slide Time: 53:34)



Stabilization of the Network (run periodically)

```
n.stabilize() begin
    x ← successor.predecessor
    if x ∈ (n, successor) then
        successor ← x
    end
    successor.notify(n)
end
n.notify(n') begin
    if (predecessor is null) OR n' ∈ (predecessor, n)) then
        predecessor ← n'
    end
end
```

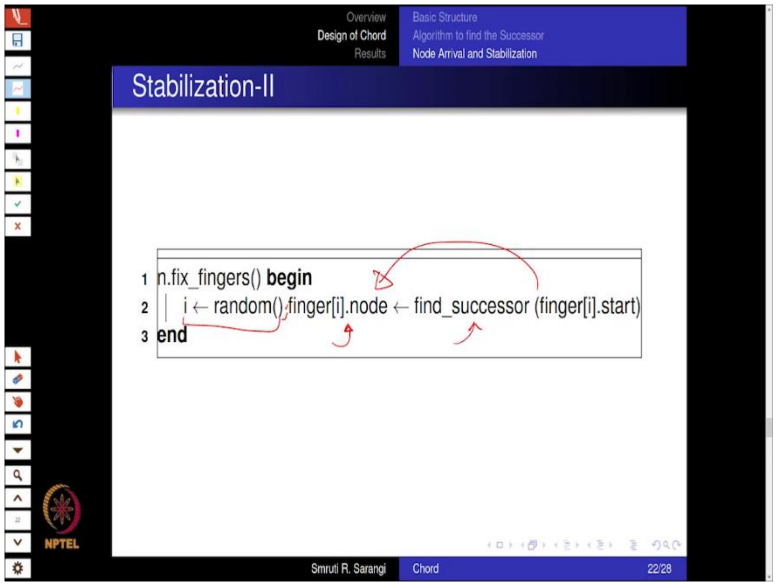
So, then we will do a little bit of node stabilization. So, the node stabilization is required particularly if you have race conditions and so on. So, what I will do is I will look at my successor's predecessor, so I will look at, if this is the node, I look at my successor's predecessor. It can be the previous predecessor or it can be me.

So, if let us say that my successor's predecessor, so, this is essentially looking at this, this is me, this is my successor and this could be my successor's predecessor, if that is the case and this node x is between n and the successor then the successor will be made x. So, this is one way of periodically fixing the thing. So, I check that am I my successor's predecessor, if I am not, then is there a new node x that lies between me and my successor.

If it lies and still the network is in a position of getting updated because as I said, it is a race condition. If that is the case, I will simply set x to be my successor and I am done. And furthermore, I will notify the successor the new successor that look, I am your predecessor. So, this notify function will be called.

And in the notify function, if there is no predecessor or the node that is claiming to be the predecessor is actually the predecessor, because it lies between me and my previous predecessor, then I set predecessor to n'. So, both these operations are what are called stabilization operations. So, they help to fix any temporary inconsistencies in a network. So, those issues get fixed.

(Refer Slide Time: 55:31)


Overview
Design of Chord
Results
Basic Structure
Algorithm to find the Successor
Node Arrival and Stabilization

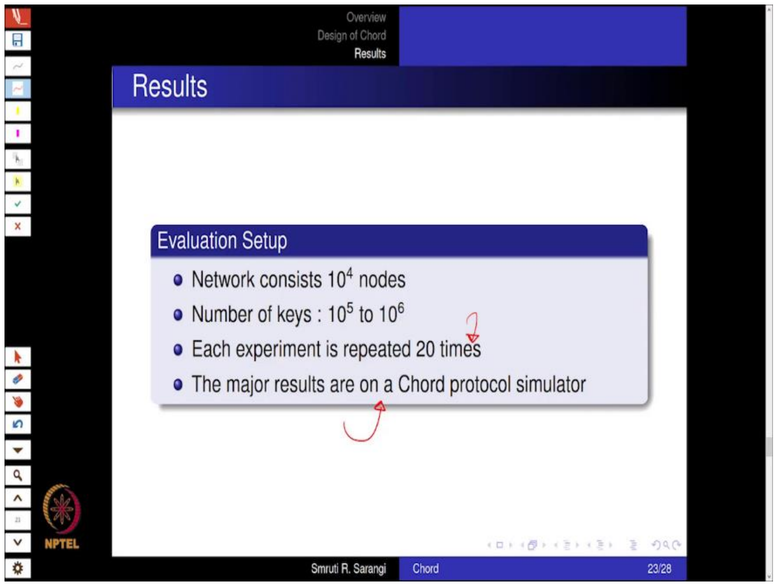
Stabilization-II

```
n.fix_fingers() begin
    i ← random(); finger[i].node ← find_successor (finger[i].start)
end
```

So, then also, we can fix the fingers, so let us say that some error has cropped up somewhere. And because of that, there is a little bit of inconsistency in the bookkeeping information that is there all over the network. What I can do is I can fix fingers, which means that at random if let us i is a random number at random I should have a semicolon over here.

So, finger[i]. node, the node of the i'th finger i can set it to find the successor of finger[i]. start, so this should hold anyway, but because as I said because of some temporary inconsistency that may crop up this may not be holding. So, this is where there is a need to periodically run the script and keep on fixing the network.

(Refer Slide Time: 56:28)



Overview
Design of Chord
Results

Results

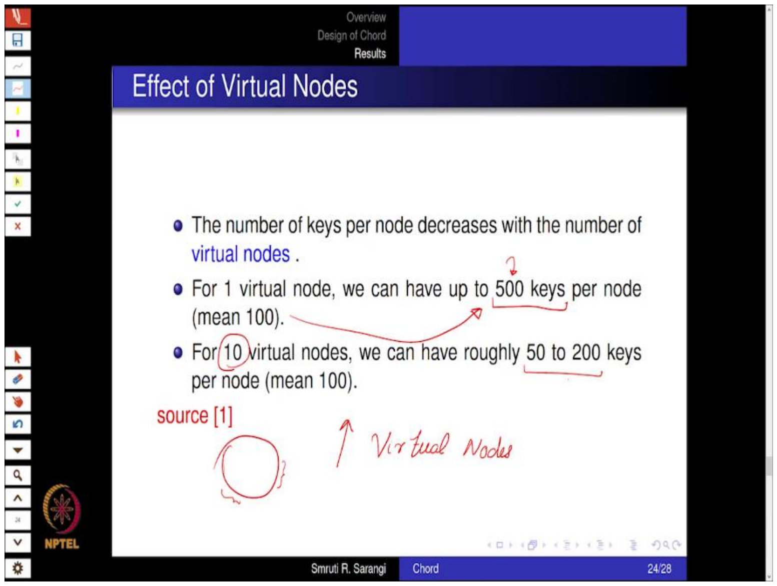
Evaluation Setup

- Network consists 10^4 nodes
- Number of keys : 10^5 to 10^6
- Each experiment is repeated 20 times
- The major results are on a Chord protocol simulator

Smruti R. Sarangi Chord 23/28

So, now we will discuss a thing or two about the evaluation set up. So, the network here consisted of 10,000 nodes, and the keys varied from 100,000 to a million and the experiment was repeated 20 times, you can also see error bars in the figure. And they did not actually do it on a real data center on a real distributed system. This experiment was done on a simulated environment.

(Refer Slide Time: 56:59)



Overview
Design of Chord
Results

Effect of Virtual Nodes

- The number of keys per node decreases with the number of virtual nodes .
- For 1 virtual node, we can have up to 500 keys per node (mean 100).
- For 10 virtual nodes, we can have roughly 50 to 200 keys per node (mean 100).

source [1]

↑ Virtual Nodes

Smruti R. Sarangi Chord 24/28

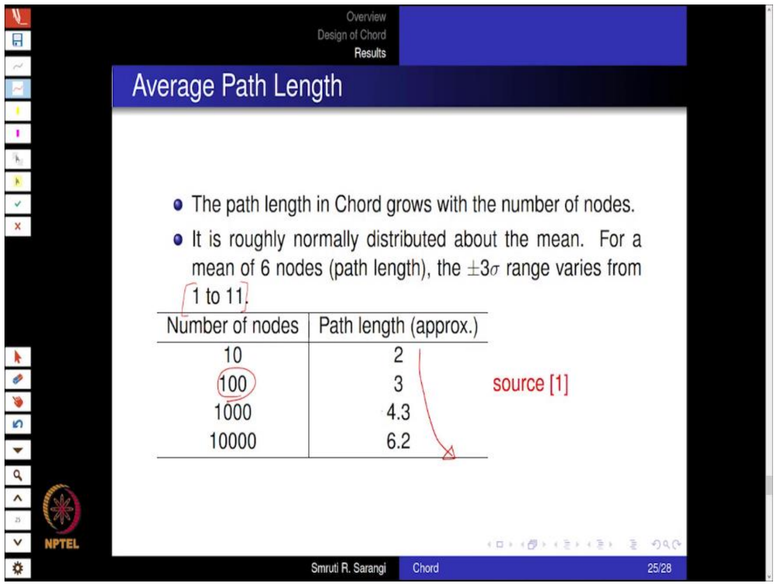
So, what we see is that the number of keys per node decreases with the number of virtual nodes and also why did we add virtual nodes, well we added virtual nodes for better homogenization and to ensure that no node becomes a hotspot. So, for one virtual node we could have up to 500 keys per node with a mean of 100. And for 10 virtual nodes, we can have roughly 50 to 200 keys per node.

So, what you can clearly see over here is that if you have less virtual nodes per physical node, then the lack of uniformity is large. So, from 100, it is going to 500. So, there is a lack of uniformity. But if I have 10 virtual nodes per physical node, the way that we have been arguing, you can see that we will only have this the uniformity is more or let us say the load is equally balanced across the physical nodes.

So, go back to the slide where we discuss virtual nodes. So, here we have roughly 50 to 200 keys per node. So, kindly hang on to this concept of virtual nodes is very important. So, this will be used in a third part of the chapter when we will discuss commercial networks because they also use virtual nodes for exactly the same reason.

So, this is used as a load balancing technique as a load balancing trick to ensure that even if we are using the best hashing algorithm, it will happen in real life, that some nodes a lot of keys will be mapped to it and for some nodes very few keys will be mapped to it. So, as you can see, you will see higher spread, if you want to reduce that have more virtual nodes per physical node, that will solve this problem to a large extent, as you are seeing over here.

(Refer Slide Time: 58:51)



Average Path Length

- The path length in Chord grows with the number of nodes.
- It is roughly normally distributed about the mean. For a mean of 6 nodes (path length), the $\pm 3\sigma$ range varies from 1 to 11.

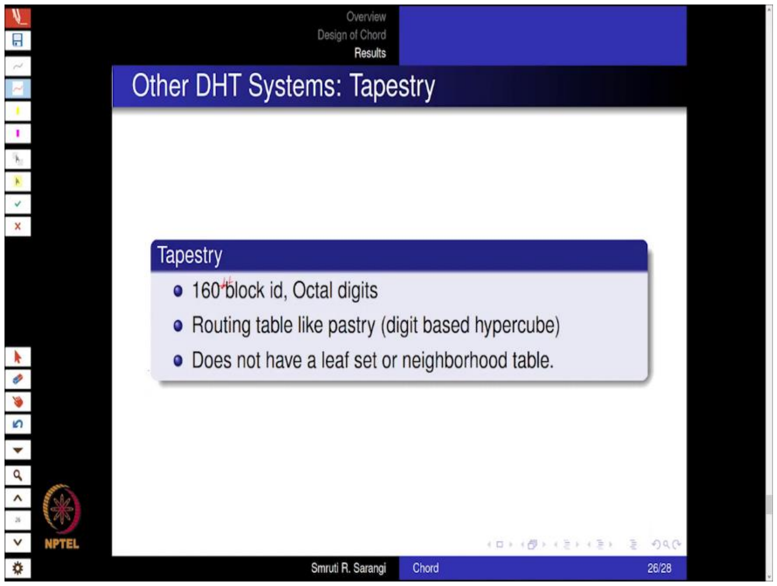
Number of nodes	Path length (approx.)
10	2
100	3
1000	4.3
10000	6.2

source [1]

So, the path length in Chord grows with a number of nodes. So, if you see the paper it will show that it is roughly normally distributed about the mean, so for a mean of six nodes, so mind you, for a similar network, the path length in Pastry was much lower, the path length in Chord is slightly more, so the ± 3σ range.

So, is approximately if you look at the mean, it will the ± 3σ range does vary from 1 to 11. And the corresponding variance in Pastry was slightly lower. And but as you can see, it increases in a nice log N kind of fashion. So, it is around 2 for a 10-node network, 3 for 100 node, 4.3 for 1,000 and 6.2 for 10,000. And so, in that sense, it is scalable, even though the overheads are slightly more than Pastry.

(Refer Slide Time: 59:46)



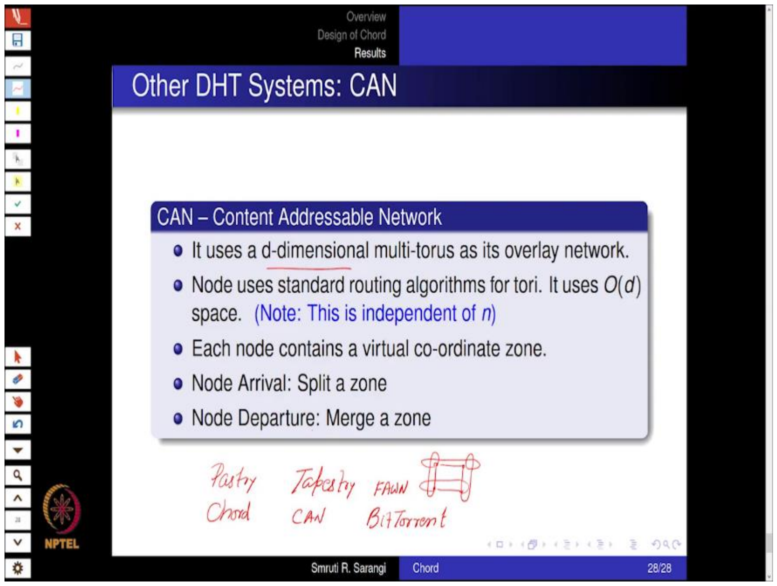
Other DHT Systems: Tapestry

Tapestry

- 160 block id, Octal digits
- Routing table like pastry (digit based hypercube)
- Does not have a leaf set or neighborhood table.

So, there are other DHT systems it is just not Pastry and Chord. So, for example, you have a Tapestry that uses a 160-bit block id with octal digits. So, the routing is like Pastry, it is a digit-based hypercube. So, it does not have a leaf set or neighborhood table. And also, it is not necessary that all designs will have a ring-based overlay like Chord and Pastry. So, in this case it is a hypercube, so it has a different kind of an overlay.

(Refer Slide Time: 1:00:24)



Overview
Design of Chord
Results

Other DHT Systems: CAN

CAN – Content Addressable Network

- It uses a d-dimensional multi-torus as its overlay network.
- Node uses standard routing algorithms for tori. It uses $O(d)$ space. (Note: This is independent of n)
- Each node contains a virtual co-ordinate zone.
- Node Arrival: Split a zone
- Node Departure: Merge a zone

So, Kademlia is the basis of BitTorrent. So, each node has a 128-bit id where each digit contains only 1 bit. So, Kademlia will be the next lecture in this lecture series. And we find the closest node to a key. So, there is some amount of reliability in the network in the sense values are stored at several nodes and nodes can also cache the values of popular keys. So, that is also doable. So, we will discuss BitTorrent next.

And one of the most influential proposals in this area is can content addressable networks, so we have discussed this earlier also where we use a d-dimensional multi-torus as overlay network. So, recall that a torus is basically a network it is a mesh with the corners connected. So, we have long connections of this type. So, this is of course 2-dimensional, we can have the same thing in 3d, we can have the same thing in n-dimensions.

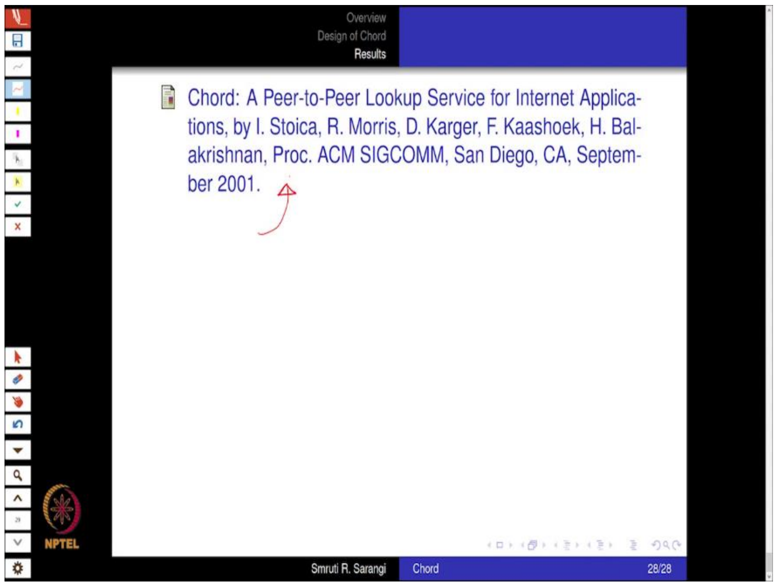
So, it uses standard routing algorithms for tori, tori is the plural of a torus and uses a virtual coordinate zone. So, whenever a node arrives, we split a zone and when a node departs, we merge a zone. So, essentially, we break this into zones and that is how we do the routing. But the can paper is available online. So, I would request viewers of this video to take a look at the can paper as well.

So, these were the most popular DHT, so we started with Pastry, Chord, so we will have a short 15-minute video after this to discuss BitTorrent and Kademlia. And that will heavily build on whatever has been taught in the last two lectures on Pastry and Chord. So, of course, TaPastry and

CAN and Kademlia are all there and there are some new papers. So, I just maybe write it down, there is something called Fawn, which is for extremely small nodes based on USB sticks.

And but BitTorrent is clearly the most popular as of today, when it comes to openly available networks, and almost all major companies like LinkedIn, Amazon, Facebook, etcetera, they have their internal DHTs which store data. So, they of course, instead of a login time access, they have a one hop access, so we will discuss them in the third part of this course.

(Refer Slide Time: 1:03:06)



Chord: A Peer-to-Peer Lookup Service for Internet Applications, by I. Stoica, R. Morris, D. Karger, F. Kaashoek, H. Balakrishnan, Proc. ACM SIGCOMM, San Diego, CA, September 2001.

So, this was the Chord paper. And so, the next lecture will be on BitTorrent, but it will be very brief.