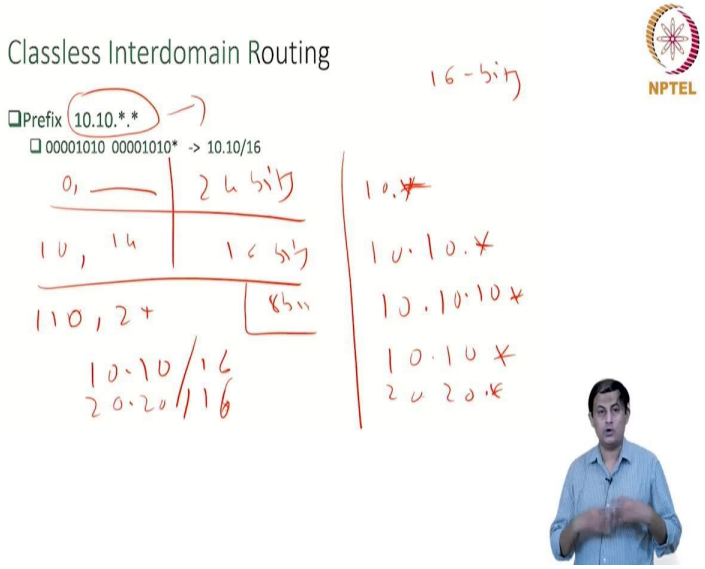
Advanced Computer Networks
Instructor. Dr. Neminath Hubballi
Department of Computer Science Engineering
Indian Institute of Technology, Indore

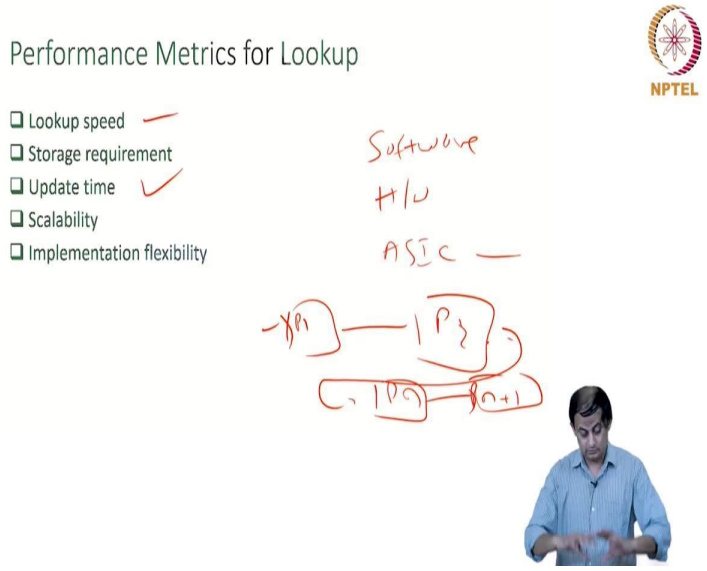
Lecture 4
IP Table Lookup – Part 2

(Refer Slide Time: 00:19)



Now, given the scale of this operation and the speed at which we need to do this, how do I actually do this, what is the thing that I need to have inside the router?

(Refer Slide Time: 00:33)



And how do I actually evaluate the performance of some forwarding mechanisms, there are multiple options you can think of, one is I can implement this as a part of the software solution, I write a code, and then I execute that, and every time you receive a packet, and that code actually goes and searches inside the table, whether some entries are matching.

And the second thing is, I can implement this in the hardware and give this routing table to that hardware and then that will do the comparison and give you what is the best route or the port number on which you need to forward this. And the third option that you got is something called the ASIC or application-specific integrated circuit.

So, one option is you implement using some data structure, you implement as in the software code. Or the second option is you can use the hardware something like FPGA or other components. Or third option is you build something called as ASIC, ASIC stands for Application Specific Integrated Circuit, what it means is, this hardware is customized for only doing that lookup operation forwarding decisions, nothing else.

This is not a generic purpose order, meaning that you can compare that our computers are generic purpose processors; you can do a lot of things on that I can run in Word Processor application, I can play a video, I can do something else, I can write a C program and execute all of this is running on the same processor. But this ASIC is not something like that.

This is if it is designed for one purpose, then only for that purpose; it can be used, if it is meant for doing the lookup operation in the routers, only for that you can actually use this ASIC. So,

that is how those are the options that I got. Now, which one is better? So, whether ASIC is better, whether the software implementation is better, or I can use the hardware and then do the implementation. So, I need to make an informed choice.

So, how do we evaluate the performance of the lookup operation? What are the metrics that are available? Whether the software implementation is good enough for me, or is it not enough? So, one obvious metric that we want to use is something called the lookup speed; how much time it takes to do the search operation or the string matching?

So, larger the time it takes, worse it is. So, lesser the time it takes to do these operations better it is. That is one obvious metric that we want to use. And the second thing is, how much amount of memory or storage the routing table itself takes? So, if it is taking too much time, and too much space, then probably that is not enough.

So if I had to store the entire 32-bit type address in the routing table, then that is going to be actually a lot of, and other than this, the data structure that you are going to implement will also consume some amount of the memory. So, once I represent the routing table in the form of some structure, how big that is going to be, whether it is going to be very optimized, or it is going to take lots of memory, is one another metric that is going to govern.

And the third thing is, as I said, as new people join the network, there will be new entries that are supposed to be created inside the routing table. It requires an update operation. So, what it means is, if I structure the routing table using some data structure, let us say hypothetically for a data structure X, and what it means is when new routing entries come in, I want to add this it means that I need to modify that data structure X to accommodate that new entry.

So, what is that it takes to do the real-time? Do I need to completely change X and rewrite from scratch? A large number of changes are required. Let us say I represent in the form of a linked list. And every entry in this one tells about prefixes, this is the prefix number 1, this is the prefix number 2, and so forth. Let us say I have got n number of such prefixes. And I want to add n+1 prefix. So, this might be just adding a link from the P_n to P_{n+1}, this probably does not require much of an addition.

But on the other hand, think of this case; all of these are actually in the sorted order. And you are going from the one sequence P_1 to P_2 and up to P_n. And if the new entry n+1 is putting, let us say

after the 100th entry, then you need to traverse all of them, insert that node at the 101 position push the existing 101th node to the right, and then do the work.

So, if I come up with some data structure to implement or represent this routing table, and new entries are going to come, our existing entries might need to be deleted. So, in that case, how much effort it takes to do that update operation or the deletion or the addition of an entry in the routing table? that is another performance metric. Something which is taking less amount of effort and time to do the update operation is the better one for us.

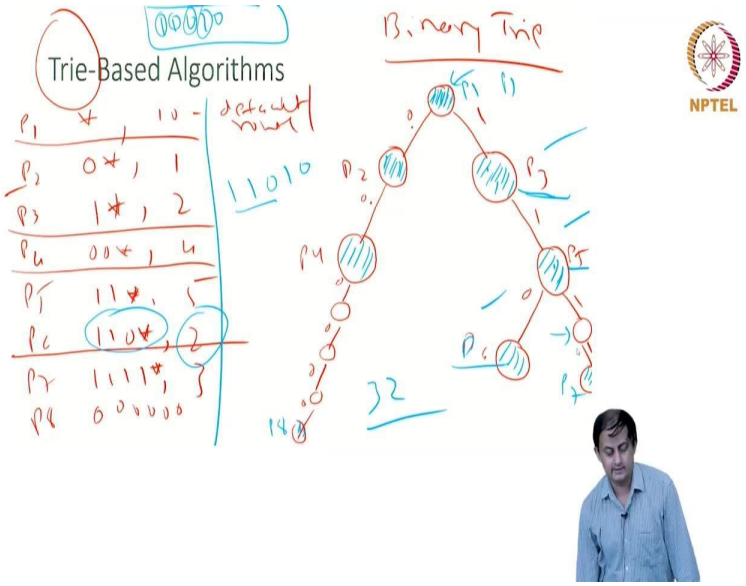
And fourth metric is obviously scalability; we are talking about the routing tables, which are going to keep the entries for the entire Internet, for the routers, typically for the router, which are running in the backbone of the internet service provider networks. So, obviously, new people will be going to join, new prefixes will be coming, and you need to have a scalable implementation. This means that the data structure that you put should be able to handle large, large amounts of the prefixes.

And the fifth metric that one uses is something called implemented flexibility. So, can I make the changes quite quickly? If I want to migrate from one to another one, can I do that? If the data structure is able to give you that flexibility, then better it is. So, in nutshell, I got a bunch of options. So, the lookup operation needs to happen quickly. And I have got an option to implement that lookup operation as a part of the software code. Or I can build customized hardware, or I can use something called ASIC.

So, in the beginning, obviously, when the network started, people started implementing the routing table in the software, and our implementation would be obviously slower than the hardware. And then slowly, people migrated to the hardware, and you know we do have a mixture of the software implementation and also the hardware implementation.

So we will seek the exact data structures or some of the data structures used for the software implementation if I have to do that and then if I have to do the search operation on the routing table for this inside the hardware, how do we exactly do that both the things inside this course.

(Refer Slide Time: 08:36)



Let us take a look first at the software-based implementations, how they exactly look and what kind of data structures can I use to accommodate or store this routing table. So, what I am going to do is, I am going to have a bunch of prefixes, so let us call that prefixes with the notation P, P_1 is one prefix, maybe I can put the * , irrespective of any IP address that you got, you forward it to port number 10. That is what the prefix number routing table number entry 1 is saying. And similarly, this is also called the default route.

And most of the routers actually have a such default route. So, if you do not find a matching entry, you will do the routing by this entry. So, a routing table is set to be complete when you are able to make a forwarding decision for every incoming packet, does not matter what is the destination IP address of that particular packet; you should still be able to do the routing.

So, if you are not having entries for some of the prefixes inside your routing table, in order to make the routing table complete, what you do is insert this default routing entry inside your routing table. So, that makes it complete. So, you have some entries for some prefixes, and the rest of the prefixes for which you do not have the entries you do the routing by this default.

The second one P_2 prefix might say that it is starting with 0*. So, any IP address which is starting with 0 does not matter what the rest of the 31 bits are, you need to forward it to port number 1. The third one is saying if your IP address is starting with 1 and does not matter what the rest of the 32 bits are, you forward it to port number 2.

Similarly, I got might be 00* this is going to port number 4 and P_5 prefix number 5 is saying if it is 11*, then you forwarded it to port number 5 and P_6 is saying 110* , if the starting IP address 110, then you forwarded it to port number 2 and the P_7 is saying 1111*, you need to forward it to port number 3.

There are seven prefixes inside my routing table. And these are the entries. Now, if I have to do a software implementation, what is the thing that I can do? So, one of the simplest algorithms that you can think of here is something called the Trie-Based structure. Trie is a generic name is pronounced as try. Trie is n-ary tree, a node can have n different children.

You can construct a trie incrementally for these prefixes, meaning that today I have these seven prefixes; I am going to build a tree or the trie for only the seven prefixes. And I get a new one, which is the 8th prefix, then I am going to incrementally modify this trie structure to accommodate that 8th prefix.

So, how do I actually build this, the simplest form of the trie is called something called the binary trie. So, as the name suggests, there are at the max two children for every node. I start with the way we construct the routing table or the trie for this set of prefixes. I start with the root node and I could pick up one prefix, and then add the nodes to the left of this root node or to the right of root nodes to accommodate what it corresponds to whatever the prefix is saying.

So, every time I have a 0 in the prefix, I take the left part; every time I have 1 in the prefix, I go to the right part. So, let us take this for a time being, let us ignore this P_1, let us take this P_2, what it says is? It is 0*. So, I am going to add the left child to this root node. This is my P_2 and third one P_3 saying 1*, I start with the root node, and then this is 0, this is the 1 and I add a node here. This is my P_3.

Every time some prefix is matching, what I am going to do is I am going to highlight that particular node with a color, maybe this is my P_3 indicating this prefix is matched, 1* is matched. So, that is why this is marked, and 0* is matched here. And that is why this is marked. Similarly, * is there, you do not have to read any IP address, it is identified by marking the root node itself. So, anything does not matter, what is the IP address, you should match that. And I continue doing this.

So, I take the other prefixes and then add the entries onto that. So, for example, the fourth prefix is saying 00* and I already have one prefix which is 0, and we come to the node P_2 to accommodate one more 0 I require, so I am going to have a left node with the 0 to this one. So, this is my P_4. And P_5 is saying 1 followed by 1, so I am going to add the right child to this node; this is my P_5, and P_6 is saying 110*. So, 11 is already there; I read it 0 and come to the left; this is my P_6.

And P_7 is saying 1111 four times two times already I have crossed here. So, another one, come here, and then another one, go to the right. So, again, I am going to mark these nodes with the appropriate prefixes. So, this is P_4; this is P_5, P_6. And then this is my P_7. So, remember that this node is actually not marked. So, wherever the prefix is matched, that node is actually marked. So, this is how the binary trie is actually constructed.

So, for the entire setup of the prefixes that you have in the routing table, you have only one such kind of a trie, here, the prefixes are having up to 4 bits. So, ideally, it might have 32 bits.

So, as the number of the bits in the prefixes increases, the height of the trie, you do also correspondingly increase. So, for example, if I have a new entry, which is saying something like this, so let us say P_8, it is saying that this is 000000 six times 0, what it means is extending this node by four times, 0 and second one, two 0s I already have read, first one, second, third one fourth one, fifth one, and then the sixth one, and I am going to mark this as the node corresponding to the prefix P_8.

So you can see this as the number of the bits or prefix length increases, the height of the trie is going to increase. So, the prefix can be at the max of 32 bits. So, corresponding to that, the binary trie can have a height of 32, so the 32 level is there. So, how does the lookup operation if I have such a trie constructed, and how is the lookup operation done.

So, let us say I receive a packet with the IP address, let us say 11010. Now I need to traverse this trie and then find out what is the best match for this particular IP address. So, the first bit is 1, you start with the root node. And then, when nothing is matching by default, this P_1 is anyway matching; that is the default entry.

So, since there is a one there is a child with the label 1, then I go here, and I remember P_3. So, far the best match that happened. And I go to the second bit and then say that either a child with the

label 1 to this corresponding node yes, there is. And then, I come back and then remember P5 is the so far best match that I found out.

And then I read, go and read the third bit of the destination IP address and again, continue the search, there is a node with the label 0, and I come to the node number P_6. P_6 is the prefix that is so far matched to the route and then I go and read the fourth bit which is 1 but unfortunately, I do not have a child node for P_6. So far whatever is the best match that is found is the P_6.

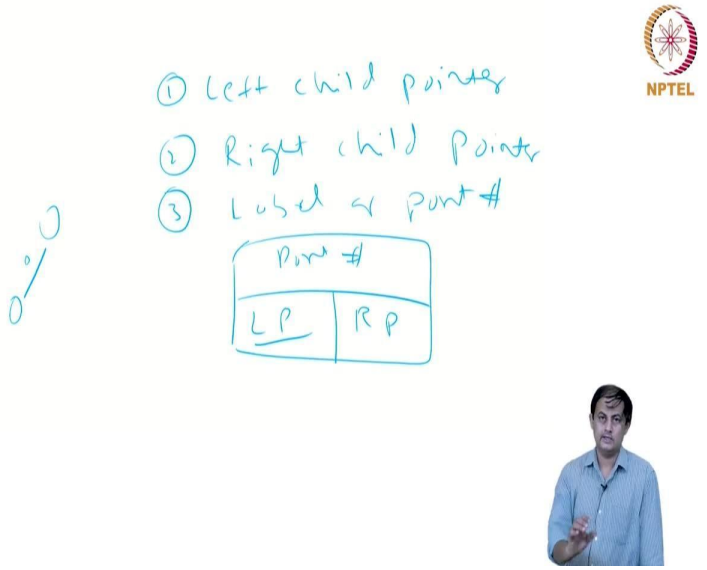
So, what it means is if your destination IP address is 11010 and if your set of prefixes are in this format, and the best match that you can find out is the P_6 which is 110*. So, 111 the other ones P_7 and P_8 are not matching, neither the P_1 and P_2 and P_3. So, this is the longest prefix that has matched this particular IP address.

Remember, P_1 has also matched, default route that is also matching, so using 1, P_3 is also matching, P_5 is also matching, and so is P6, but out of this P_6 is the one which is representing the longest prefix. So, the routing decisions need to be made by whatever the port number is indicated against the P_6 prefix. So, here in this case, you need to forward it to port number 2 which is the meaning of this.

So, that is how the binary tries are constructed, and then the routing table. So, you can think of this if I have a data structure like this, and I am saying that the height of this trie is going to be 32 in the worst case, the number of the lookup operations that you need to do is 32. So, this can be implemented using the linked list.

So, you just need to traverse this list up to a little bit, one bit at a time, you read and then find out whether there is a left child you need to take or the right child you need to visit, and you keep visiting wherever you hit the dead end and if that node is marked with the label, some prefix you actually could do the routing decision or the forwarding decision based on that particular node. So, that is simple, that is how the trie binary trie is actually constructed.

(Refer Slide Time: 20:45)



So, now we look a little bit deeper, I want to build this trie. What is there inside each of this node? So, I can think of this, a particular node inside this trie needs to have three things; one is it needs to have the pointer to the left child; left child pointer, and the second thing it needs to have is a pointer to the right child; the right child pointer and the third thing that it needs to have a label or the port number on which the packet needs to be forwarded.

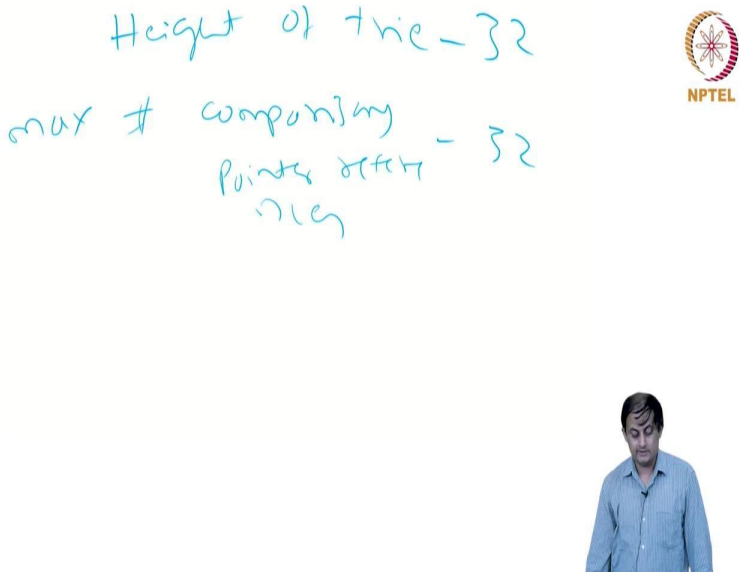
So, you can visualize something like this I have got two pointers; one is the left pointer one is the right pointer. And the upper proportion is actually telling me the port number on which the packet needs to be forwarded if it is a node with the particular label, not necessarily that it should always have.

So, for example, for any node which is not marked here in this case, this node is not marked, and this node is not marked and that node need not have this port number information only the nodes which are corresponding to representing some prefixes will have this port number information. So, P_2 will have you need to forward it to port number 1, P_4 will allow you need to forward it to port number 4, something like this.

And again, it is obvious to see that if we have both the child, you will have this left pointer set and right pointer set. If some node has only one child, either the left or right, then it will have the other child set to null. So, that is how the structure is going to work. So, you start with the root and keep searching, wherever you hit the boundary, whatever the prefix that you made there you stop it.

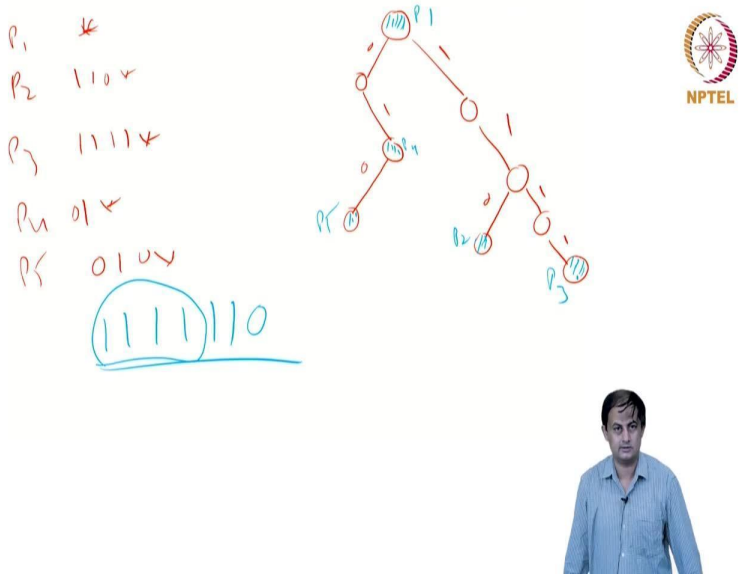
So, it might also happen that when you are doing the search, so for example, if my IP address is let us say, the destination IP address is 000. And so you start with the search the first 0 you read, you come to the P_2, and the second 0 you read to come to the P_4 in the read the third 0, you actually go to this node, but since this node is not marked, you do the routing decision based on the previous match that you found out. Here, in this case, the P_4 meaning that by using only the 00*, you are actually doing the routing decisions, which is the longest prefix that is matched for this IP address. So, that is how actually this data structure works.

(Refer Slide Time: 23:57)



So, I repeat some of the formal concepts, the height of trie is 32. And the maximum number of comparisons, in this case here is the pointer traversal, left or right. Pointer references are going to be 32 at every level; you are going to have one either you decide to go to the right or to the left of them and the node itself will have three components one is the left pointer, the right pointer, and the label or the output interface name. So, that is how the trie is actually constructed, and based on that, the routing is done.

(Refer Slide Time: 24:58)



So, maybe let us take one more example of constructing the binary trie. So, let us say the P_1 is again *, (in this case, I am going to ignore the interface number or the port number on this one just the prefix notation). So, P_2 is 110*, P_3 is 11* and P_4 is maybe 01* and P_5 is 010*, construct a binary trie for these set of prefixes. So, as usual, I start with the root node, P_1 is taken care of now, and then the P_2 is saying that 1111 and then 0 come to the left, and P_3 saying four times 1, the third one, and then the fourth one.

And P_5 is saying 0 followed by 1 so, we have not read the 0 so far read that and then followed by a 1 come to the right and the P_5 is saying 01 followed by a 0. So, you go to this label 0 followed by 1 followed by 0 and the labeling would be something like this. So, this is your default order P_1 and P_2 is 110. So, that is the node here this is your P_2 and P_3 is four times 1 i.e., 1111, this is the node which is corresponding to P_3, P_4 is saying 0 followed by 1, 0 followed by 1 is here this is your prefix P_4 and P_5 is 01 followed by 0 which is here.

So, there are 5 prefixes and there are five nodes with the corresponding label each one of them will have a reference to the port number. So, the other ones will not have as I said, so as usual if you receive an IP packet with the destination IP address 1111110. So, this is actually 32 bits. But for the sake of convenience, I have just written these 7 bits. So, how do you do the picking, read these four bits and it will be in the P_3 and that is the longest prefix that you can match.

So, based on whatever the P_3 is saying you need to route that particular packet to that particular interface. So, that is how you implement, you take all the prefixes. So, remember prefixes are

coming from the network component, how many bits are used for the networks, and you take those prefixes and then construct a binary trie like this. And this is a software implementation.

In the worst case, you need to do the 32-node traversals in this particular trie. So, this is one fault of this my implementation. But the question is, is it the best that we can do? So, the answer is no. There are some improvements that we can make to this binary trie. We will come back and see those optimizations in the next class. Thank you.