Politecnico di Milano

Dipartimento di Elettronica, Informazione e Bioingegneria

Switching and Routing

IP lookup - Bitmap algorithm

Grupo 4:
Nuno Bajanca
Vladimir Tomov
Konstantin Tsolev
Francisco Santos
Alim Mohamed

Professors:
Guido Maier
Navin Kukreja

February 19, 2016

Contents

List of Figures

List of Tables

1. Introduction

This course project is a team assignment to design network of multiple switches, each with a different subnet mask, and implement Binary Trie and Bitmap IP lookup algorithms.

To take conclusions on the performance of these two algorithms the performance metrics are going to be discussed and the Lookup speed is going to be tested.

In this documentation file we will start by providing a basic theoretical introduction to the components, tools and algorithms that are being used or developed in this project.

We will then explain the decisions relevant for the implementation of this project.

To end we will analyse the results obtained and conclude on them.

In the Annex section there is a Tutorial on how to run this project.

2. Theoretical Basis

This chapter is dedicated to the introduction of the Theoretical principles behind the algorithms and to the introduction of the tools used in this project.

2.1 SDN

SDN (Software Defined Networking) is a network architecture that breaks the vertical integration by separating the control logic (control plane) from the underlying routers and switches that will become simple forwarding elements (data plane). The main pillars of SDN are:

1. The control and data planes are decoupled;
2. Forwarding decisions are flow-based instead of destination-based;
3. Control logic is moved to an external entity, called SDN controller or NOS (Network Operating System);
4. Network is programmable through applications running on top of the controller (the fundamental characteristic of SDN).

The SDN Architecure is represented in the figure 1. In order to put all the theory into practice, we will use Mininet, that is explained in the following section.

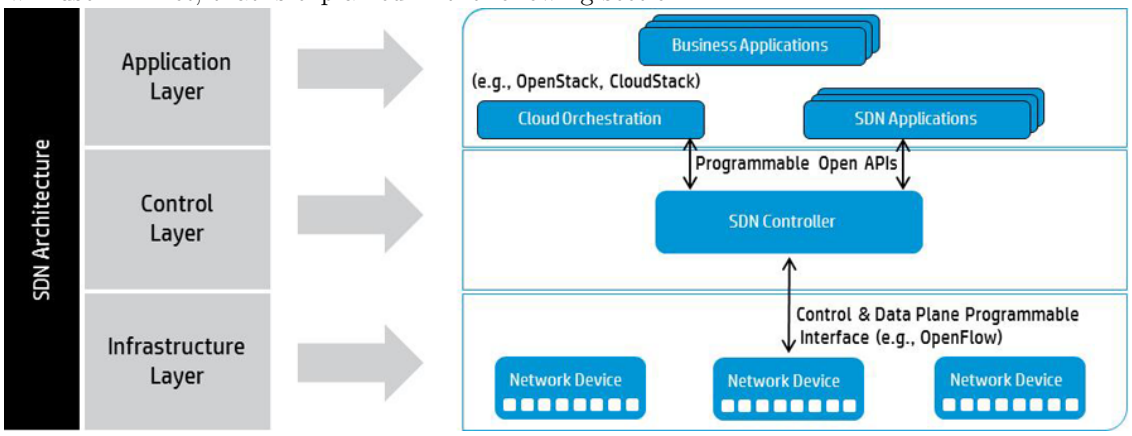

Figure 1: SDN Architecture [source: http://networkstatic.net/hp-announces-their-sdn-controller/]

2.2 Mininet

Mininet is a virtual network environment that runs on a single machine and provides many of the OpenFlow features built-in. Mininet will emulate an entire network of switches, virtual hosts (running standard Linux software), the links between them and, optionally, a SDN controller.

Mininet supports research, development, learning, prototyping, testing, debugging, and any other tasks that could benefit from having a complete experimental network on a laptop or other PC.

2.2.1 Why Mininet?

Among the multiple advantages that provides Mininet,some of the keys points that motivated its use for our project are:

- Mininet provides fast network setup, easy to use, custom topologies;
- Mininet can run real Linux programs;
- Mininet switches support OpenFlow;
- Mininet is Open source, under active development.

2.3 OpenFlow

OpenFlow (OF) is considered one of the first software-defined networking (SDN) standards. It originally defined the communication protocol in SDN environments that enables the SDN Controller to directly interact with the forwarding plane of network devices such as switches and routers. The arquitecture of a switch network is demonstrated in figure 2.

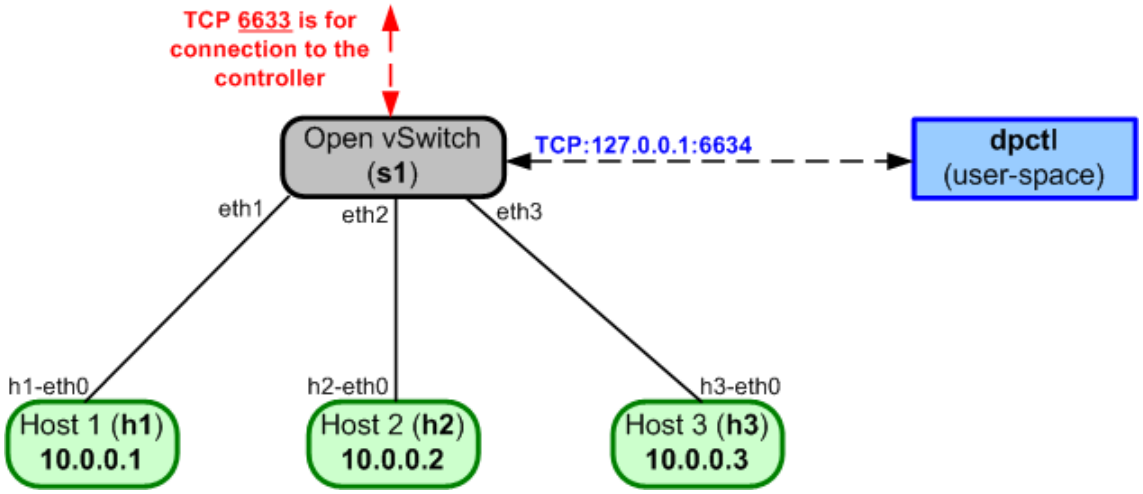

Figure 2: OpenFlow switch network arquitecture [source: http://www.costiser.ro]

Most OpenFlow switches have a passive listening port running on TCP 6633 (by default) that can be used to poll flows and counters or to manually insert flow entries.

The open flow hardware consists of the following:

- A flow table, represented in figure 3, containing flow entries consisting of match rules and actions that take on active flows;
- A transport layer protocol that securely communicates with a controller about new entries that are not currently in the flow table.

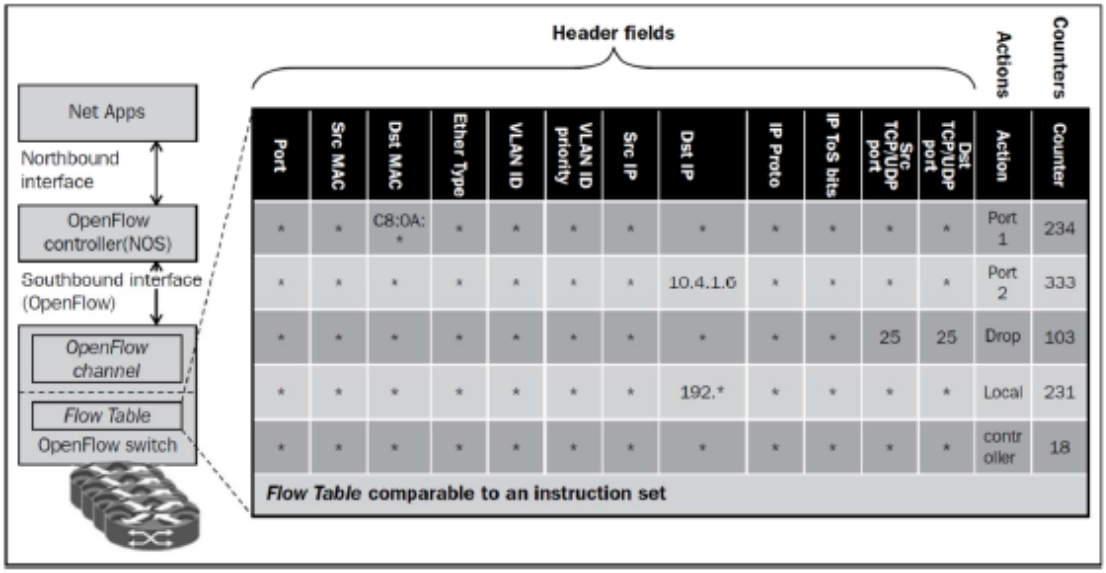

Figure 3: OpenFlow flow table

2.4 Ryu Controller

Ryu Controller, is an open software-defined networking (SDN) Controller designed to increase the agility of the network by making it easy to manage and adapt how traffic is handled. In general, the SDN Controller is the "brains" of the SDN environment, communicating information "down" to the switches and routers with southbound APIs, and "up" to the applications and business logic with northbound APIs. The Ryu Controller is supported by NTT and is deployed in NTT cloud data centers as well.

The Ryu Controller provides software components, with well-defined application program interfaces (APIs), that make it easy for developers to create new network management and control applications. This component approach helps organizations customize deployments to meet their specific needs; developers can quickly and easily modify existing components or implement their own to ensure the underlying network can meet the changing demands of their applications.

2.4.1 How a Ryu Controller Fits in SDN environments

Written entirely in Python, all of Ryu's code is available under the Apache 2.0 license and open for anyone to use. The Ryu Controller supports NETCONF and OF-config network management protocols, as well as OpenFlow, which is one of the first and most widely deployed SDN communications standards.

The Ryu Controller can use OpenFlow to interact with the forwarding plane (switches and routers) to modify how the network will handle traffic flows, has showned in the figure 4. It has been tested and certified to work with a number of OpenFlow switches.

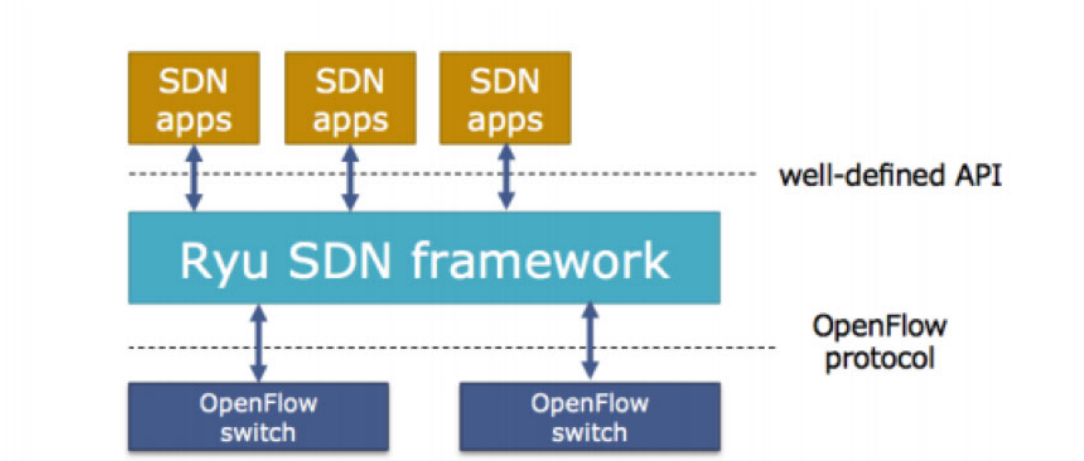

Figure 4: Ryu in SDN [source: https://www.sdxcentral.com]

2.5 IP lookup

2.5.1 Performance metrics

During the analysis of the algorithms we will focus our attention to lookup speed, storage requirement and update time. It's important to refer that Scalability and Flexibility are important performance metrics, but not relevant in this case to compare bitmap and binary trie algorithms.

Lookup speed

The lookup speed is becoming more important every time advancements in the link bandwidth are made. The algorithms that are analysed in this project are slow when compared with hardware implementations, and for that reason they aren't usually used in backbone routers.

Storage requirement

In order to have fast memory access speed and low power consumption we need to assure small storage. For cache-based software algorithm this is very relevant.

Update time (of lookup tables)

Nowadays the number of updates of lookup tables is increasing and they shouldn't interfere with normal lookup operations.

Scalability

The size of the forwarding tables is increasing at a speed of approximately 25k entries per year so the algorithms should be able to support large forwarding tables.

Flexibility in implementation

Most of the algorithms can be implemented in different ways, and either software or hardware. These flexibility is important.

2.5.2 Binary Trie algorithm

The Binary trie algorithm is based on binary tree structure. In this structure every node contains next hop information plus pointer to child nodes if there are any. If there is a default switch its information would be in the root node.

Each node has two arcs:

- "1" corresponds to the right arc of the tree;
- "0" corresponds to the left arc of the tree.

Each arc is terminated by node:

- If the node corresponds to a Prefix entry is marked as grey node;
- If not it is marked as a white node.

White nodes must have at least one child rather grey node can be without any.

Route lookup is done by simply search for the longest prefix match by examining bit by bit of the ip address and matching it upon the tree. If longest prefix match is found algorithm returns the corresponding prefix, to that match. Otherwise binary trie algorithm performs backtrack to the last known prefix that was matched, which will be the longest match.

Performance

- Ip lookup speed, in the worst case, would be 32 memory accesses, O(W);
- The worst update would translate in adding all the nodes so also 32, O(W);
- Storing complexity would be, in the worst case, 32N*S, O(NW).

Where N is the number of prefixes, W the lenght of the prefixes (32 bits is maximum for ipv4) and S the memory space required for each node.

2.5.3 Bitmap Tree algorithm

The tree bitmap algorithm is based on multibit tree and it has bitmap compression to avoid wasting memory in the tree and achieve fast lookup.

In the bitmap tree each node has a fixed size containing a head pointer to the block of the child nodes, the extended bitmap for them, a head pointer to the next hop information and the extended bitmap for it. The update time is bounded by the size of the trie node and because of that the can't be bigger than 8 otherwise the update would have a compromising cost. One of the key points of tree bitmap is the use of memory so each node is as small as possible in order to occupy the minimum memory possible. The child nodes of each node are stored contiguously in memory, there is only need to store one pointer and with help of the bitmap is possible to access to all of them. For the information of the next hop associated with the internal prefixes each node stores a head pointer and a bitmap for an array that stores that information and the array is associated to that node. The first bit of the bitmap is associated with the length 0, the second with the length 1 and so on.

To do the longest prefix match on the tree and according to the selected stride (consider n) chosen we start to take the first n+1 bits of the prefix we want to search. Starting in the root node we search in the child bitmap, go to index correspondent to the bits we have and see if there is a '1', that means that this node has a child. If there is a '1' we need to see how many '1' there are to the left to have the offset and then compute calculate the pointer to that child. Next, we check the internal bitmap to see the longest prefix. In first place we remove the rightmost bit and then go to the position that the bits we have corresponds and check to see if there is a '1 or not. If there is a '1' it means that there is a prefix. We count the number of '1' to the left to see the offset and then calculate the pointer and store it as prefix match. If there was a '0' we just remove another bit and check it again, this is done until there are no more bits left. When the search in the internal bitmap is done we go the child node, and now we are taking the next n bits of the

prefix we are searching and then do whole process again. We will keep doing this until there is no more children in the node we are or there are no more bits on the on prefix we have and then the search ends.

Performance

- Ip lookup speed, in the worst case, would be 32 memory accesses divided by the stride, O(W/k);
- The update time is relatively fast but bounded by the size of the trie node, so no more than eight strides are used;
- Storing complexity is good because both multi-bit tries and bitmap allow to reduce the wasted storage.

Where W the lenght of the prefixes (32 bits is maximum for ipv4) and k the number of strides. This algorithm has a tradeoff between complexity and memory access, where the complexity is prefered is to the memory accesses.

2.5.4 Comparison between algorithms

Based on the theoretical introduction done in this section and general observations it is possible to conclude the following:

- Bitmap algorithm uses fast search when compared to the binary trie;
- Binary trie consumes more memory than bitmap.

Besides that bitmap has Scalability in terms of lookup speed and table size, the capability of high-speed updates, and feasibility in size to fit in a Level 3 forwarding engine or packet processor with low overhead.

During this project both algorithms will be implemented and the lookup speeds will be tested and compared.

3. Implementation

3.1 Topology and configuration

Our choice for topologies is listed in the table 1.

Table 1: Topologies and distribution of ips

TOPOLOGY IP DISTRIBUTION DETAILS					
TOPOLOGY TYPE	NUMBER OF HOSTS	FIRST HOST IP	LAST HOST IP	FIRST HOST NAME	LAST HOST NAME
TOPOLOGY_4_SWITCHES	16 HOSTS	195.0.0.1/8	195.0.0.4/8	H1	H4
		192.168.0.1/24	192.168.0.4/24	H5	H8
		154.128.0.1/16	154.128.0.4/16	H9	H12
		197.160.0.1/24	197.160.0.4/24	H13	H16
TOPOLOGY_8_SWITCHES	32 HOSTS	170.0.0.1/8	170.0.0.1/8	H17	H20
		160.110.0.1/24	160.110.0.4/24	H21	H24
		130.128.0.1/16	130.128.0.4/16	H25	H28
		110.10.0.1/24	110.10.0.4/24	H29	H32
TOPOLOGY_12_SWITCHES	48 HOSTS	166.120.10.1/8	166.120.10.4/8	H33	H36
		135.110.20.1/24	135.110.20.4/24	H37	H40
		100.160.30.1/16	100.160.30.4/16	H41	H44
		10.0.40.1/24	10.0.40.4/24	H45	H48
TOPOLOGY_16_SWITCHES	64 HOSTS	192.168.10.1/8	192.168.10.4/8	H49	H52
		192.168.20.1/24	192.168.20.4/24	H53	H56
		192.168.30.1/16	192.168.30.4/16	H57	H60
		192.168.40.1/24	192.168.40.4/24	H61	H64

There are allways 4 hosts per each switch, because the number of hosts in each switch is not relevant for the IP Lookup process.

The respectively distributed IPs are listed in the middle columns. In each topology 4 switches were introduced, two with a subnet of mask 255.255.255.0, one with a subnet mask of 255.255.0.0 and one with a subnet mask of 255.0.0.0. This allows the switches to be well distributed inside the trees.

Every host has a unique id and it is connected to the responding switch.

To apply these configurations to the project there are 4 different config and topology files. The information provided in them will be explained in the following lines.

3.1.1 Configuration file

The config file is usually associated with the topology file.It is the file which defines the ip adresses and default gateway distribution associated to a particular topology.

3.1.2 Topology file

The topology file defines the physical structure of the network.It defines the number of switches,hosts and interconnections between them.

3.2 Lookup algorithms

In this section the implementation of the tree bitmap and binary trie algorithms are going to be explained. The UML for this implementation is the one in figure 5.

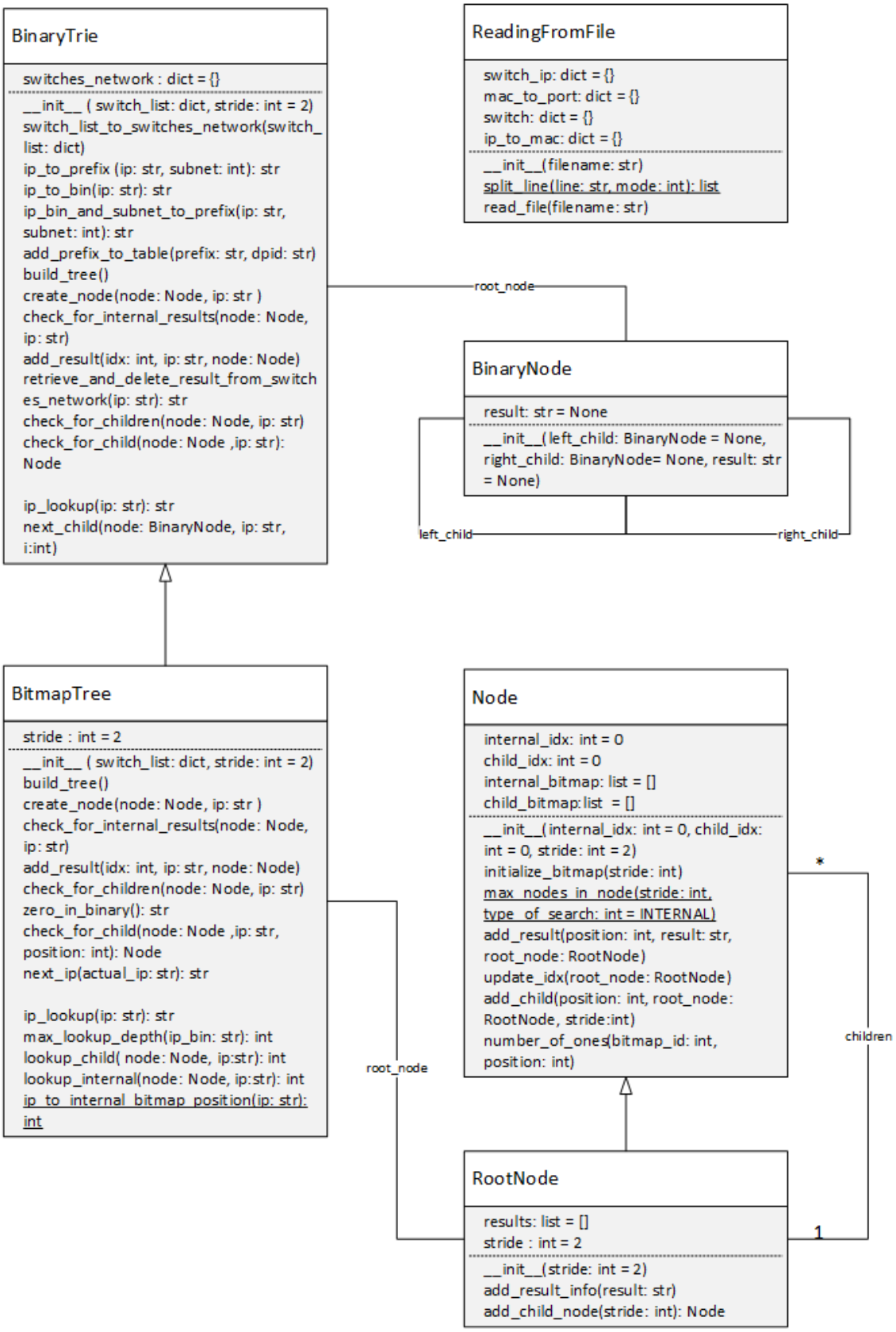
BinaryTrie
switches_network : dict = {}
__init__ (switch_list: dict, stride: int = 2)
switch_list_to_switches_network(switch_list: dict)
ip_to_prefix (ip: str, subnet: int): str
ip_to_bin(ip: str): str
ip_bin_and_subnet_to_prefix(ip: str, subnet: int): str
add_prefix_to_table(prefix: str, dpid: str)
build_tree()
create_node(node: Node, ip: str)
check_for_internal_results(node: Node, ip: str)
add_result(idx: int, ip: str, node: Node)
retrieve_and_delete_result_from_switches_network(ip: str): str
check_for_children(node: Node, ip: str)
check_for_child(node: Node ,ip: str): Node
ip_lookup(ip: str): str
next_child(node: BinaryNode, ip: str, i:int)
ReadingFromFile
switch_ip: dict = {}
mac_to_port: dict = {}
switch: dict = {}
ip_to_mac: dict = {}
__init__(filename: str)
split_line(line: str, mode: int): list
read_file(filename: str)
root_node
BinaryNode
result: str = None
__init__(left_child: BinaryNode = None, right_child: BinaryNode= None, result: str = None)
left_child
right_child
BitmapTree
stride : int = 2
__init__ (switch_list: dict, stride: int = 2)
build_tree()
create_node(node: Node, ip: str)
check_for_internal_results(node: Node, ip: str)
add_result(idx: int, ip: str, node: Node)
check_for_children(node: Node, ip: str)
zero_in_binary(): str
check_for_child(node: Node ,ip: str, position: int): Node
next_ip(actual_ip: str): str
ip_lookup(ip: str): str
max_lookup_depth(ip_bin: str): int
lookup_child(node: Node, ip:str): int
lookup_internal(node: Node, ip:str): int
ip_to_internal_bitmap_position(ip: str): int
Node
internal_idx: int = 0
child_idx: int = 0
internal_bitmap: list = []
child_bitmap:list = []
__init__(internal_idx: int = 0, child_idx: int = 0, stride: int = 2)
initialize_bitmap(stride: int)
max_nodes_in_node(stride: int, type_of_search: int = INTERNAL)
add_result(position: int, result: str, root_node: RootNode)
update_idx(root_node: RootNode)
add_child(position: int, root_node: RootNode, stride:int)
number_of_ones(bitmap_id: int, position: int)
*
children
root_node
RootNode
results: list = []
stride : int = 2
__init__(stride: int = 2)
add_result_info(result: str)
add_child_node(stride: int): Node
1

Figure 5: UML

3.2.1 Nodes

The nodes are represented through classes.

In the Binary Trie all the nodes are equal, and represented by the class BinaryNode. This class has three references and a result. Being a binary trie we have the left and the right children, but as it is used for lookup.

In the Bitmap Tree the root node is different from the rest of the nodes, so there are used to classes for the nodes:

- Node - Has two bitmaps, the internal and the children' one;
- Root Node - Subclass of Node, besides being a node it also has two lists, the results and the children.

This is done because python doesn't have pointers so we couldn't alloc memory for the children of a node and get the first memory position, instead we use a list (works like a vector) and the first memory position, present in every node, is just the index of the first object.

3.2.2 Tree Creation

Both algorithms have the same fluxogram, represented in figure 6, for the construction of the tree so the Bitmap Tree is a subclass of the Binary Trie. This assures that all the methods done in the Binary Trie are either used or overridden by the Bitmap Tree.

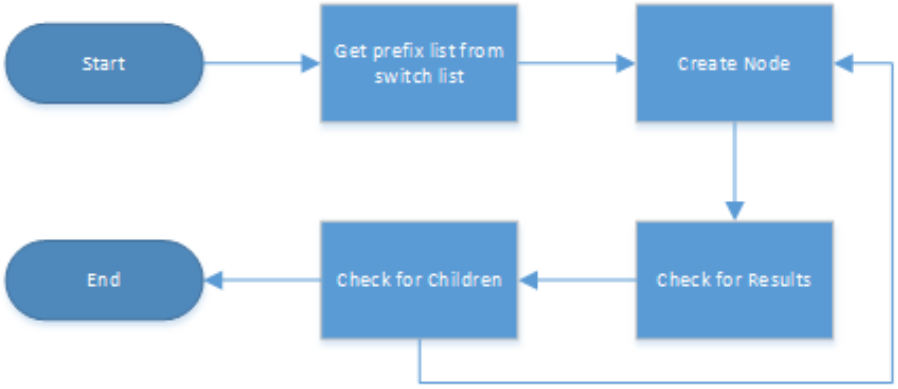

Figure 6: Fluxogram for the Tree Creation

The construction of the trees starts with the conversion of the switch list to a prefix table in the method switch_list_to_switches_network(). This method converts the switch ips to binary and then trims them using the prefix information (mask).

Next, the root_node is created. For both algorithms this is a simple call to the constructor of the respective class. From now on the construction of the trees differs so we will explain them in separate sub-chapters.

Binary Trie

The creation of the nodes in the binary trie is done in a recursive way starting by the left nodes, this means that the first node to be constructed is "0" (if there is any prefix that starts with it, obviously) and then "00" and so on until there is no prefix correspondence. When that happens the algorithm starts returning and constructing the right nodes until all the prefixes are represented in the nodes.

The sequence of methods called in the code are the following:

1. create_node() : A method that will them call:
   (a) check_for_results() : If an exact match between the ip of the node and one switch in the list is found then that result is added to the node;

(b) check_for_children(): Checks for a child in the left node, if it finds one it will call create_node() for it. If it doesn't or if the create_node() already returned then it will check for the right child, applying the same principle.

In the print screen presented in the figure 7 we can see that from the node "0110" we build all the nodes to the left ("01100" and its children) and only after we return to the "0110" and we start to build to the right ("01101" and its children).

```
--- Building Node (01)
----- Adding Child
--- Building Node (011)
----- Adding Child
--- Building Node (0110)
----- Adding Child
--- Building Node (01100)
----- Adding Child
--- Building Node (011001)
----- Adding Child
--- Building Node (0110010)
----- Adding Child
--- Building Node (01100100)
----- Adding Child
--- Building Node (011001001)
----- Adding Child
--- Building Node (0110010010)
----- Adding Child
--- Building Node (01100100101)
----- Adding Child
--- Building Node (011001001010)
----- Adding Child
--- Building Node (0110010010100)
----- Adding Child
--- Building Node (01100100101000)
----- Adding Child
--- Building Node (011001001010000)
----- Adding Child
--- Building Node (0110010010100000)
----- Adding Result
----- Adding Child
--- Building Node (01101)
----- Adding Child
--- Building Node (011011)
----- Adding Child
--- Building Node (0110111)
```

Figure 7: Print screen from controller building the binary trie for the configuration with 16 switches

Bitmap Tree

The same logic of the Binary Trie is applied but in this case the implementation is more complex and requires the help of the following methods:

- zero_in_binary() : This method returns the zero in binary according to the stride so, for example, if stride is 2 this method returns "000", that corresponds to the first child associated with this node;
- next_ip() :This method returns the next_ip and works for both type of searches:
  - For internal result search the "binary tree"(There isn't any binary tree, but the works as there is) in snake like movement ("”-¿"0"-¿"1"-¿"00"-¿"11"-¿..);
  - For child search the "binary tree" is searched only on one level (for ex: from "000" to "111")

In the create_node() method we start by searching for results in the check_for_internal_results() method, which used next_ip() to go true all the possible correspondences in the prefix table. If an "exact match" is found a "1" is putted in the correspondent position in the internal bitmap

and the correspondent is deleted from the prefix table and added to the result list (the results are appended in sequence so the index in the node corresponds to the first result in the node in question).

After the internal search we start searching for children. When a child is found it is added to the list and a "1" is putted in the bitmap. To assure the index stays correct we search all children from a certain node and only after finishing the search we do a second loop (only through the children found) where we enter the recursive, calling the create_node() method for that child.

In the print screen presented in the figure 8 we can see that for the Root Node we found several results (as the stride is 8 we can conclude that the switches 9, 5, 13 and 1 have a 255.0.0.0 mask) and only after that we search for children. After finding all the children we went to the first child and repeated the process, not finding any result but finding at least one result.

```
-- Building Bitmap Tree (stride = 8) with 16 Switches
--- Creating Root Node
--- Building Node
---- Building Internal Bitmap
----- Adding Result
------ [421] : 9
----- Adding Result
------ [425] : 5
----- Adding Result
------ [447] : 13
----- Adding Result
------ [450] : 1
---- Finish Internal Bitmap
---- Building Child Bitmap
----- Adding Child
------ [20] : 000010100
----- Adding Child
------ [201] : 011001001
----- Adding Child
------ [220] : 011011100
----- Adding Child
------ [261] : 100000101
----- Adding Child
------ [270] : 100001110
----- Adding Child
------ [309] : 100110101
----- Adding Child
------ [320] : 101000000
----- Adding Child
------ [385] : 110000001
----- Adding Child
------ [395] : 110001011
---- Finish Child Bitmap
--- Building Node
---- Building Internal Bitmap
---- Finish Internal Bitmap
---- Building Child Bitmap
----- Adding Child
------ [0] : 000010100000000000
```

Figure 8: Print screen from controller building the Bitmap Tree for the configuration with 16 switches

It's important to refer that the prefix table gets its entries deleted when they are added to the tree. This is a responsibility of the method retrieve_and_delete_result_from_switches_network() which is shared by class and subclass.

3.2.3 Lookup implementation

The lookup in both algorithms is implemented precisely as described in the theoretical section of this documentation.

Binary Trie

For the binary trie the ip_lookup we need backtrack so a recursive function was implemented for the search:

1. Starting from the root node the trie is crossed until the last node in the path of the ip received is reached;
2. The recursive method returns:
   (a) The next-hop information present in its structure (None if it's a gray node);
   (b) The next-hop information returned by the child node, if it is not None.

In this way when the end of the recursive is reached we will have the longest match switch information.

Bitmap Tree

In the bitmap case no backtracking is needed so we a loop. The loop is between the following methods:

1. lookup_child(): In this method we look for the next node, if it exists, the position(index in the bitmap plus the number of ones) is saved;
2. lookup_internal(): In this method we look for the longest match prefix in the bitmap, this makes this method to be also a loop. When, and if, a match is found the position is saved.

If a child is found the same procedure is done in it, if it isn't than the last result obtained is returned.

3.3 Reading Configuration File

Instead of having to do manually the creation of the topology of the network we read the config file and generate the topology. In the reading of the file we take all the information about the switches we need (IP, Mask, id) and then associate a sequential MAC address to each one and create the switch_ip list and the switch list. We also read the ip of the hosts and to which switch they are connected. We also assign sequentially a MAC address to each one and then create the ip_to_mac list and mac_to_port list.

In order to do this we basically use to methods:

- split_line(): This method receives the lines from the config file and returns a list with the important values that we need;
- read_file(): This method first stores the information in the config file to a general buffer and then puts it in the specific dictionaries that we want assigning sequentially the MAC addresses needed;

3.4 Timers

3.4.1 Timers in the controler

Timers were implemented in the code to obtain the following results:

- The lookup time;
- The percentage of time needed to respond to an incoming packet that is spent in the lookup algorithm.

Having this is consideration there are two timers:

1. Starting when a packet is received, finished when the packet is forward. This timer is ignored if the packet doesn't require ip lookup, for example arp requests;
2. Starting when the ip lookup starts and finishing when the next-op information is obtained.

For the correct analysis of the performance both algorithms are performed in the same response. We will then obtain the following values, for each request:

- processing
- lookup_bitmap
- lookup_trie

And we compute the following values:

- processing_bitmap = processing - lookup_trie
- processing_trie = processing - lookup_bitmap
- lookup_percentage_bitmap = (lookup_bitmap/processing_bitmap)
- lookup_percentage_trie = (lookup_trie/processing_trie)

3.4.2 Timers average via Rest API

To make the Lookup time performance metric easier to evaluate the results are presented through the RYU Rest API. For these the method list_timers() was added to the class Lookup-Controller with the route '/v1.0/lookup/timers'.

In this class, beside having the value of the timers, it was also added some code to make averages and changing the units, so a response to a GET request is:

- average_lookup_bitmap (ms)
- average_lookup_trie (ms)
- average_processing_bitmap (ms)
- average_processing_trie (ms)
- average_lookup_percentage_bitmap (%)
- average_lookup_percentage_trie (%)
- lookup_bitmap (table - values in s)
- lookup_trie (table - values in s)

4. Results

We derived several conclusions from the timers implemented, starting from the stride until the comparison between the two algorithms. This results were split in different sections, presented in this chapter.

4.1 Stride

From the results in the table 2 we can understand that the lookup decreases with the stride. This proves the lookup performance assumption that it is O(W/k) being k the stride and W the number of bits.

We chose to use stride of 8 for the rest of the tests but it is relevant to understand that we are not accessing the update time, which becomes worse with bigger strides.

Table 2: Lookup time for different stride values in the 4 switches configuration.

# switches	stride	Lookup (ms)	Processing (ms)	Lookup Percentage (%)
4	2	0.3016	1.4076	21.4870
4	3	0.3007	1.4832	21.6354
4	4	0.2875	1.6085	19.3794
4	6	0.2710	1.5359	19.0111
4	8	0.2441	1.4749	17.9382

4.2 Network sizes

The table 3 shows the difference between in the lookup time in different topologies. This results goes against what was supposed because there is a clear deterioration of the lookup time with the number of switches. As explained before the number of switches should only influence the Storage but in according to the results they influence the results. One explanation for this is related to the fact that the bitmap representation used is a list and so the time spent in counting the number of ones every time a result or a child is found its significant.

Table 3: Lookup time for different switches configuration.

# switches	stride	Lookup (ms)	Processing (ms)	Lookup Percentages (%)
4	8	0.2441	1.4749	17.9382
8	8	0.4567	2.1263	18.9730
12	8	0.5023	2.4325	18.8732
16	8	0.5260	2.7361	19.9275

4.3 Binary vs Bitmap

Once again there are discrepancies in the results because the bitmap algorithm should be faster than the binary one.

For these results, presented in table 4, the explanation is related to the way the speed of the algorithm was evaluated. The speed is evaluated through memory accesses, and in fact bitmap has less accesses to memory, but it doesn't take into account the tradeoff introduced by bitmap that is the complexity.

Table 4: Lookup time for Bitmap and Binary

	Bitmap		Binary	
# switches	Lookup (ms)	Lookup Percentages (%)	Lookup (ms)	Lookup Percentages (%)
16	0.3471	16.0834	0.2589	11.2175

5. Conclusions

In a conclusion manner, we were able to complete the objectives proposed:

- Design of 4 network topologies, each one with a different number of switches and a different subnet mask;
- Implement the two lookup algorithms, Binary Trie and Bitmap Tree;
- Measure performance parameters on these algorithms.

Unfortunately the tools used, and the implementation done didn't allowed the results to reach the theoretical conclusions, but these discrepancies were explained in the result analyze section.

We can then conclude that for mininet running on a linux virtual machine the complex lookup computations of bitmap don't compensate the reduction in memory accesses.

We consider that this project was very important for the better understanding of SDN and to learn how to work with new tools, as mininet and RYU, that we will possible use in the future.

6. Bibliography

1. H.J. Chao, B. Liu, High Performance Switches and Routers, John Wiley & Sons, 2007;
2. Theoretical classes slides;
3. Mininet - http://mininet.org/;
4. OpenFlow - https://www.opennetworking.org/sdn-resources/openflow;
5. RYU SDN Framework - http://osrg.github.io/ryu/.