SDN Controller

Team : **Radiance**

Team members **:**

1. 140110047 Shachi Deshpande
2. 140050002 Deep Modh
3. 140050007 Neeladrishekhar Kanjilal
4. 140050087 Ashna Gaur

Index

2. Abstraction

Input :

1. SDN controller takes traffic request inputs from the application interface.
   This is in the form of N*N*k*m matrix, where
   N is number of nodes controlled by our controller,
   k is number of service levels,
   m is number of instances of a particular service for given pair of source and destination.
2. The traffic request matrix with specification of protocols and priorities comes from the application interface via the northbound interface.
3. The N*N connectivity matrix input will be taken as vectors from the network interface at the interval of every 20 clock cycles.
4. The values taken for various variables are:
   m=10
   k=1

The controller is scalable to larger values but in test benches it takes value as 7
N is represented by 3 bit binary and “000” is reserved for ‘-1’. Hence 7 nodes can be represented

A network graph (adjacency matrix) means the following in our context-

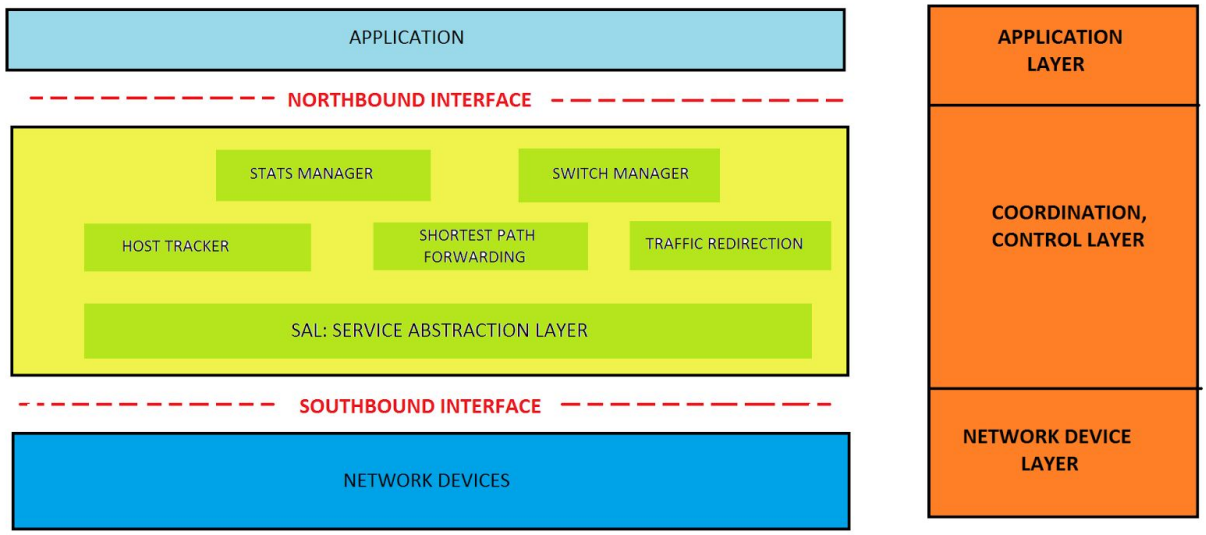
1. The nodes in network are vertices of graph.
2. If direct connectivity exists between 2 nodes in the network, then there is an edge between corresponding edges in the graph.
3. So, when path (i.e. continuous sequence of edges) exists between 2 nodes of a network, we can say that these nodes are connected in the graph.

Functionality :

Send maximum number of messages successfully.

3. High Level Architecture

3.a) High Level Block Diagram :



Functionalities:

Stats Manager: Maintaining statistics
Switch Manager: Manages switches connections with input and output ports
Host Tracker: Information related to messages of that host
Shortest Path Forwarding: The main algorithm
Traffic Redirection: Implements work conservation & priority management of requested services
SAL: A crossbar connecting the protocol plugin to the network function module

3.b) High-Level Idea of Working

1. **Computing shortest path** :
   We get the N*N connectivity matrix, and we divide the network into some predetermined number and sizes of sub-graphs.
   After every 20 clock-cycles, we will get the input of this connectivity matrix.
   Every time few new connections (or edges) are made and/or few edges are lost. Accordingly our graph changes.
   So, we compute the shortest path in the sub-graphs at the end of receipt of connectivity matrix every time. We keep the shortest path between every pair of nodes updated in this way.
   We plan to implement the algorithm in **higher level languages like python** and integrate it with our project. Otherwise we will implement it in VHDL itself.

2. After 200 clock-cycles we get the traffic matrix. For every pair of nodes, we **create a routing string** using the pre-computed shortest paths, and associate this string along with every message when we decide the next-hop router for a message. This ensures that we are routing the message along shortest possible path. This acts as **forwarding table**(We will can it **NH Matrix**(Next Hop)) entry for the particular packet, for the concerned router in the routing string.

3. In the initial stage of our project we will keep k=1, i.e. we have only one service, and thus no different relative priorities. In later stage of project k is made greater than 1, we basically have for each router a priority sensitive forwarding.
   Using the routing string we have already determined the sequence of nodes to be followed. Now, we need to determine which information packet is forwarded initially for given node pair. This is done by accepting the incoming packets into different section of router-related memory. So, during forwarding, we give higher priority of forwarding to the corresponding memory section storing those packets.
   This functioning would be similar to **priority arbiter**, who gives more time to process of higher priority.

4. **Protection Path**- It might happen that during forwarding particular edge of graph vanishes (i.e. direct connection between a pair of points is lost). In such case, we wait

for the acknowledgement packet from receiver. If there is such a problem, the message packet fails to reach the receiver, and hence after a predetermined waiting time, we compute the shortest path between the corresponding source destination pair once again. Since it would automatically not contain the vanished edges, we can be sure that this new path, called protection path, is completely edge disjoint from the failed link, and can successfully transfer the packet.

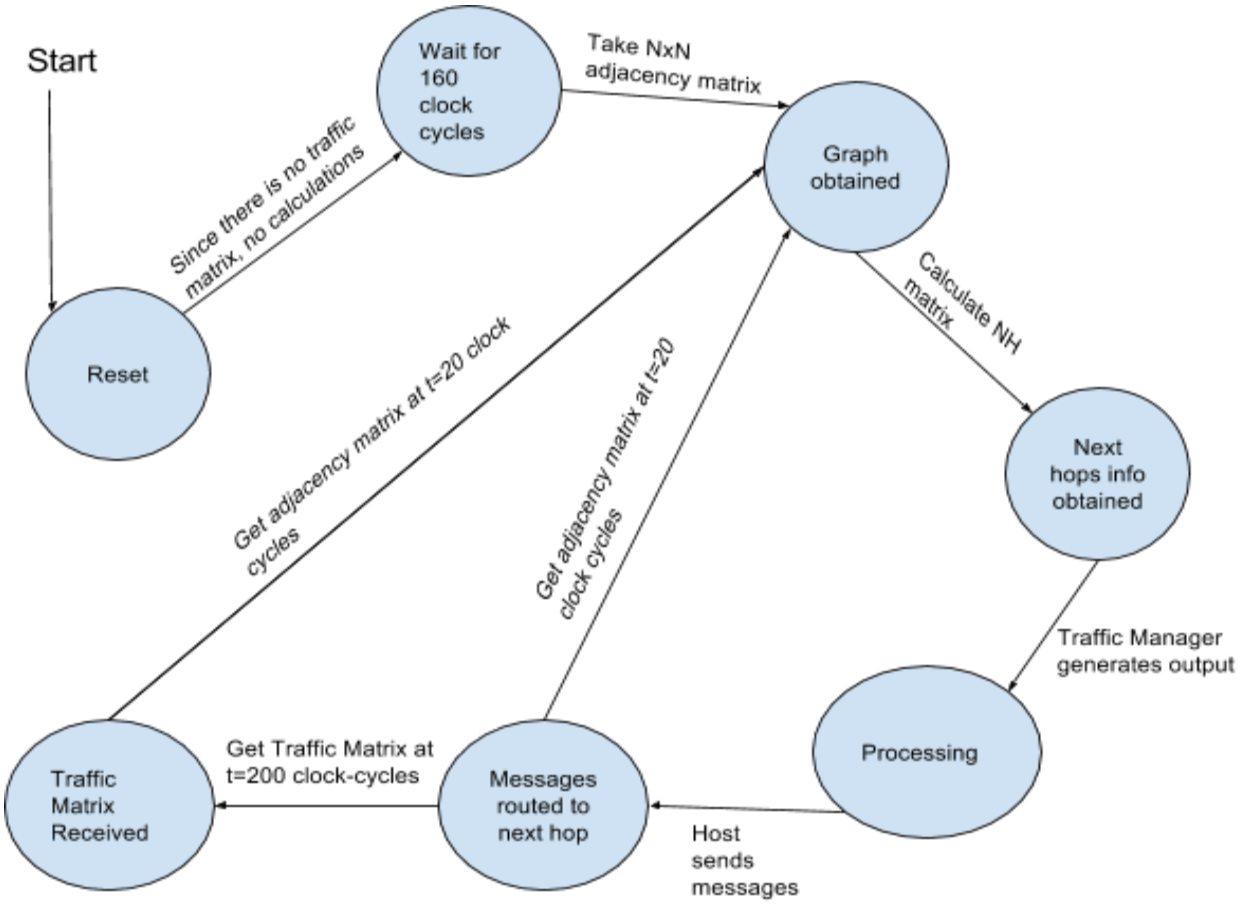
5. **Statistics Collection-** The application interface has an output corresponding to network statistics obtained (jitter, average latency for each path, and packet drop rate). For this we implement a statistics module in our code.

    a) We collect the difference between received time and sent time for each packet delivered for respective source destination pair and maintain this record to get **average latency**. This difference can be achieved by means of a stopwatch module inside statistics module.

    b) Also, the same information can be useful to calculate network **jitter** as variance of latency of packet delivery between each source destination pair.

    c) **Packet drops** for each source destination pair are maintained in 2 ways- we increment total packet drops when every time a router is full and message packet forwarded to that particular router gets dropped. We also increment packet drop number when suddenly a connection link fails (as described in point 4) and the packet gets lost. The answers to average latency, jitter and and packet drops appears as output of this statistics module.

3.c) State Machine Diagram



This presents a high level state machine diagram, which defines the functioning of the blocks with respect to current state.

Explanation :

1. In the 1st 200 cycles, since we don't have a traffic matrix, we won't do any forwarding or act upon the adjacency matrices.

2. At t=160 clocks, we begin processing adjacency matrices for the 1st time.

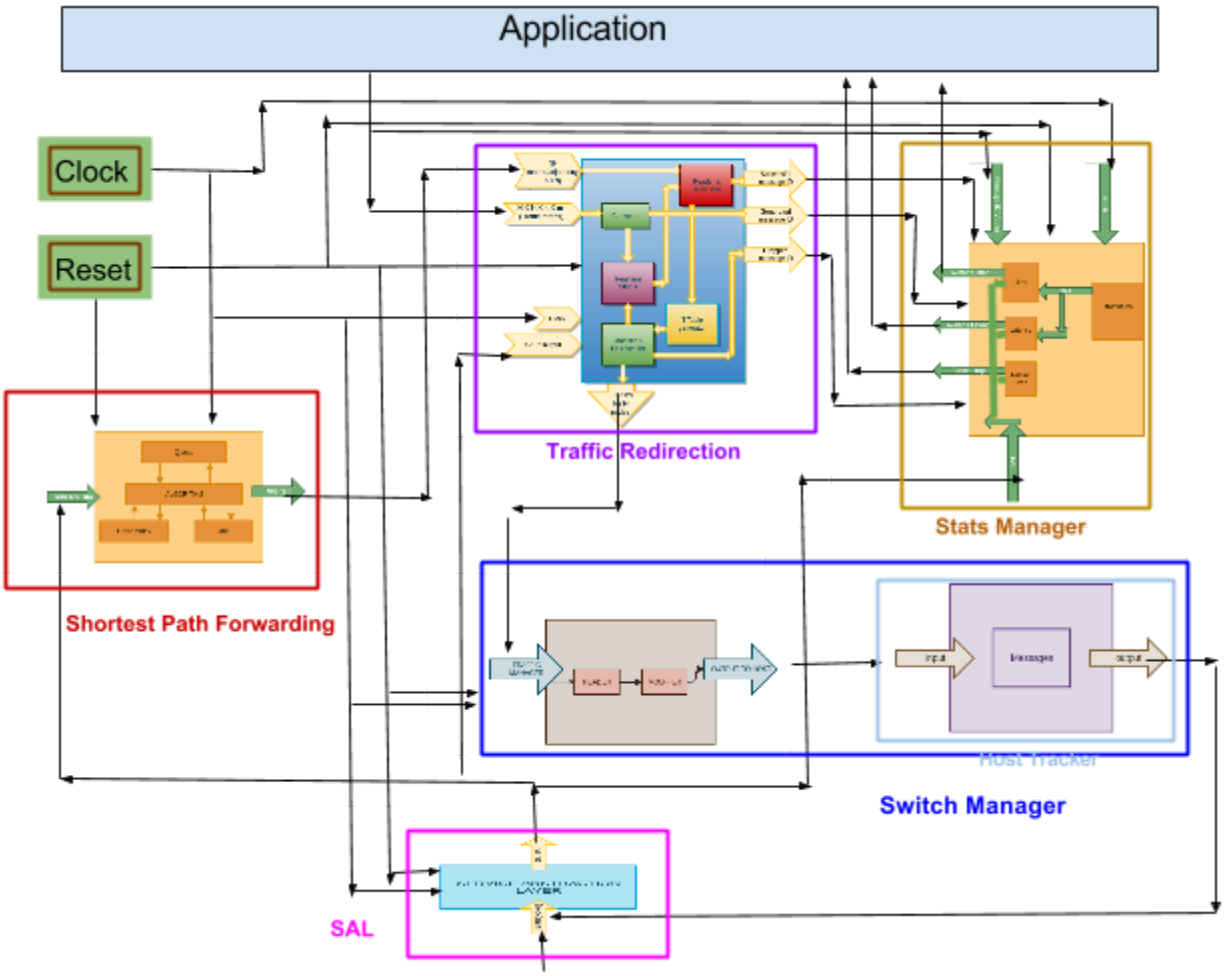
3. At t=200 clocks, we obtain the Traffic Matrix as well, and thus we can calculate the next-hop router by our algorithm block.

4. After that the processing is triggered, which basically informs each router which message to forward at each port, using output of traffic manager.

5. This results in the routers actually forwarding the messages. After that, at 20X clock cycles, we go to the previous state of obtaining graphs. At t=200X, we get both a new traffic matrix, and a new adjacency matrix, and go to the previous state of obtaining graphs. Now this cycle of states repeats over and over every 200 clock-cycles.

4. Description of functionalities

4.a) Flow of functionalities Diagram



Each of the block is explained on the following pages.

4.b) Algorithm Block :

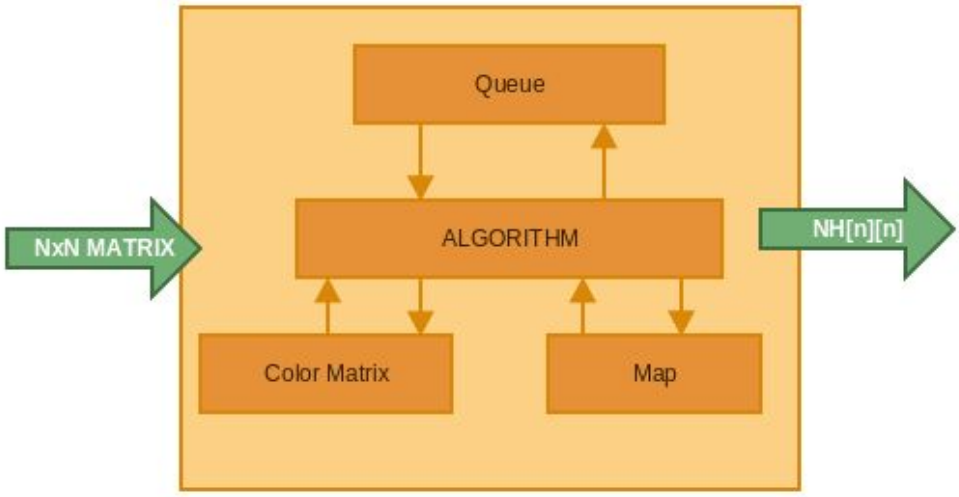
Store : NxN adjacency matrix here

SHORTEST PATH BLOCK:
Implements the path finder algorithm. The functionality is clearly described in the algorithm section.
Assumptions: No special assumptions. If the graph is connected, we get proper shortest paths via BFS and get the appropriate next-hop router.:

Input : adjacency matrix A[n] [n]

Output : NH[n] [n]
where

1. n is number of nodes.
2. NH [i] [j] is node number of next-hop of shortest path from node i to node j ;
   -1 if no such path.



Note : Algorithm computes and stores NH matrix within 20 clock cycles.

Algorithm :

1. Let i be any node.
2. Apply BFS on graph given by matrix A with starting node i.
3. Every node j in BFS tree is reachable by node i.
4. Shortest path from i to j in graph is same as path the path from i to j in BFS tree.
5. NH[i] [j] is the child of i that is ancestor of j.

Pseudo Code :

```
For each i;
        queue q // DFS queue, initially empty
        initialize NH[ i ] [ j ] to -1 for j = 1 to j = n

        color [ 1,2,...,n ] // color[ i ] is color of i-th vertex, initially -1
        M is map from color to node.

        integer c = 0; // color : unique to each child of i
        add node i in q and mark node i visited.

        for every child w of i
                color[ w ] = c;
                increment c by 1
                add key c with value w in M
                add node w in q and mark node w visited
        pop first element of q // remove node i from queue

        while q is not empty
                p is the first element of q
                for every child w of p
                        if w is not visited
                                color[ w ] = color[ p ] ;
                                NH[ i ] [ w ] = M[ color[ w ] ] ;
                                add node w in q, mark node w visited
                pop p from q
```

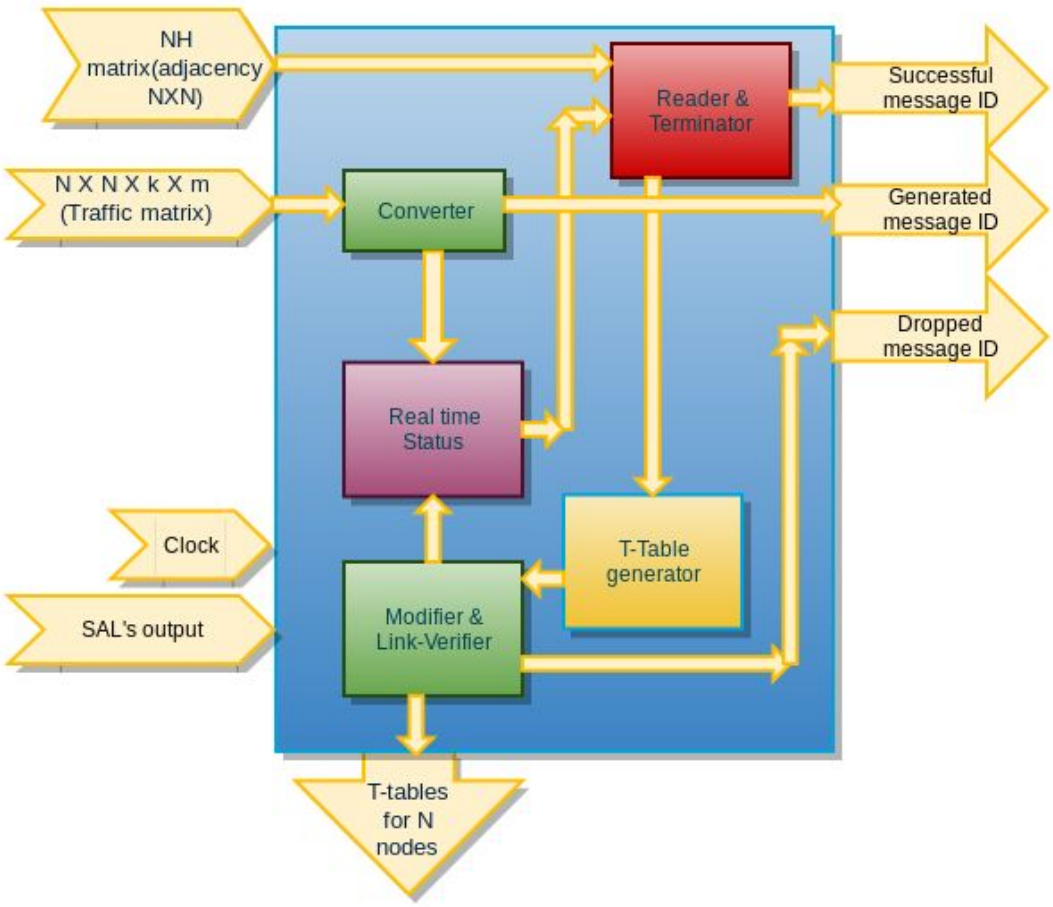
Analysis : O(n^2 * log n)

4.c) Traffic Controlling Module :

Store : NH matrix here

Input :

1. All packets available in every Node.
2. Every packet contains the Source-Node , Destination-Node.
3. Every Node has its own forwarding table corresponding to every Destination-Node which is shortest path.(NH matrix)



Assumption:

Every packet takes exactly '**1**' clock cycle to reach to its corresponding next hop node.

Expectation:

All the packets that can be “Sent” in the corresponding clock cycle from every node such that there is never a case when there is a packet available at the node and the corresponding next hop node wire is empty.

Working-Out:

1) The key Idea is to maintain a temporary table(**T-table**) corresponding to every Node having a single column and ‘p’ rows(p = number of immediate neighbours). Where every row is dedicated to every next hop node(Need to verify for existence of link from SAL(***‘Link-Verifier’***)).

2) ***Real Time Status***:
    a) This has N vectors(integer) where each vector corresponding to every node and the integer it contains represents the message IDs of the messages contained in it.
    b) It also assigns IDs to every message uniquely and gives the ID

3) ***Converter***: Its job is to take in the Traffic Matrix from standard input, Assign distinct **ID**s to every packet (and give it at output) and append it to the ‘Real Time Status’ data for continuing the packet sending method.

4) Now we do the following steps :
    a) Go through all the packets in the Node and see their Destination Node.(***‘Reader’***) If they are already at their destination remove them from the Real Time Status and output the IDs(***‘Terminator’***).

    b) We get their corresponding next hop node from the NH matrix taken in input.

    c) If the row corresponding to the next hop is empty(or if the packet it contains has priority less than the priority of the packet in view) we put the packet in view in that row. (Thus we achieve **applications/services prioritization**)

    d) Once we have traversed through all the packets in a Node what we get is the list of packets to be sent in the corresponding clock cycle. (Thus we achieve **Work conserving memory controller**)(***‘T-table generator’***)

e) This we do for every node. The corresponding listed packets are removed from the memory of their contained Nodes and appended to the memory of their next hop nodes(***'Modifier'***)

   i) Here we implement our logic for packet dropping. In case the number of packets in the Node is already 10 then we decide to **Drop** that packet(output the IDs corresponding to them) (This can be improvised by making it wait for a constant number of clock cycles instead of dropping it. Note that this does not affect the work conserving scenario as that wire cannot be used by any other node.
   ii) If the Link to a next hop is destroyed for a particular Node(Info obtained from SAL) then the packet is supposed to buffer till a new graph is obtained at the 20th clock cycle(***'Link-Verifier'***)

Example:

1. Consider the following representation of a packet(M). M = (message id , next hop node number , application priority , etc.)

2. Consider a node(node number 3) who has 3 possible next hops(immediate neighbours). Suppose it initially has the following packets and its T-Table is empty.

3. (1,4,1),(2,4,1),(3,5,1),(4,9,1),(5,4,2),(6,9,1),(7,9,1),(8,9,1)

4. After processing we will have the following T-Table for this node:

(5,4,2)
(4,9,1)
(3,5,1)

Which means that from this node we must send these three packets to their respective next hops and retain the other packets in the memory(**buffering**).

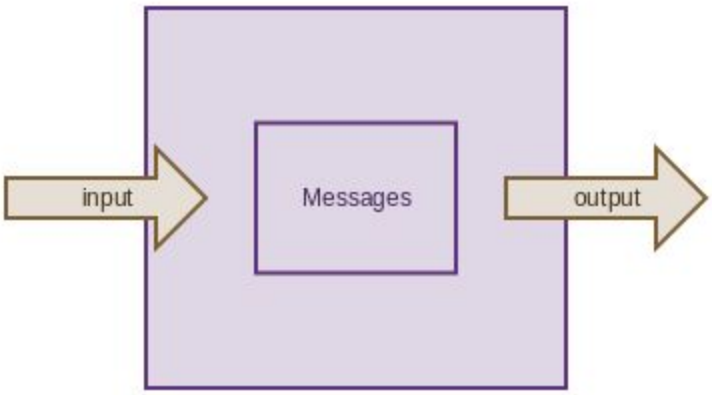
Note that for packets with same next hop and same priority application we are using our first-in-first-out logic.

4.d) Host Tracker :

Store : All the messages at every Node

Input : Messages contained at every Node and its current status

Output : Starting Node, Destination Node and the Current Node where the Message is present for each and every Message



Description :

Input will be taken from Traffic Manager output.
It keeps an account of message ID's present at every node.
This thus stores the position status of every message ID with respect to its routing string.
For every message ID, we get the node number where that message is present, as an output.

Assumption:

We assume that we get the status of each message ID from the traffic manager correctly at each clock cycle.

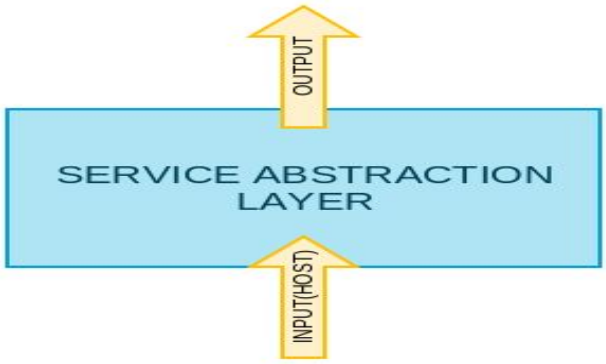
Note:The host tracker is combined with the Traffic Module in the code as its functionality was being satisfied in it.

4.e) Service Abstraction Layer : Top Module

Store : Real Time Status of each and every link between Nodes

Input : The connectivity of the links at every Node and instances of new link Generation

Output : The Adjacency matrix at every 20 clock cycles according to the status of the Links. Also information of New links created or Old links destroyed is given at running clock cycle.



Description :

The input to this is the south-bound traffic activity.
We basically know if suddenly a link fails, via SAL input.
Also, if there is packet drop, or there is packet delivery, this is intimated to SAL as input, and SAL processes this information giving it as its own output.
This output becomes input to all modules of SDN controller

Assumption:

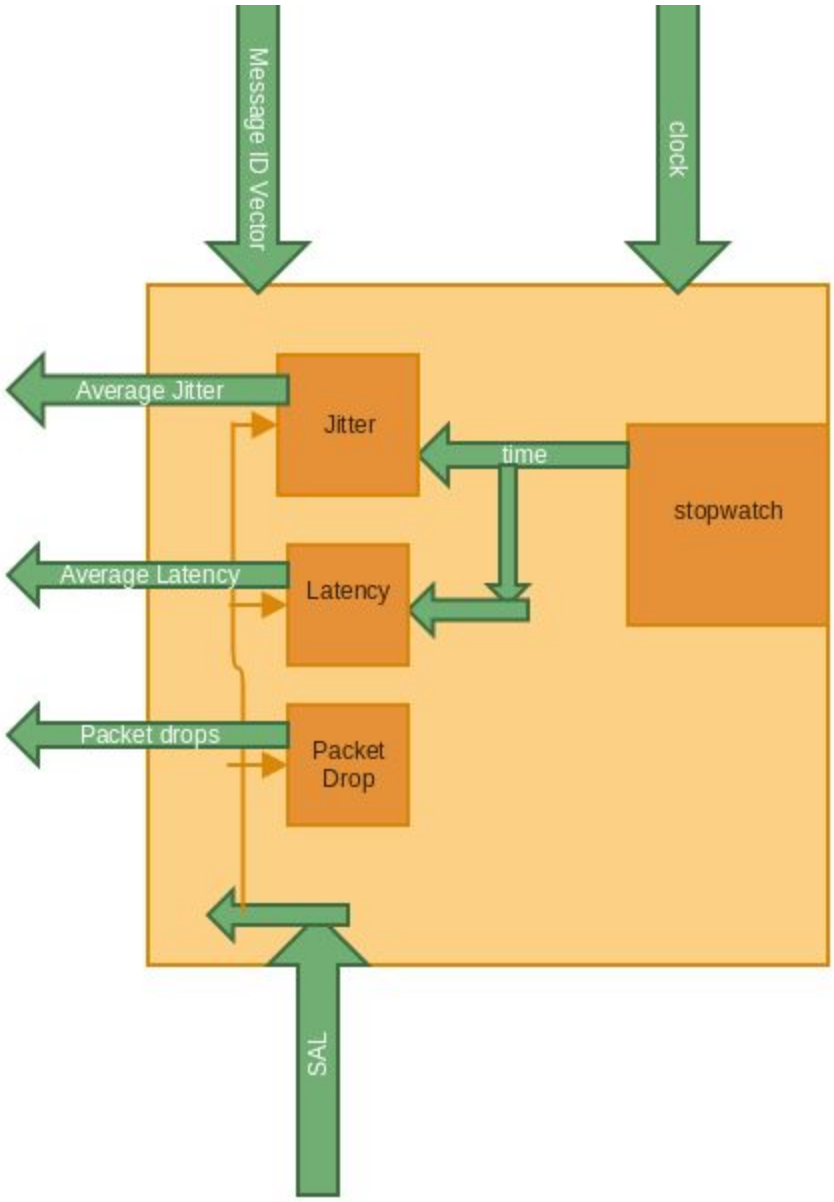
We assume that we are able to monitor the status at each router (regarding what packets a router has received at every clock cycle, what packets are dropped at a clock-cycle, and what packets are received finally at destination).

4.f) Statistics Manager :

Store : Time of traversal for every packet and total number of packets dropped.

Input : Message IDs of new messages added to request, reached destination and dropped

Output : Average Jitter, Average Latency and Total number of packets dropped.



Description :

1. This module takes in the message ID vector, clock and SAL output as its input.
2. SAL output relevant to this module simply consists a signal such that
    a. Signal indicates event when there was packet delivery at a particular destination node with corresponding message ID.
    b. Signal also indicates event whenever a router drops packet of a message ID because of queue length limitation.

3. So, whenever a destination node ‘x’ receives packet from source node ‘y’, the SAL informs the statistics module that this packet delivery is done.
4. The jitter and latency modules will attach the corresponding time at which the packet was received to that particular message ID entry.
5. Then jitter and latency is calculate in O(1) time for each such delivery.
6. Similarly, drop counter is incremented in the packet drop module whenever there is packet-drop at any router in O(1) time.

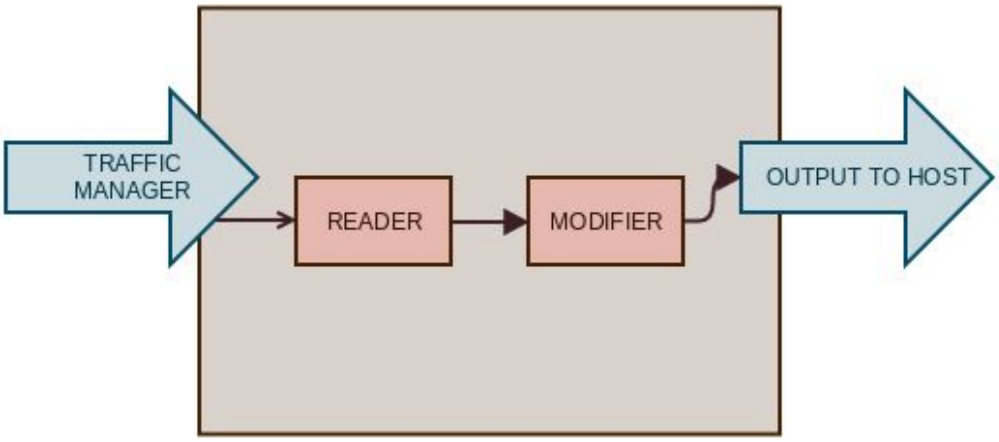
Assumption:

No special assumptions. As long as this module is getting inputs at the correct clock tick event, it will calculate the correct statistics.

4.g) Switch Manager (coded in the traffic module):

Input : Packets that are to be sent along with their respective traversing node and next-hop node

Output : Change the location of the packets according to the input along the links that are already existing. Also if a link is destroyed then send the negative acknowledgment to the source node of message



Description :

Takes input from Traffic Manager at each clock cycle and gives as output to update the message status with respect to routing string at different nodes into the host tracker.

Assumption :

We are getting the message position status from the traffic manager at every clock cycle for every message ID.

5. Plan for testing

Generation :

Generating 45-50 test cases using random function in python, which consists of:

1. NxN matrices
2. NxNxkxm matrix for messages

Expected Output :

1. We will collect the statistics and observe the trends for messages which reached , number of messages lost, time statistics.
2. If these matches real time statistics then our model is working fine.

Corner cases:

Case 1 :

Every node has number of messages to be sent equal to saturation limit.

Case 2:

Every message has high priority or every message has a common destination

6. Further planning

1. Increment k from k=1.
2. Show how message is being transmitted.
3. If possible we would like to simulate the network by network simulation tools and record data during the simulation. For this we need to consult our network's instructor and if time allows would continue with this implementation.

STAGE - 2

Index

1. Contribution of Work:

Stats Manager : Shachi
Algorithm : Deep
Traffic Module, Switch Manager, Host Tracker: Neeladrishekhar and Ashna

Service Abstraction Layer, Combining, Testing, Debugging: All members did this together

2. Notable Milestones Achieved :

1) Any level of of priority (k) [Will vary the input style accordingly]
2) Number of nodes and buffer strength of each node can be varied. All the variables need to be changed in the package and generic values in the respective Components accordingly.
3) Ability to display all the message packets currently in transit via real_time_T_Table signal.
4) Ability to show all the packets currently existing at every node in real_time_status signal.

3. Further scope of project:

We can add weights to the paths, handle things like congestion control and flow control in the network by dynamically adjusting rate of flow of packets into network, etc.

4. Changes Made in proposed Design:

1. Host tracker inside Traffic module: The functionality of host-tracker and Switch manager is added into Traffic Management module itself.

   This reduces memory requirements, though modularisation reduces.

2. Jitter and Latency Calculation:

   This is done as following

New Jitter=(Old Jitter) + (Difference between of recently delivered packet from latency of last packet delivered)/16

Also, the values of latency and jitter are approximate, since we have used integers for those calculations, and hence there is truncation many times. To avoid divide-by-zero, we increase the denominator by 1 while doing all divisions.

3. TTL:

   We assume that each packet in the network has certain Time To Live (TTL).
   due to which packets don't stay in the network for more than TTL time.

   Here we assume that TTL is less than 200 clock cycles,
   hence if some packet is not delivered to destination within 200 clock-cycles
   (i.e. before arrival of new traffic matrix), such packets get dropped automatically.

5. Instructions to run code :

1.) open Radiance/design
2.) extract RadianceCode.zip
3.) open RadianceCode.xise
4.) in design tab of xilinx window, choose Implementation view.
5.) click on SAL- Behavioral(Radiance_top.vhd)
6.) run Synthesize - XST
7.) in design tab of xilinx window, choose Simulation view
8.) click on SAL_test - behavior (Radiance_testbench1.vhd)
9.) run Behavioral Check Syntax
10.) run Simulate Behavioral Model
11.) rerun the simulation for atleast 3us

NOTE :
It may take some time to synthesize the code.

6. Explanation of Simulation Inputs and Outputs :

a) Inputs :

'A matrix' is NxN matrix representing adjacency Matrix
A(i)(j) is true iff there is edge from node i to node j
traffic_matrix(k)(i)(j)(m) = "t" represents from node i to destination j and m'th message of priority "k_max - k - 1"

number of 1 in "t" represents the number of messages to be sent with above specification

en_in is true when "A Matrix" input is valid. (must be true for at least 1 clock rising edge)
traffic_en is true when "traffic Matrix" input is valid. (must be true for at least 1 clock rising edge)

b) Outputs :

NextHop: type array of array of std_logic_vector
Here NextHop(i)(j) represents the next hop node from i'th node to j'th final destination.
Real_Time_Status:
Real_Time_Status(i)(j): represents the packet at node i's => j'th memory block.
Real_Time_Status(i)(j).cur : represents the node its currently present at(which is 'i').
Real_Time_Status(i)(j).dst : represents the destination node the packet is targeted at.
Real_Time_Status(i)(j).src : represents the source node the packet started at.
Real_Time_Status(i)(j).idN: represents the id of the packet .
Real_Time_Status(i)(j).k : represents the priority level
(application)
of the packet.

Real_Time_T_Table:
Real_Time_T_Table(i)(j) : represents the packet at node i destined to node j and will go to its next hop.
Real_Time_T_Table(i)(j).nxt : represents the next hop of that packet.
Real_Time_T_Table(i)(j).cur : represents the node it is currently present at.
Real_Time_T_Table(i)(j).idN : represents the id of the packet.
Real_Time_T_Table(i)(j).idx : represents the memory position of this packet in its current node.

Latency: integer average end to end delay over all nodes
Jitter: integer variation of latency over all nodes
Total_Dropped: integer total packets dropped
Latency_Status: array of array of integer
Latency_Status(i)(j) represents the average latency of path from node i to node j.
Jitter_Status: array of array of integer
Jitter_Status(i)(j) represents the jitter of path from node i to node j.

Note : average values may not be exact. (to avoid division from zero)

7. Overall view of the working of simulation

Note : All images are available in Radiance/others folder with same name as name of figure.

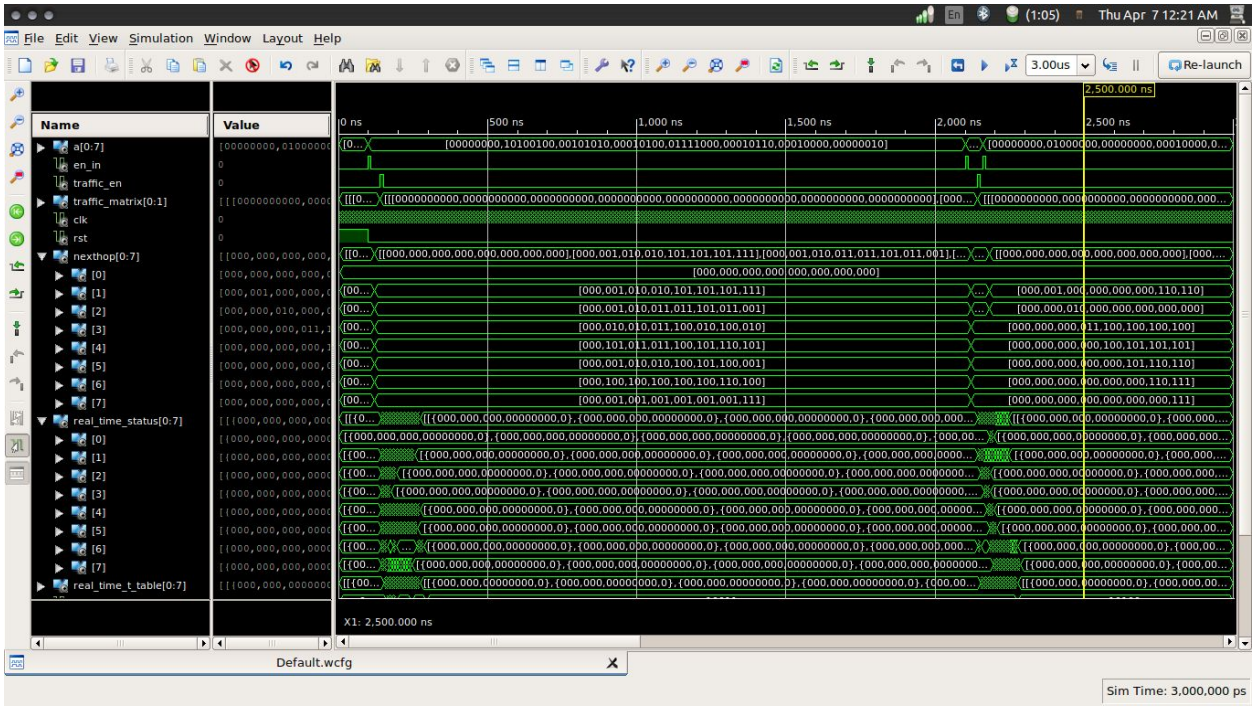
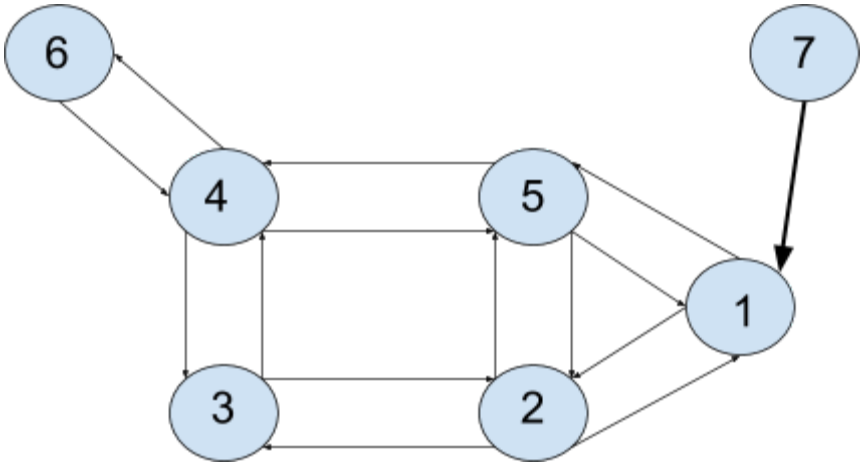


Figure - 1_Representation

a) Graph in Consideration :



b) Details of real_time_status of a packet :

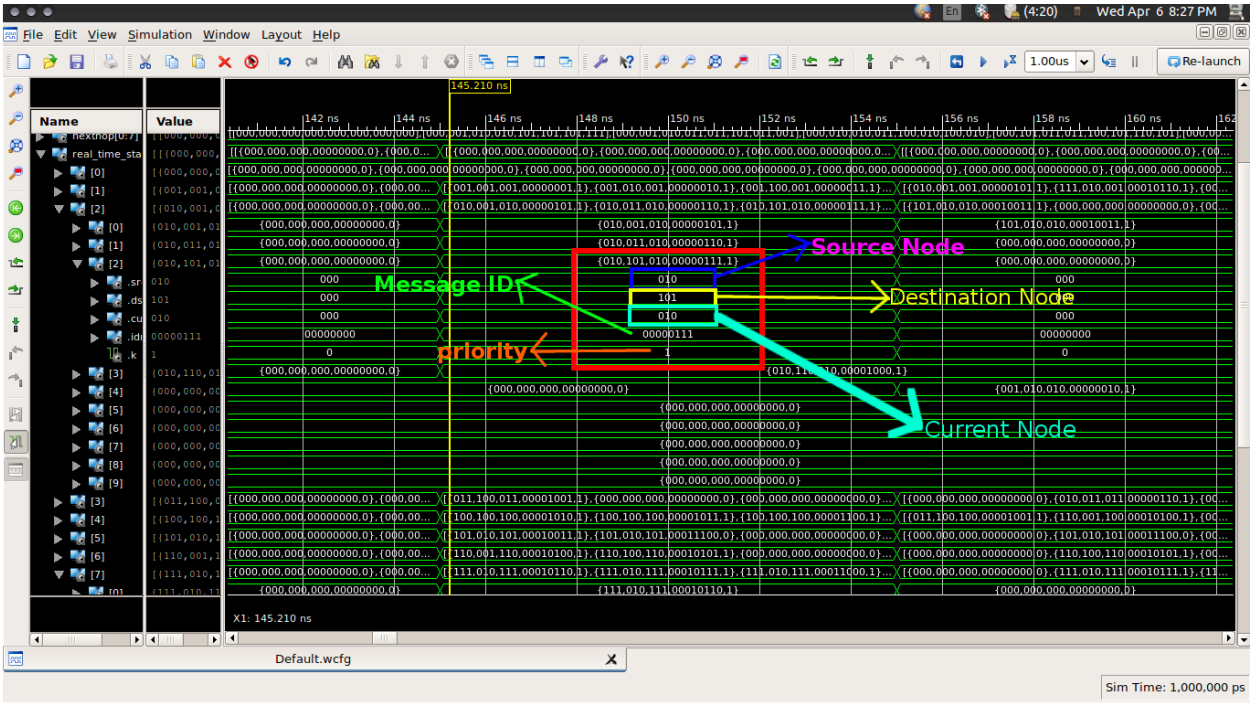


Fig-2_Work_Conservation

This image shows the packet travel from node 2 to nodes 1,3,5, and 6.
Here, we show the information of packet traveling from node 2 to 1.

c) Tracing the path of a general packet

We see the travel of packet from node 2 to node 6, the ID of this packet being 1000 (i.e. 8)
Since a packet is traveling to node 5 (with ID 111, i.e. 7)

this current packet cannot go to this next hop node 5 via the same link, and hence it stays at the same node in next clock cycle.

After this, it travels progressively to node 3 (time 165 ns), node 4 (time 175 ns), and node 6 (time 185 ns), since it chooses one of the 2 available shortest paths (e.g. 2-5-4-6 and 2-3-4-6).

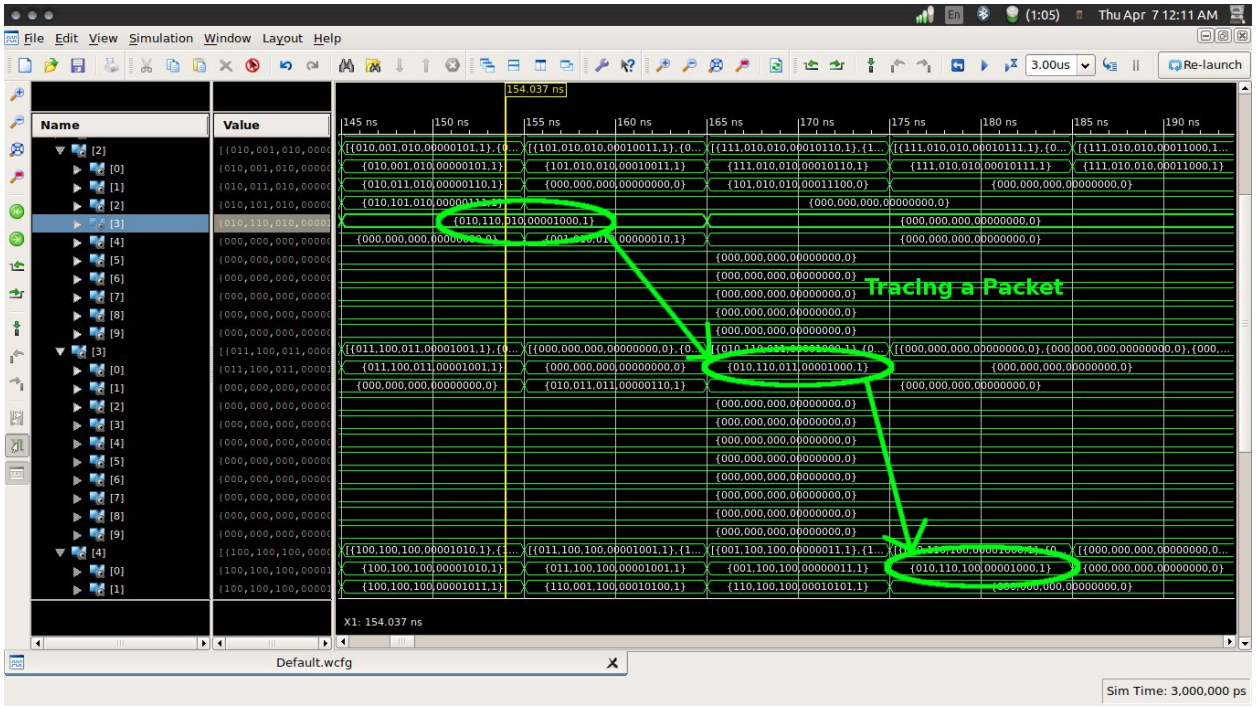


Fig-3_Priority

In this image we see the above package reaching the destination node 6, and in the immediate next clock-cycle, this particular package is passed on to higher layer, and is thus cleared off from node 6.

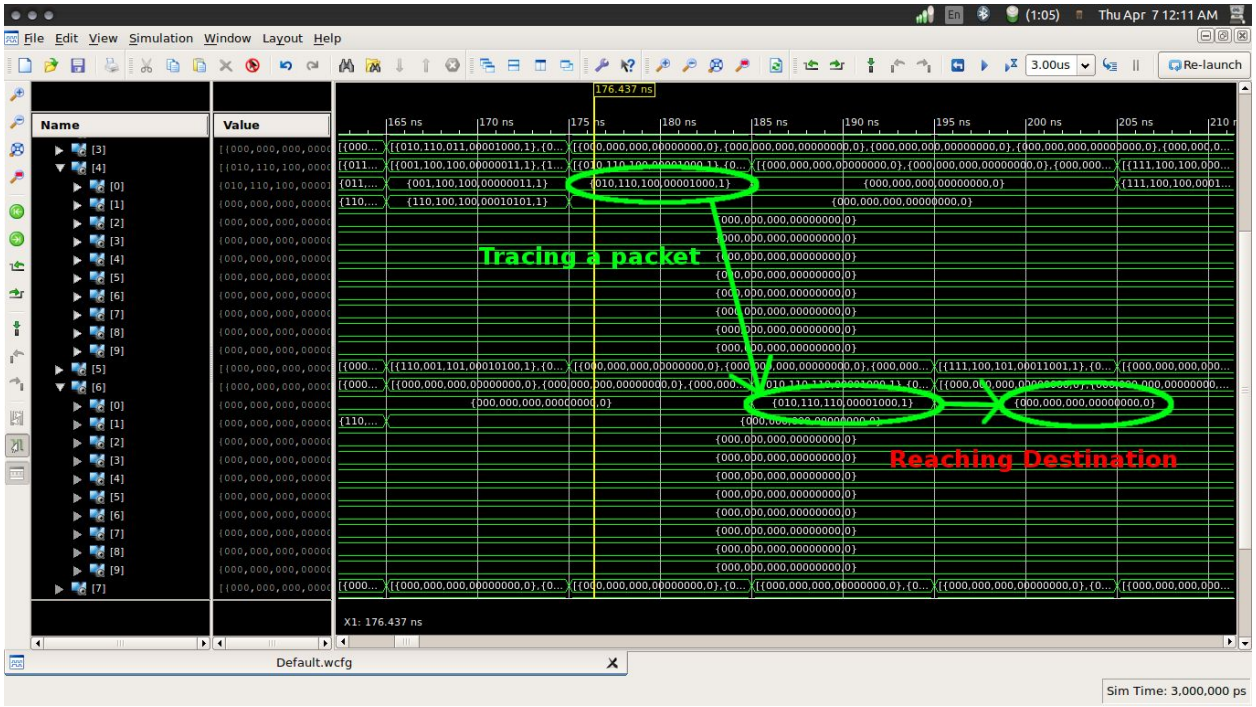


Fig-4_One_message_at_a_time_on_a_link

d) Packet Details

Source	Destination	Priority Level	ID
1	1	1	1
1	2	1	10
1	4	1	11
1	7	1	100
2	1	1	101
2	3	1	110
2	5	1	111
2	6	1	1000
3	4	1	1001
5	2	1	10011
5	2	0	11100
6	1	1	10100
6	4	1	10101
4	4	1	1010 to 10010 (9 messages)
7	2	1	10110 to 11000 (3 messages)
7	3	1	11001 to 11011 (3 messages)
7	5	0	11101, 11110 (2 messages)
7	6	0	11111, 100000 (2 messages)

e) handling different priority levels of packets :

Here, 2 packets originate from node 5, with destination node 2, but different priority levels, i.e. 0 and 1, with 1 being packet of higher priority.

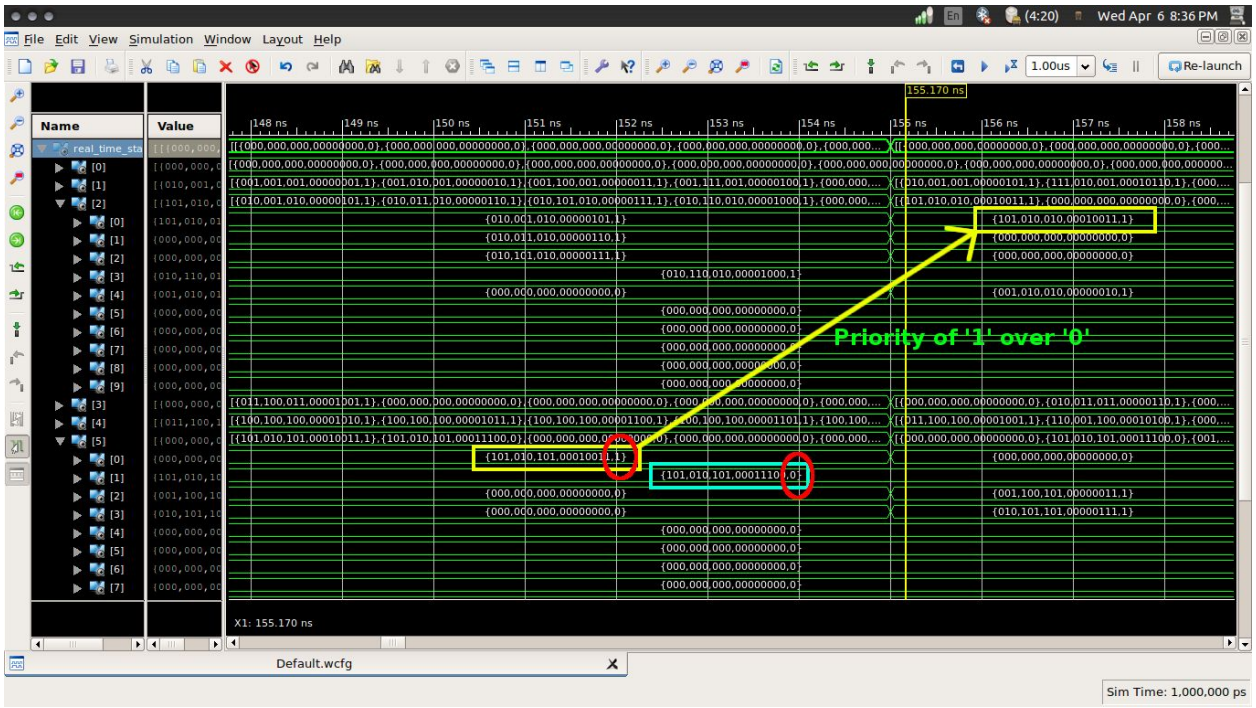


Fig-5_Memory_management

Thus, the packet with higher priority level (ID 10011) gets delivered to node 2 in the next clock cycle, while the packet with lower priority (ID 11100) stayed back at node 5.

f) Implementing work conserving model:

Here, we have 3 packets originating at node 2, with destination nodes 1 (ID 101), 3(ID 110) and 5(ID 111).

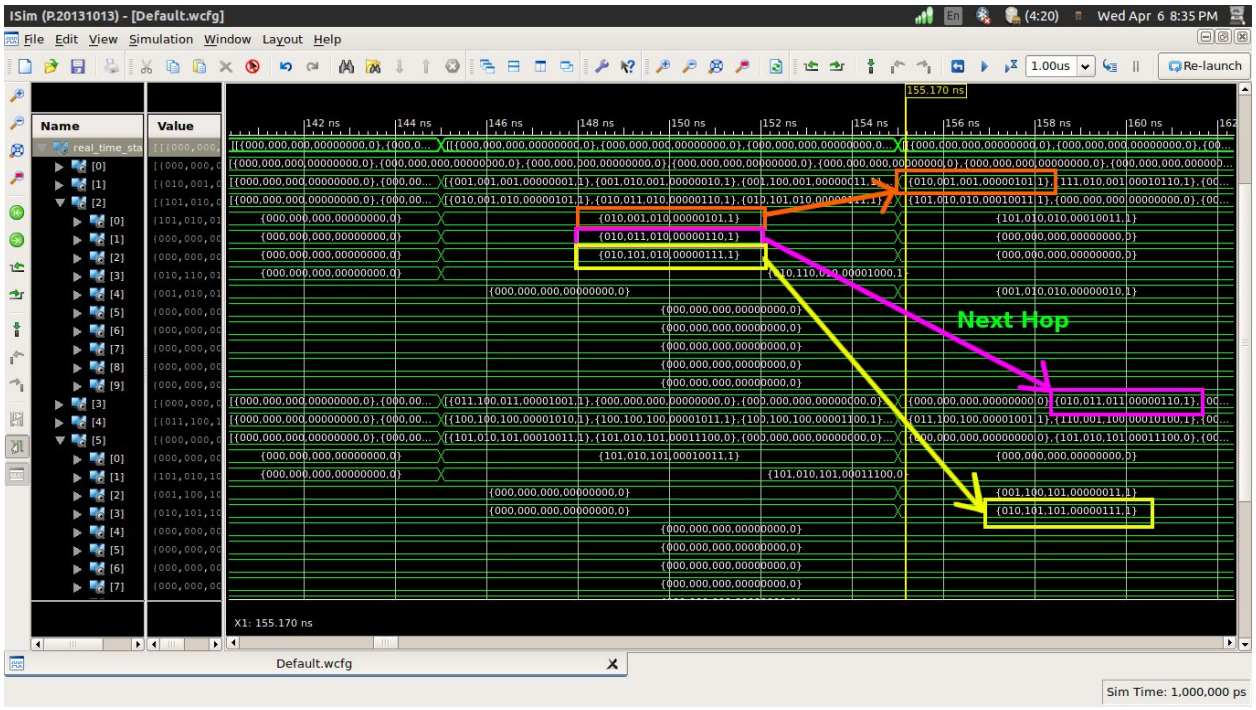


Fig-6_buffer_capacity_and_priority

We see that in the immediate next clock-cycle, these packets get forwarded to the correct next-hop nodes, which are destination nodes themselves (since 2 is directly connected to these 3 nodes).

g) Priority-wise drop of packets due to maximum buffer capacity:

Here we see the following packets originating from node 7:
With priority level 1:
111 and 0100001001
With priority level 0:
11 and 1111

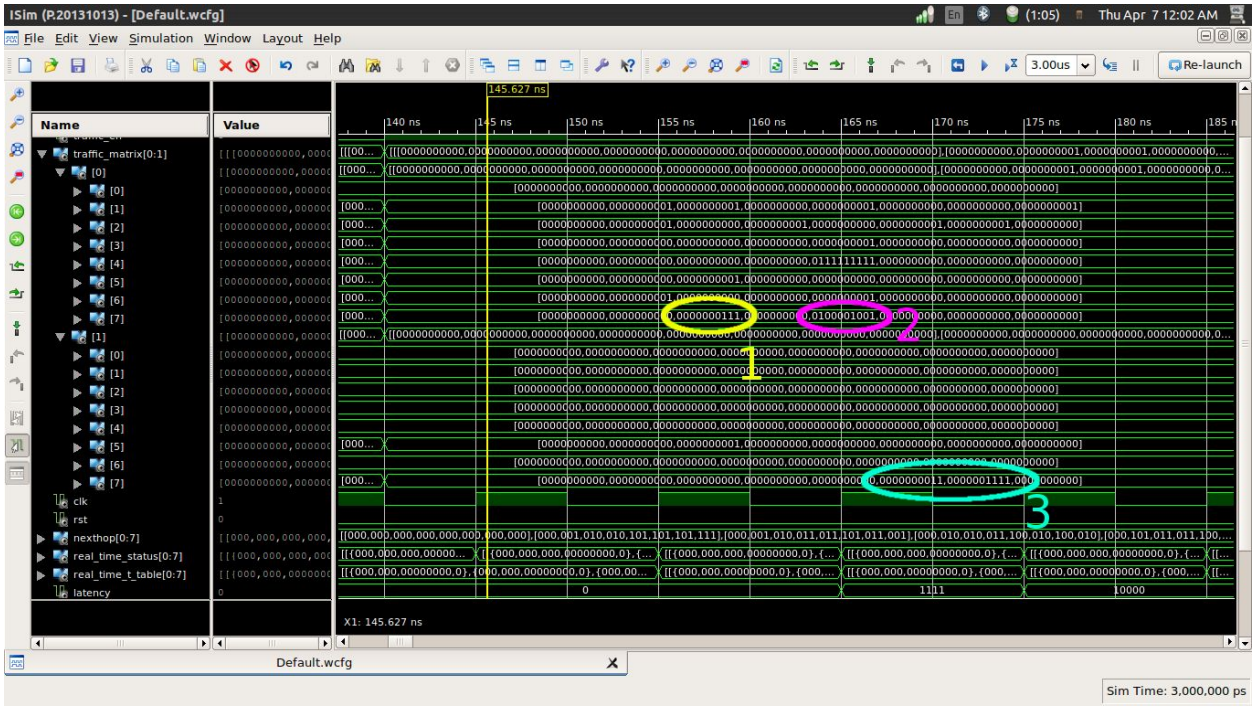


Fig-7_priority_management_while_sending_packets_from_full_buffer

Note : in circle number 2 we can see that it is not necessary for the packets to be consecutively filled, and can be filled in any order.

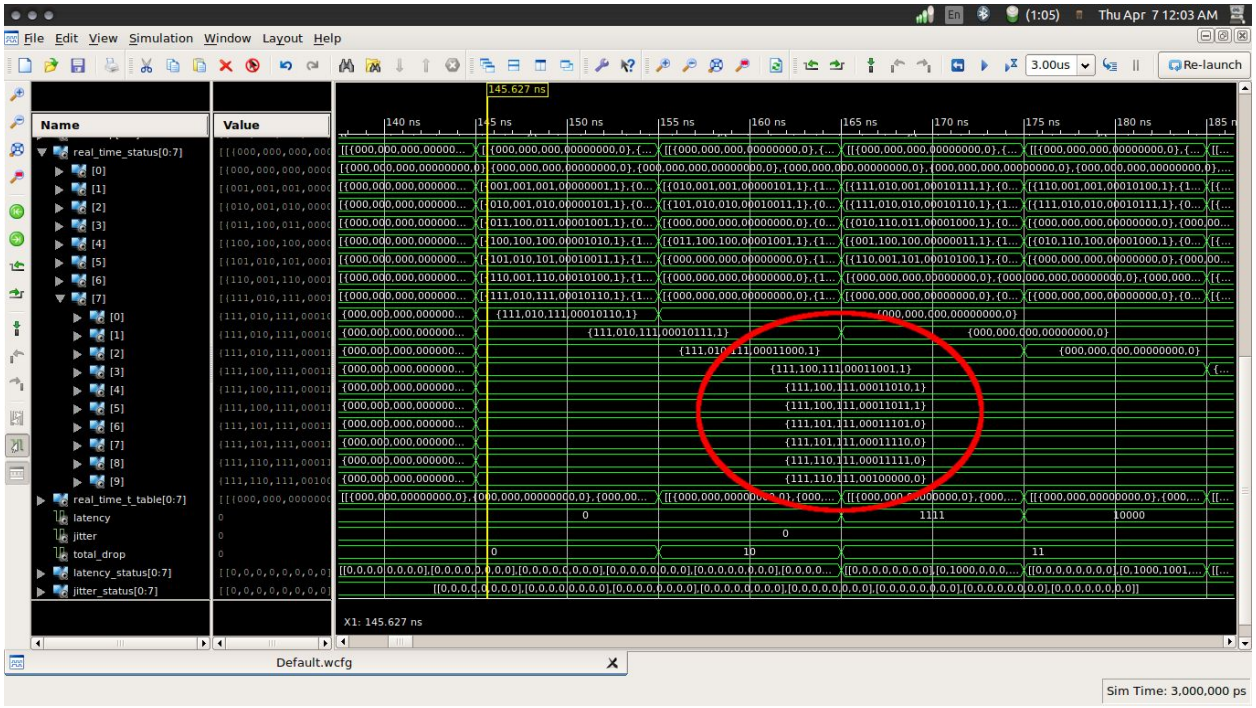


Fig-8_managing_unreachable_nodes

We thus see packet drop counter rise here, with the loss of 2 packets from lower priority levels.

h) Loopback Support, Action in case of unreachable node (due to half-duplex link), correct next-hop node calculation

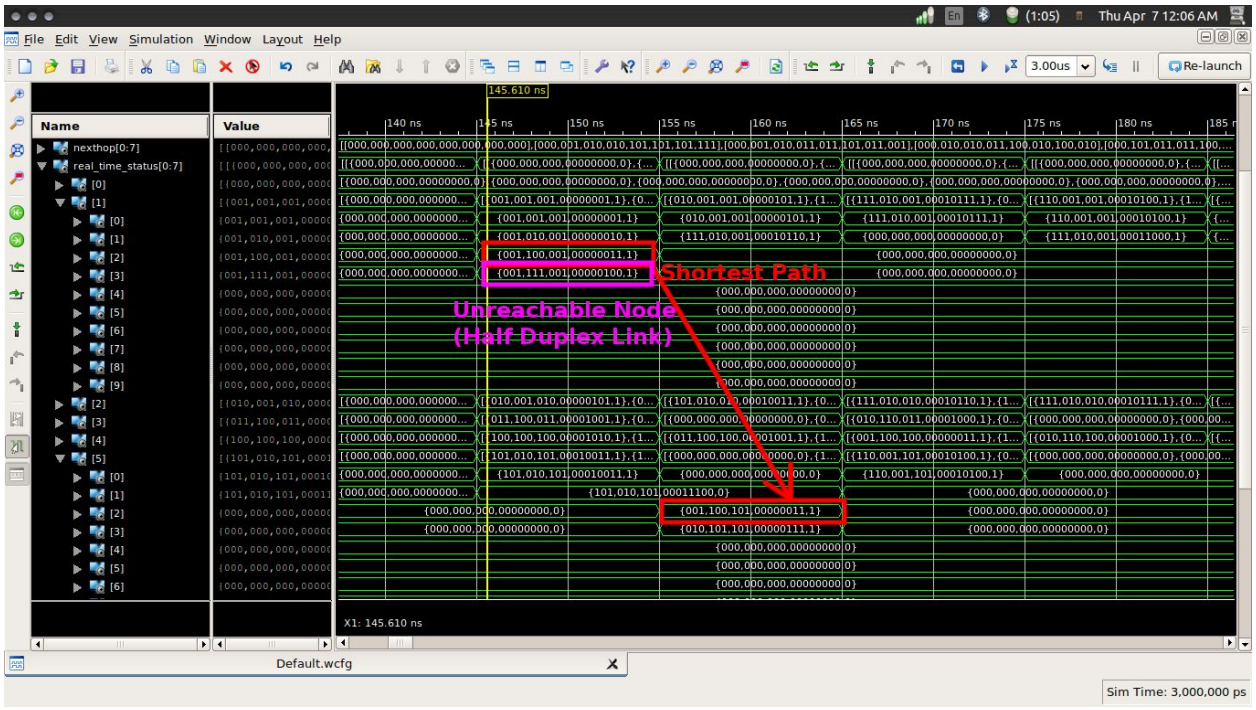


Fig-9_tracing_packet1

Here we see 4 packets originating from node 1:

node 1 to node 1 , i.e. loopback (ID 001)
node 1 to node 2 (ID 10)
node 1 to node 4 (ID 11)
node 1 to node 7 (ID 100)

In case of node 1 to node 1, packet is basically delivered to self, and is handed to higher layer of same node.
Thus it disappears from the node 1 in next clock cycle.

For node 1 to node 4, we see that the next hop node can be either 3 or 5. By the choice of algorithm between 2 shortest paths possible, the path via node 5 is chosen, and we see this packet coming to node 5 in next clock cycle.

For packet going from node 1 to node 7, the packet is dropped directly, since the shortest path to this node does not exist at all.

This happens since there is a path from node 7 to the rest of network, but no path from rest network back to node 7 (half duplex link). Thus we see this packet being dropped in the next clock cycle.

i) Self-loop and limited buffer space leads to following interesting observations:

Only one packet is selected to be forwarded between 2 or more packets wanting to go to same next hop node from given node.

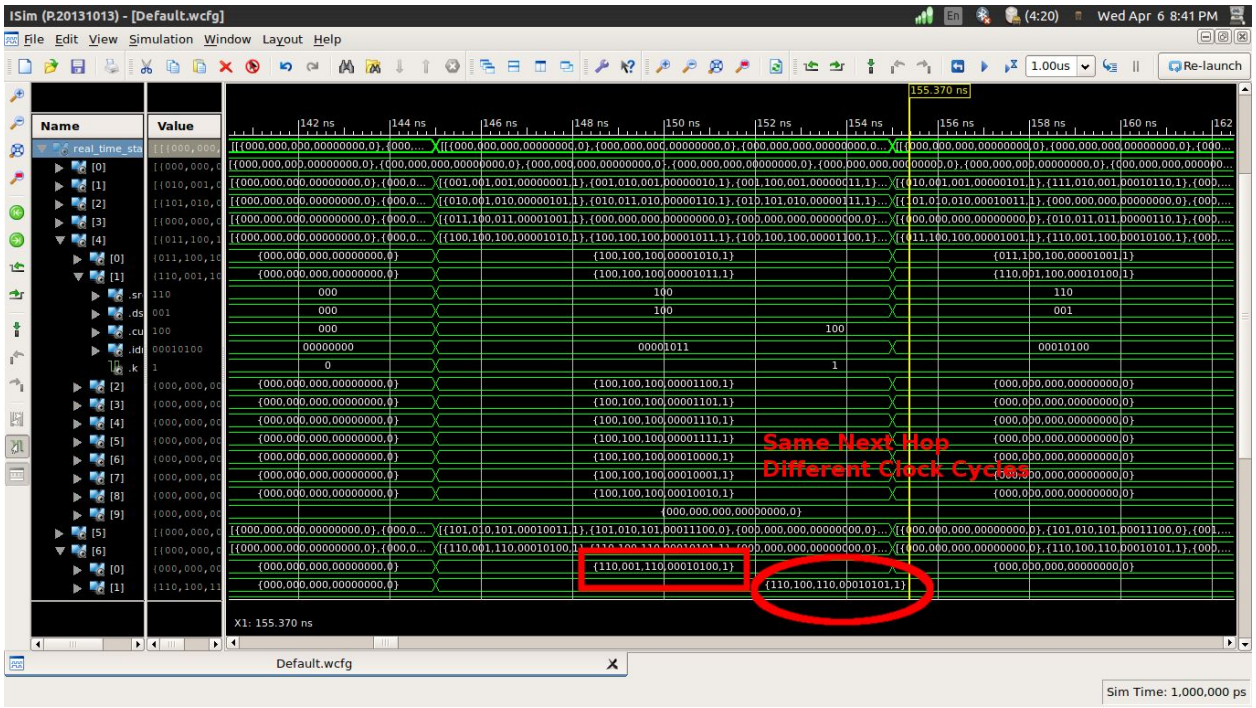


Fig-10_tracing_packet2

We see that one of the packets destined to go to node 4, i.e. a packet from node 6 and a packet from node 5, actually reaches node 4 due to space constraints at node 4 (limited buffer of 10)

There are 9 packets with same source and destination node, i.e. node 4.

Since they are destined to go to the same node, they get passed to higher layers within same node and space gets freed up.

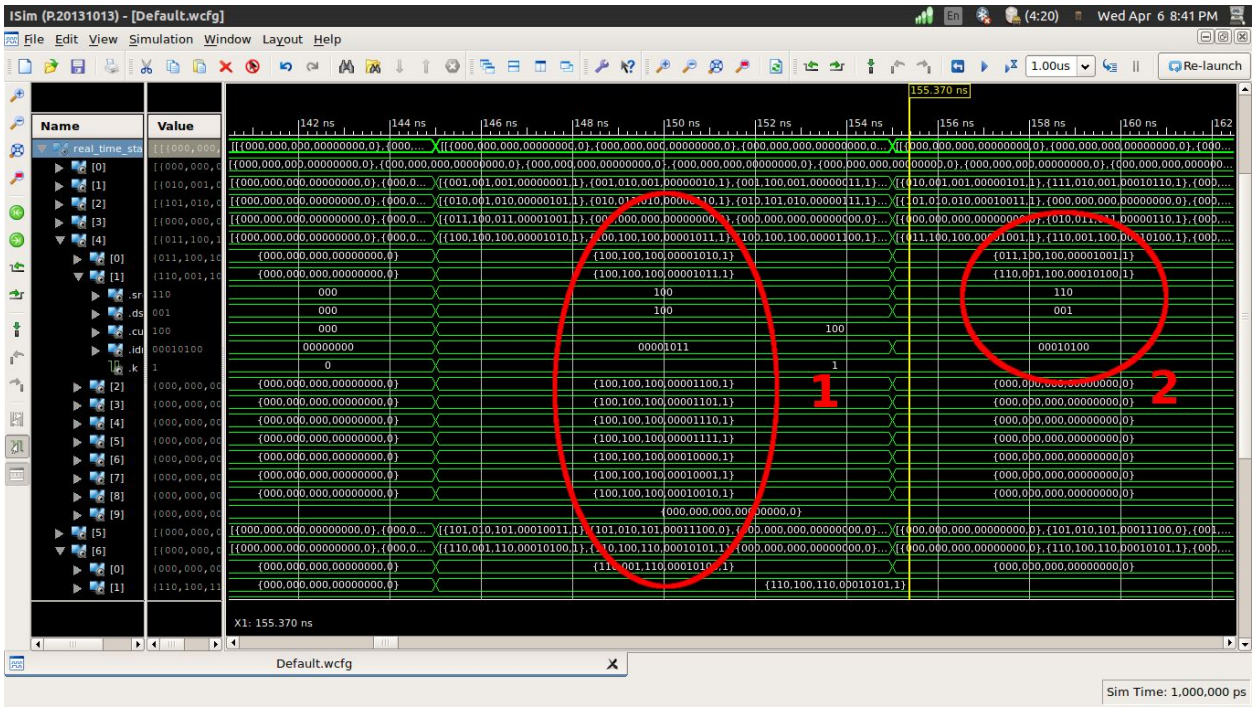


Fig-11_change_of_traffic_matrix

Thus packets coming to node 4 from node 3 and node 6, both get accommodated in node 4 as they arrive there, without getting dropped.

j) Change in traffic matrix after 200 clock cycles

After completion of 200 clock cycles, we see that there is a new traffic matrix at input, and the algorithm computes new shortest paths based on new adjacency matrix

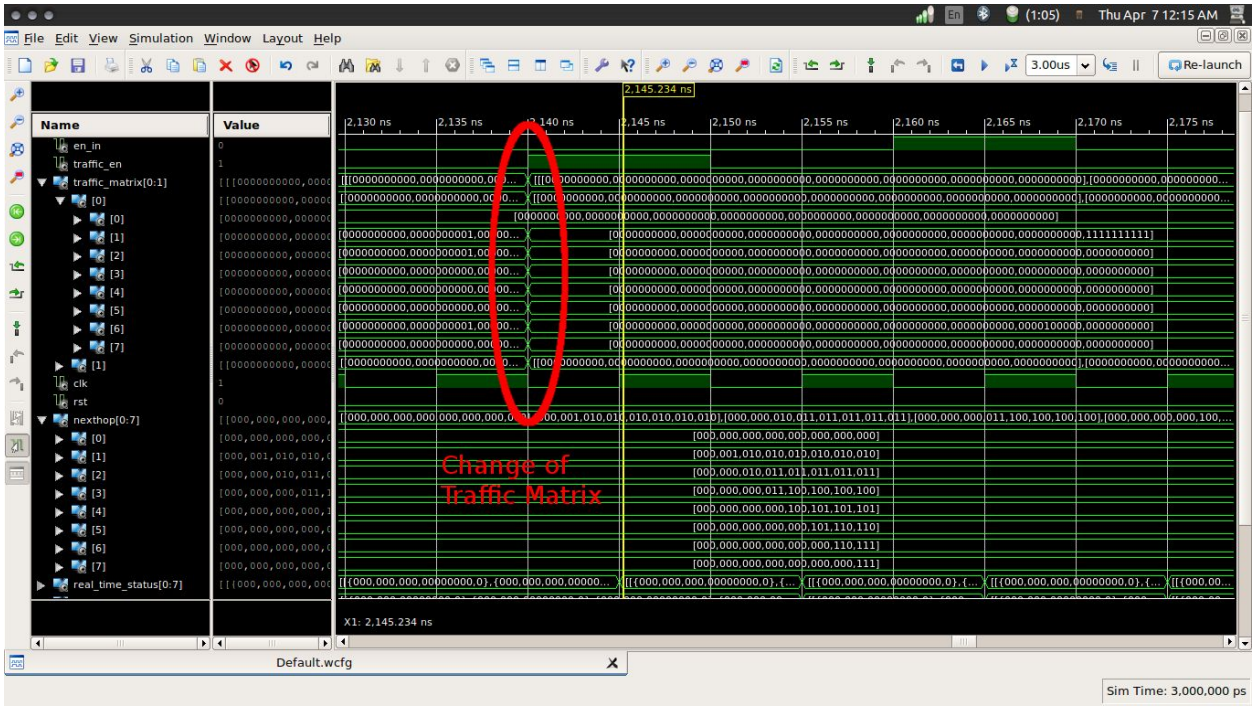


Fig-22_NHMatrix_Drop

Thus we also see change of next hop node matrix at this clock-cycle, based on new output of algorithm block.

k) Effect of change in adjacency matrix as follows:

Drops from nodes that become disconnected from destination node in new adjacency matrix

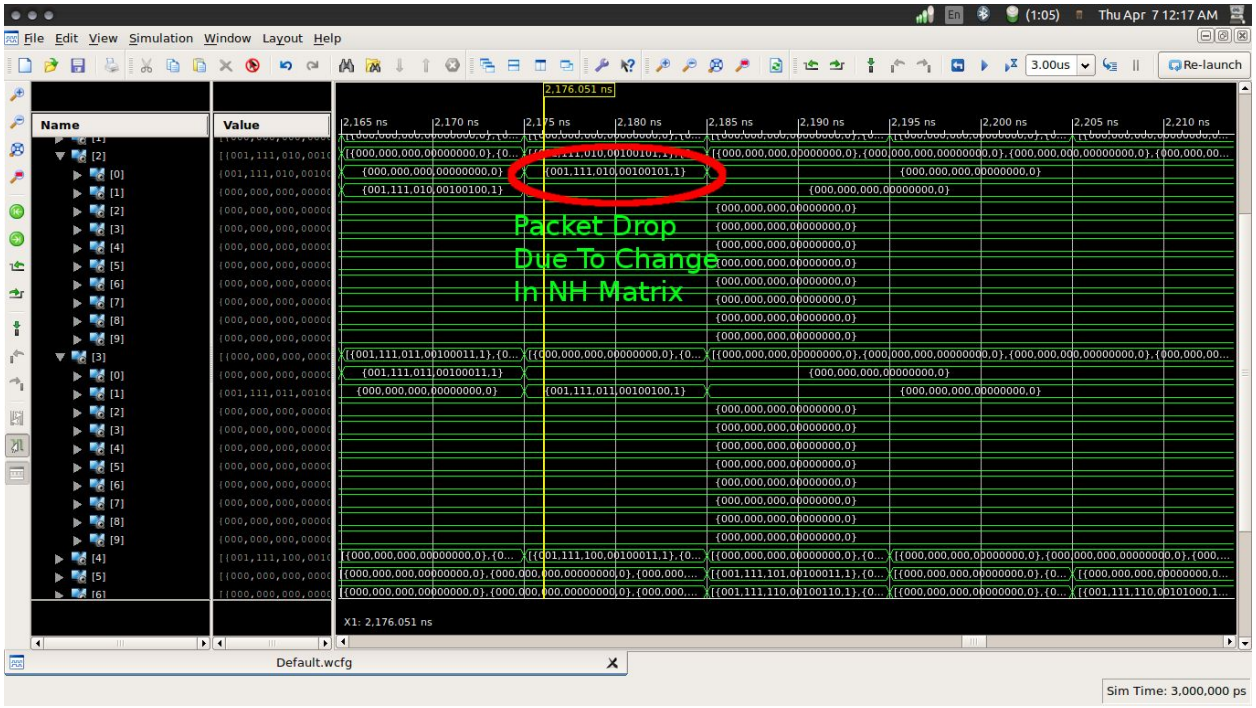


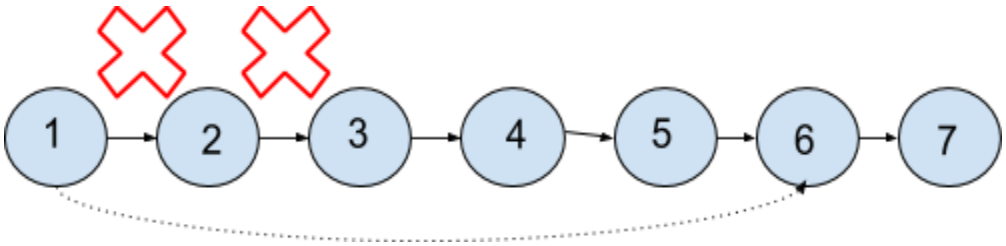
Fig-23_change_of_paths_due_to_change_of_adjacency

Here, there is change in adjacency matrix for the previous traffic matrix.

node 2 was previously connected to node 3, and node 1 to node 2.

Since these 2 links are broken, now the packets destined to go to node 7 from node 1, that had been at node 2 have nowhere to go.

Hence we see drop of packet from node 2 (ID 100101).



I) Change of next-hop nodes due to change in adjacency matrix

Since the adjacency matrix changes, as mentioned above, the algorithm block calculates shortest path again.

thus re-computes new next-hop nodes, and hence we see that the paths change at this clock-cycle.

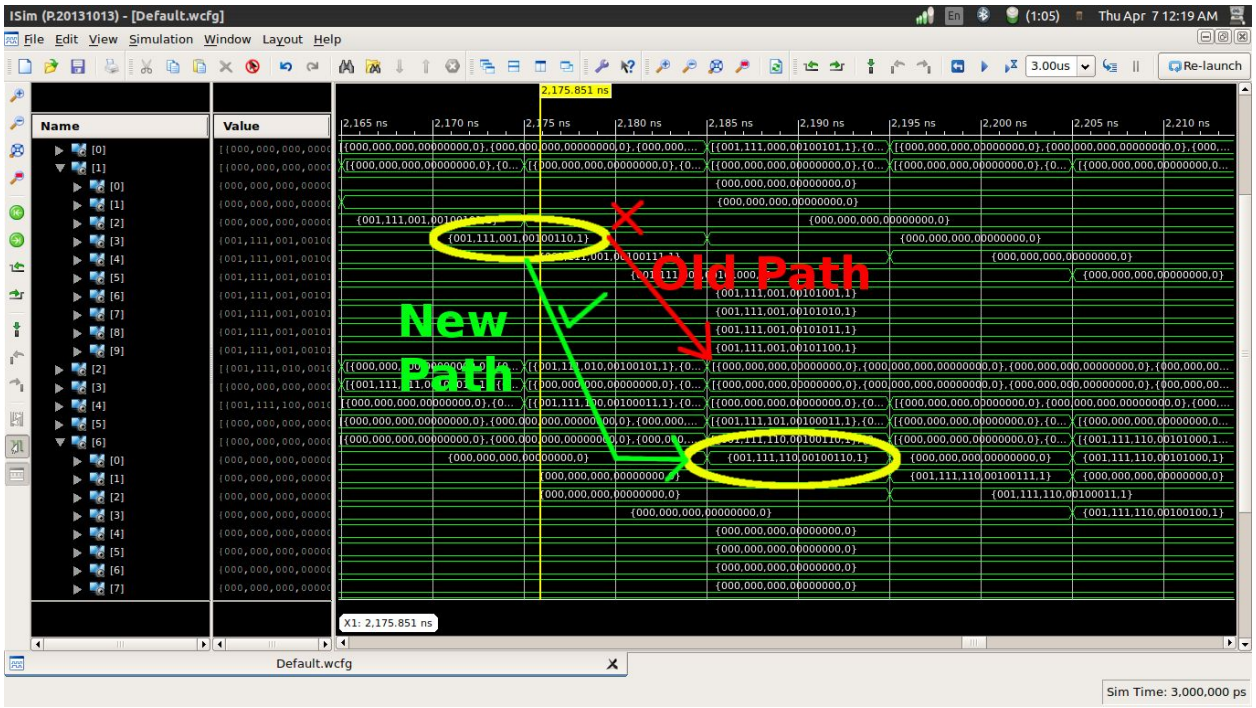


Fig-24_change _of_NH_matrix

Hence now packets are forwarded to fresh next-hop nodes. Before change of adjacency matrix, node 2 was next-hop node for node 1.

However now the next hop node for node 1 is node 6 as can be seen here.

So, we can see the packet with ID 100110 which was on node 1 and destined to node 7 go to node 6 in next clock cycle.

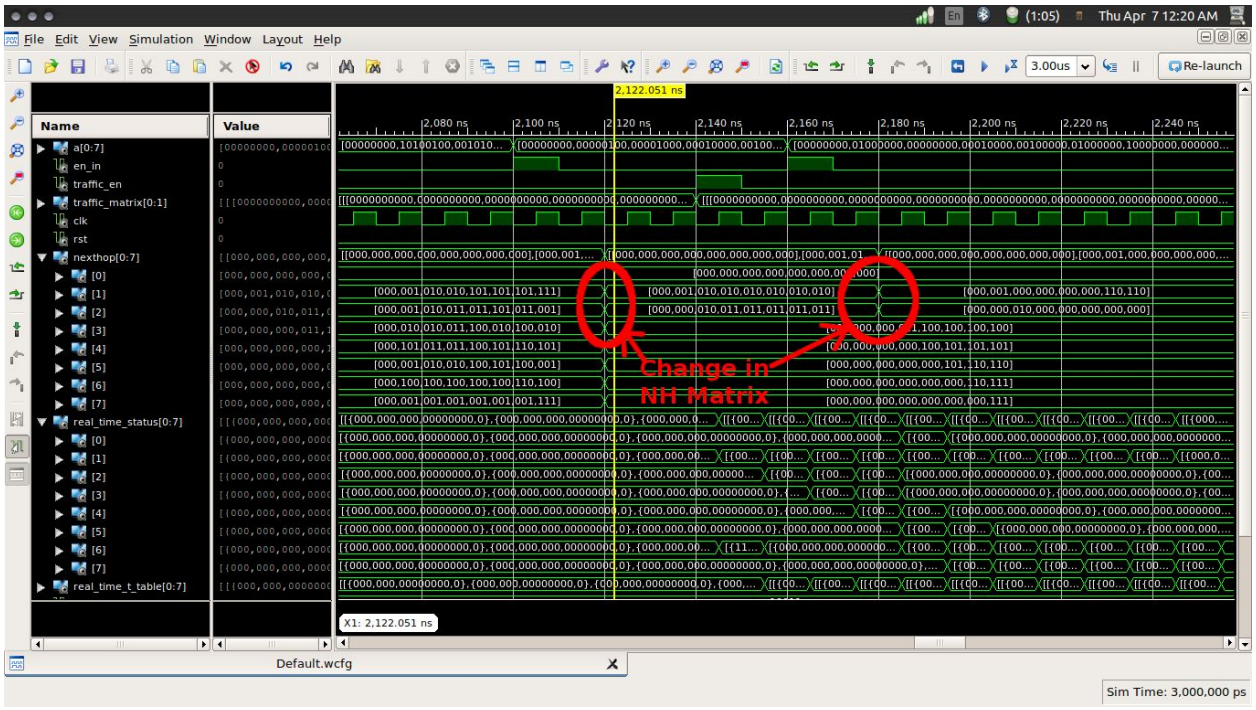


Fig-Latency_1

Here we can directly see the change of NH matrix (i.e. the next hop matrix) between 2 clock cycles.

The change in connectivity is already explained above. As the connectivity changes, we can see corresponding changes as in figure.

e.g. see the row 1 of NH matrix as (000,001,010,010,101,101,101,111). Thus the next hop for going to node 1 is 1, for node 2 is 2, node 3 is is 2, and so on.

This changes to (000,010,010,010,010,010,010,010).

Thus now the next hop for all destinations other than self, from node 1 are now node 2 due to the specific new adjacency matrix as in figure.

m) Latency and Statistics :

Here, we see the overall calculation of latency and jitter.

We are calculating latency for each path individually via the latency matrix. Also, we are calculating the total latency for entire network.

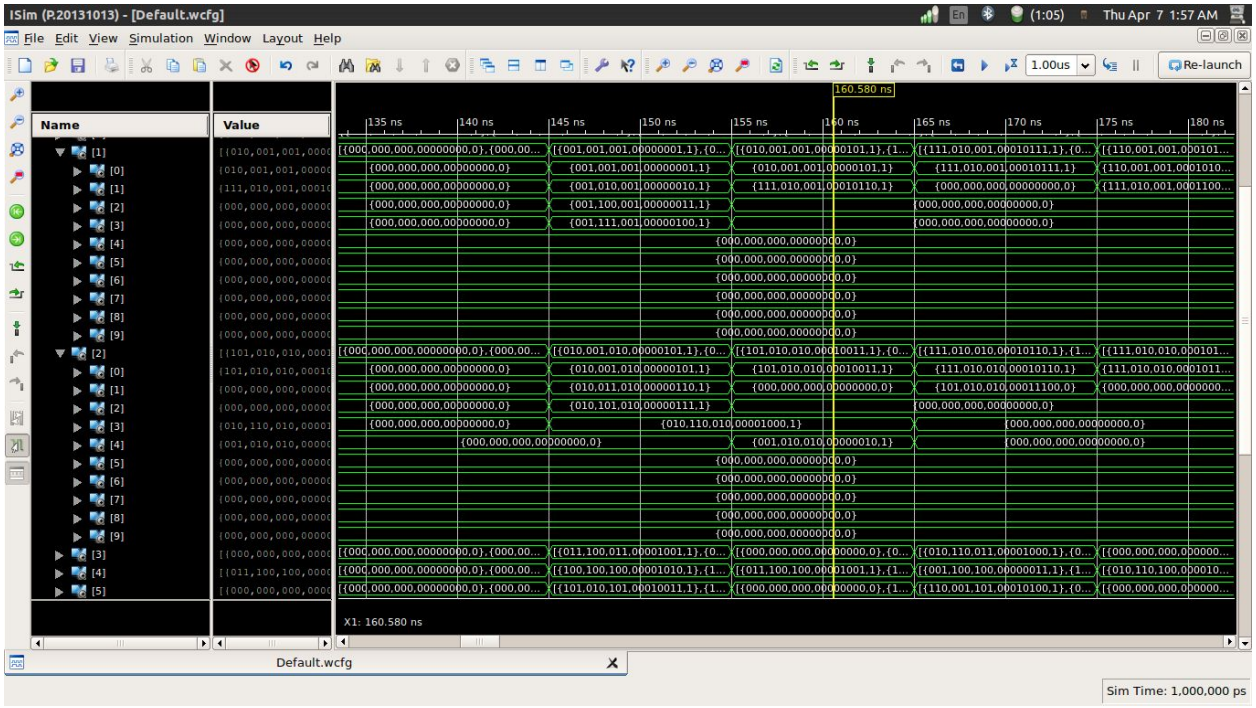


Fig-Latency_2

In this image, we see that as packet from node 1 destined to node 2 (ID 10 ,i.e. 2) reaches node in the next clock-cycle, the latency corresponding to pair of nodes 1 and 2 changes correspondingly from 0 to 1001.

Similarly there are changes in average end-to-end latency for each pair of nodes, as each new packet gets successfully delivered.
We however don't see any jitter in the present network, due to the specific formula for jitter calculation, which divides the change in latency by 16 and adds this to present jitter.
Hence this quantity is negligible for the network of size in this test case.Since we are using integers for these calculations, such minor changes get truncated and we see jitter to be zero throughout the simulation.

8. Using the current model for larger network size via generic arguments:

In order to scale the current simulation on 8 nodes (one of which is dummy), we change the values assigned to generic variables as follows:

1. Algorithm Block :
Lines :
line 61: change N_width to ceil(log N) where N is new number of nodes
line 62: change N to new number of nodes

2. my_package :
Lines:
line 15: change N_width as above
line 16: change N as above
line 17: change node_memory to any maximum buffer space required
line 18: change mid to maximum number of bits required to assign unique message ID for given size of traffic matrix
line 19: change k_max to whatever number of services are to be supported

3. Traffic Module :
Lines
line 40: change N as above
line 41: change numN to maximum number of nodes possible for this network
line 42: change N_wid to ceil of log numN value, which is needed to hold node numbers as logic_vectors
line 43: change mid as above
line 44: change k_max as above
line 45: change mem_w as width of memory for each node required (related to buffer size allowed for each node)

4. Overall_stats :
Line 132 : change N as above