

UNIVERSITY OF ALBERTA

Cost Adaptive OSPF Routing Evaluation

by

Iman Ghamari

A PROJECT DISSERTATION

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE IN INTERNETWORKING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

EDMONTON, ALBERTA

OCTOBER, 2010

© IMAN GHAMARI 2010

## **Abstract**

Traditionally, a single metric is used to calculate the shortest path to destination in OSPF. Network administrators typically set the OSPF cost metric manually for each connected interface in routers. This way of configuring OSPF in a network can lead to a very uneven traffic flow that can cause some links along some paths to become more congested compared to others. This problem is known as local congestion of network, owing to traffic aggregation.

When the utilization ratio of the links along the current shortest path is not high, then of course, the traditional OSPF packet forwarding is the best selection. However, when this current shortest path gets congested over time, the better selection may be to choose other paths whose costs are higher but their link utilization are lower. Therefore, modifying OSPF in such a way that it can dynamically detect that the current link utilization along the shortest path is higher than a certain threshold value and switch to a path with higher cost but the lower link utilization, results in controlling the local congestion, balancing the network traffic and hence improving QoS.

In this project, OSPF is modified so that it may dynamically adjust its interface's cost according to the link utilization of the interface's bandwidth, and as a result, the traffic load is distributed more evenly in a single-constraint routed OSPF network.

Consequences are improved QoS and resource utilization ratio.

### **Acknowledgements**

I would like to thank my supervisor Dr. Mike MacGregor, Professor and Chair of Department of Computing Science, for suggesting this interesting and challenging project and financially supporting me. I am thankful to him for the inspiration which kept me going through the project.

## Table of Contents

Abstract .....	1
Acknowledgements .....	2
Table of Contents .....	3
CHAPTER ONE: INTRODUCTION .....	5
1.1 OSPF .....	5
1.2 OSPF Cost .....	7
1.3 Cost Adaptive OSPF .....	9
CHAPTER TWO: ALGORITHM DESIGN .....	12
2.1 Hill Climbing Algorithm .....	12
2.2 Monitoring Subsystem .....	14
2.3 Cost Adaptive Subsystem .....	15
CHAPTER THREE: SIMULATION .....	18
3.1 Designing Network Topology .....	18
3.2 A Note on OPNET .....	19
3.3 NS-3 Design .....	21
3.3.1 Node .....	21
3.3.2 Channel .....	22
3.3.3 Net Device .....	23
3.3.4 Topology Helpers .....	23
3.3.5 Node Container .....	24
3.3.6 P2P Links .....	25
3.3.7 Net Device Container .....	26
3.3.8 Internet Stack .....	27
3.3.9 Assigning IP Addresses .....	28
3.3.10 LAN Design .....	30
3.3.11 Traffic Generation .....	32
3.3.12 Call Backs .....	34
3.3.13 CA-OSPF .....	37
3.3.14 Simulator .....	39
3.4 Test Case Results .....	40
3.5 Routing Oscillations .....	47
CHAPTER FOUR: LAB IMPLEMENTATION .....	52
4.1 Open Source Routers .....	52
4.2 Designing OSPF Network with Cisco Routers .....	52
4.3 Implementing CA-OSPF with Automated Scripts .....	54
4.4 SNMP, OIDs and MIB-II .....	56
4.5 Scripting the Monitoring Subsystem with Perl .....	57
4.6 Scripting the Cost Adaptive Subsystem with Expect .....	62
4.7 Test Case Results .....	66

CHAPTER FIVE: CONCLUSION.....74

REFERENCES .....75

## Chapter One: Introduction

### 1.1 OSPF

The Open Shortest Path First (OSPF) protocol, defined in RFC 2328, is an Interior Gateway Protocol used to distribute routing information within a single Autonomous System. OSPF protocol was developed due to a need in the internet community to introduce a high functionality non-proprietary Internal Gateway Protocol (IGP) for the TCP/IP protocol family. The OSPF protocol is based on link-state technology, which is a departure from the Bellman-Ford vector based algorithms used in traditional Internet routing protocols such as RIP.

OSPF uses a shortest path first algorithm in order to build and calculate the shortest path to all known destinations. The shortest path is calculated with the use of the Dijkstra algorithm. The algorithm by itself is quite complicated. This is a very high level, simplified way of looking at the various steps of the algorithm:

1. Upon initialization or due to any change in routing information, a router generates a link-state advertisement. This advertisement represents the collection of all link-states on that router.
2. All routers exchange link-states by means of flooding. Each router that receives a link-state update should store a copy in its link-state database and then propagate the update to other routers.
3. After the database of each router is completed, the router calculates a Shortest Path Tree to all destinations. The router uses the Dijkstra algorithm in order to

calculate the shortest path tree. The destinations, the associated cost and the next hop to reach those destinations form the IP routing table.

4. In case no changes in the OSPF network occur, such as cost of a link or a network being added or deleted, OSPF should be very quiet. Any changes that occur are communicated through link-state packets, and the Dijkstra algorithm is recalculated in order to find the shortest path.

The algorithm places each router at the root of a tree and calculates the shortest path to each destination based on the cumulative cost required to reach that destination. Each router will have its own view of the topology even though all the routers will build a shortest path tree using the same link-state database.

A router periodically advertises its routing information, also called link state, by flooding. The flooding makes sure that all the routers in the network have the same database of the routing information i.e. link state database. Link state is also advertised whenever the state of a router changes. The unit of these data describing the local state of routers or network is called Link State Advertisement (LSA). RFC 2328 defines five types of LSAs namely: Router-LSA (Type 1), Network-LSA (Type 2), Summary LSA (Type 3 or 4) and AS-External-LSA (Type 5). In this project Router-LSA is the LSA adopted to propagate the link cost changes. This LSA is originated by all the routers in the network and describes the collected state of the router's interface. Updating a new LSA in link-state database as result of flooding or router's self-generated LSA, may invoke recalculating the OSPF routing table and hence the change of best paths. The content of

the new LSA is compared to the current instance in link-state database, if there is no difference; there is no need to recalculate the routing table. But, if the contents are different; parts of the routing table may be recalculated based on the new LSA type. For Router-LSAs, the entire routing table must be recalculated and reconfigured.

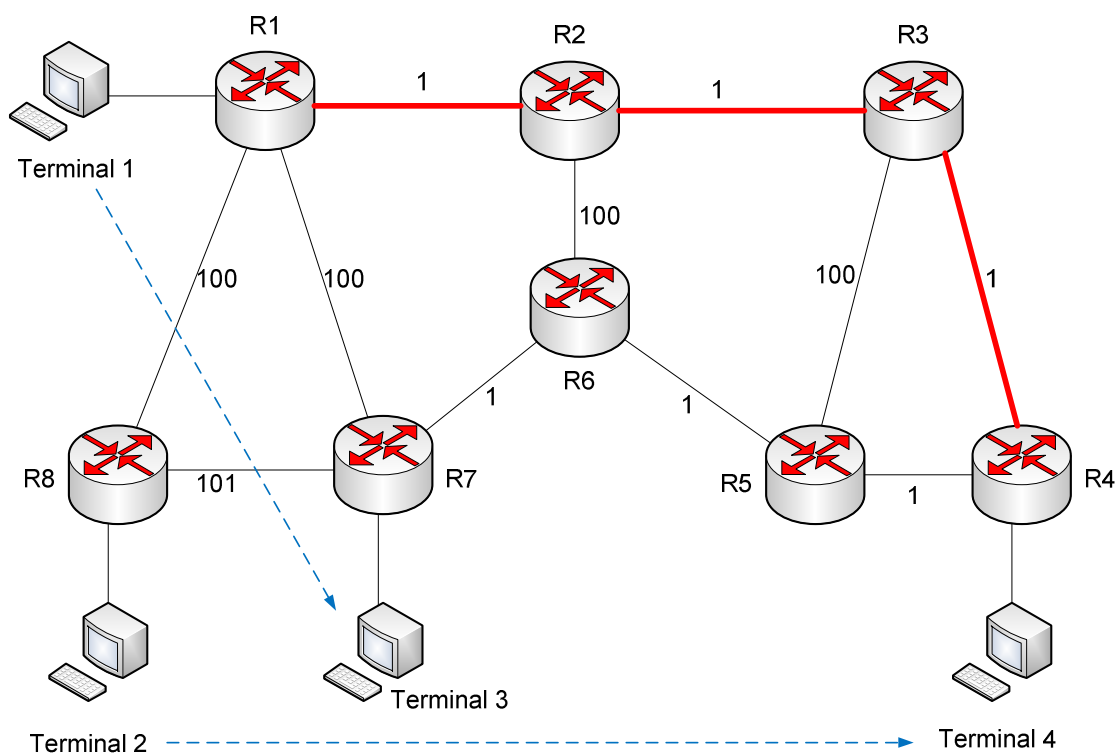
## 1.2 OSPF Cost

The cost (also called metric) of an interface in OSPF is an indication of the overhead required to send packets across a certain interface. The cost of an interface is inversely proportional to the bandwidth of that interface. A higher bandwidth indicates a lower cost. Obviously there is more overhead (higher cost) and time delays involved in crossing a 56k serial line than crossing a 10M Ethernet line.

In Cisco routers, by default, the cost of an interface is calculated based on the bandwidth, but the cost of interface can be forced with the **ip ospf cost <value>** interface sub-configuration mode command. Network administrators usually set the link costs manually during the network setup. Designing a bad cost scheme is root of the problem that is to be addressed in this project.

Consider the OSPF network shown in figure 1-1. There are two traffic flows in the network each at 50Mbps. The first one is from Terminal 1 destined to Terminal 3 and the second one is from Terminal 2 destined to Terminal 4. Let say the links connecting the routers are 100BASE-T links. By looking at the figure we can say that traffic-1 could flow the path along routers R1-R7 and traffic-2 along R8-R1-R2-R3-R4.





**Figure 1-1 An OSPF network with a bad cost (metric) setup**

But, in this network a bad scheme has been used for interface cost assignments by the network manager. This bad manual assignment of interface costs has caused the traffic-1 from terminal 1 to terminal 3 to flow along the path R1-R2-R3-R4-R5-R6-R7. Therefore this traffic doubles the load on path R1-R2-R3-R4 as traffic-2 is flowing through the same path too. As a result of this, path R1-R2-R3-R4 gets overloaded (shown in red color in figure) while some other paths like R1-R7 are totally unused. In this situation, the entire routing performance is affected and it can be observed how traffic has been jammed on a particular path while other better paths exist.

Measurements from the Internet indicate that for almost 80% of the taken traffic paths, alternative paths exist which offer higher bandwidth and lower round-trip delay (S. Savage et al. 1999). Major reason these superior quality alternative paths being unused is due to poor links' cost assignments. Network managers normally set the value of link costs proportional to the link's physical distance between network's nodes or according to some rules such as link priority which might be prone to human mistakes. The standard heuristic recommended by Cisco is to set the link cost inversely proportional to the link capacity. But, often the main goal of the network administrator is to avoid the traffic congestion in the entire network and to achieve a better utilization of network resources.

### **1.3 Cost Adaptive OSPF**

All the attempts in OSPF for choosing the best path based on static links, assume that link costs remain unchanged during the network operation and hence the selected best path remains the best choice regardless of the current network conditions such as current link's load and availability of other better paths. A direct consequence of this is the fact that some paths known as best paths to some sources for reaching some destinations always get overloaded while other paths with less traffic jam may exist that could offer better performance.

A solution for this problem could be an enhanced routing protocol that includes a monitoring subsystem and a cost adaptive subsystem. The monitoring component monitors the network in real time and if the best paths are overloaded, notifies the core

algorithm of the cost adaptive component of the protocol which in turn dynamically associates the link costs to link congestion levels and therefore establishes the *real* best paths in the network in real time. The literature survey part of this project clearly indicated that cost adaptive routing is well known to improve the network performance by increasing its throughput and possibly lowering the end-to-end packet delay (D. W. Glazer et al. 1990, A. Khanna et al. 1989). The best application of cost adaptive OSPF could be in Internet, but, this kind of routing protocol has been largely abandoned in practice due to a problem associated with routing oscillation that will be discussed in section 3.5 of chapter 3.

In this dissertation, the concept of effective bandwidths for links will be introduced to define a new link cost for the proposed cost adaptive OSPF and its performance will be evaluated under overloading multiple traffics. In essence, the effective bandwidth for links is a link utilization parameter which allows us to define a new concept of “Network Quality” different than QoS for connections. Effective bandwidth estimations used in adaptive OSPF let us know the network load considering quality constrains desired for links.

As explained in section 1.2, multiple traffics in a network can easily overload some links if the initial cost assignments are susceptible to cause this situation. A network running cost adaptive OSPF as its routing protocol should be able to converge eventually in the event of overloaded links. In other words, the cost adaptive OSPF

should be able to adjust the link costs with respect to the live traffic load on links and hence cause an even distribution of traffic on the network.

In next chapter, detailed steps of our algorithm design will be explained. In chapter 3, first the proposed algorithms will be simulated in ns-3 and run under multiple overloading traffics and then, in chapter 4, the functionality of the core algorithm will be implemented in real Cisco routers in the internetworking lab to measure and analyze performance against projected optimization targets obtained from the simulation.

## Chapter Two: Algorithm Design

### 2.1 Hill Climbing Algorithm

Hill climbing algorithm is a mathematical optimization algorithm which belongs to a larger family of algorithms known as *local search* algorithms. It's relatively a simple algorithm to implement that usually makes it a standard first choice. There are other more advanced algorithms as well but for the purpose of this project, hill climbing is a better choice as for real-time systems it can get a better solution in a limited time.

Hill climbing can be used in problems that have many solutions (called search space). Different solutions in search space usually have different values. Hill climbing starts with a random or presumed solution which is usually a poor solution and then makes small changes (its value is called hill climbing step size) to it to generate the next neighbour solution in each iteration. If the newly generated solution is better than the current one then it replaces the current solution. This iteration continues until there are no more better solutions available for the problem. At this point it's safe to say that hill climbing has generated a solution close to optimal but it's not guaranteed.

Mathematically, hill climbing attempts to maximize (or minimize) a function  $f(x)$ , where  $x$  are discrete states. These states are typically represented by vertices in a graph, where edges in the graph encode nearness or similarity of a graph. Hill climbing will follow the graph from vertex to vertex, always locally increasing (or decreasing) the value of  $f$ , until a local maximum (or local minimum)  $x_m$  is reached.

In this project, to design the core algorithm of our adaptive OSPF protocol, a variation of hill climbing algorithm is used. In this variation algorithm, each interface of each router in the network is sensed periodically for its link utilization ratio. In each iteration, the link utilization ratio of the link attached to that interface is measured and if it's below a first threshold or above a second threshold, then the link cost of that interface decreases or increases respectively. Here, the value by which the interface cost is increased or decreased is the step size of our variation hill climbing algorithm. The iteration continues until the measured interface link utilization ratio is no longer above or below the two thresholds i.e. it's between the two threshold values. At this point, a *local* optimal solution is found and as long as the thresholds' conditions are satisfied on future iterations, the current local optimal solution is valid.

In section 1.3, we briefly introduced the two components of our design. The monitoring subsystem is responsible for periodical sensing of links to measure their *live* link utilization. Based on the measured values, the appropriate part of the core algorithm in the cost adaptive subsystem will be invoked. The cost adaptive subsystem works closely with monitoring subsystem to change the interface costs based on the value obtained from the monitoring subsystem. These two components compose the core algorithm of cost adaptive property of CA-OSPF and enable the protocol to change its costs in real time with respect to network's links utilization ratios in order to distribute the traffic more evenly in the network and enhance the performance. In the next two sections we will have a closer look at the details inside the design of these two components.

## 2.2 Monitoring Subsystem

The monitoring component of CA-OSPF includes the algorithms to check the *live* link utilization ratios of the routers' interfaces on regular basis. The monitoring subsystem can be either centralized or distributed. In centralized monitoring, a single node in the network is responsible for links' utilization measurements. In this approach, the centralized node logs in to each router in the OSPF network remotely and measures the link utilization of all of its connected interfaces. In distributed approach however, the monitoring algorithm is implemented inside each router device separately. Therefore, the OSPF process in each router is responsible to measure the link utilization ratio of its connected interfaces periodically. There are some important time synchronization issues in the centralized approach that will be discussed in chapter 4. We will use both of these approaches in this project. The distributed approach will be used in the simulation since the ns-3 simulated routers are open source and we're able to write our code inside each router in the network. In real implementation with Cisco routers however, we're forced to use the centralized approach as Cisco boxes and specially their IOS are not open source and as a result we can not run our code as a part of OSPF process inside their IOS. More details on this and the actual implementation will be covered in chapter 4.

Now there is an important question regarding the monitoring component and that is how often this component should be run during the course of CA-OSPF? To answer this question there are two factors that should be considered. The first factor is the network convergence time and the second one is the resulted overhead in the network. If the time between runs of monitoring component is small, then network will converge

faster; however there will be more overhead generated as a result of procedure calls (the cost adaptive component might be called as the result of the monitoring component call). In contrast, if the time between monitoring component calls is large, then there will be less overhead generated but probably network is going to converge slower. So, as it can be seen there is a trade-off here for choosing the best value for the frequency of calls to the monitoring subsystem. This time value can be optimized by comparing a series of test results.

### 2.3 Cost Adaptive Subsystem

The key idea of CA-OSPF is to adjust router's interface cost dynamically according to the bandwidth utilization ratio of the interface. In this way, the shortest path is the best path. According to the bandwidth utilization ratio  $U$  ( $0 \leq U \leq 100\%$ ) of router's interface obtained from the monitoring subsystem, we set two threshold values  $\Theta_1$  and  $\Theta_2$  in the same range as  $U$ , such that  $\Theta_2$  is greater than  $\Theta_1$ . Each time the monitoring subsystem measures the link utilization ratio of the router's interface, the interface state must be one of the following three states:

- (I) Under-used state ( $0 \leq U \leq \Theta_1$ )
- (II) Middle state ( $\Theta_1 < U < \Theta_2$ )
- (III) Over-used state ( $\Theta_2 \leq U \leq 100\%$ )

Let also say that initial interface cost set by the network manager is  $C_0$  and the dynamic cost is  $C$  ( $C \geq C_0$ ).



Initially  $C$  is set equal to  $C_0$ . Monitoring subsystem checks the interface link utilization periodically and depending on the link utilization state, cost adaptive component takes one of the following actions:

- (1) When the interface's link utilization is in the over-used state, the interface's cost  $C$  is increased by  $\Delta$ .
- (2) When the interface's link utilization is in the middle state, the interface's cost is kept unchanged.
- (3) When the interface's link utilization is in the under-used state, first  $C$  is compared to  $C_0$ . If they are equal, do nothing; if the current cost  $C$  is greater than the initial cost  $C_0$ , then  $C$  is decreased by  $\Delta$ , but  $C$  can never become less than the initial cost  $C_0$ .

Delta ( $\Delta$ ) is the *step size* in our hill climbing algorithm. Similar to the problem that how often the monitoring component should be run, delta size also is an important issue to consider. Larger delta size could lead the network to converge faster however a problem arises when delta is set to a very large value. Consider a situation, where an interface is in over-used state and delta size is very large. Here, the interface cost increases by delta value but since delta is very large, the resulted interface cost becomes too large and this may impact the entire network negatively in terms of proper traffic distribution. Hence, delta should not be set to a very large value indefinitely. We have a trade-off here to consider and in fact the delta size could be optimized by comparing a series of test results.

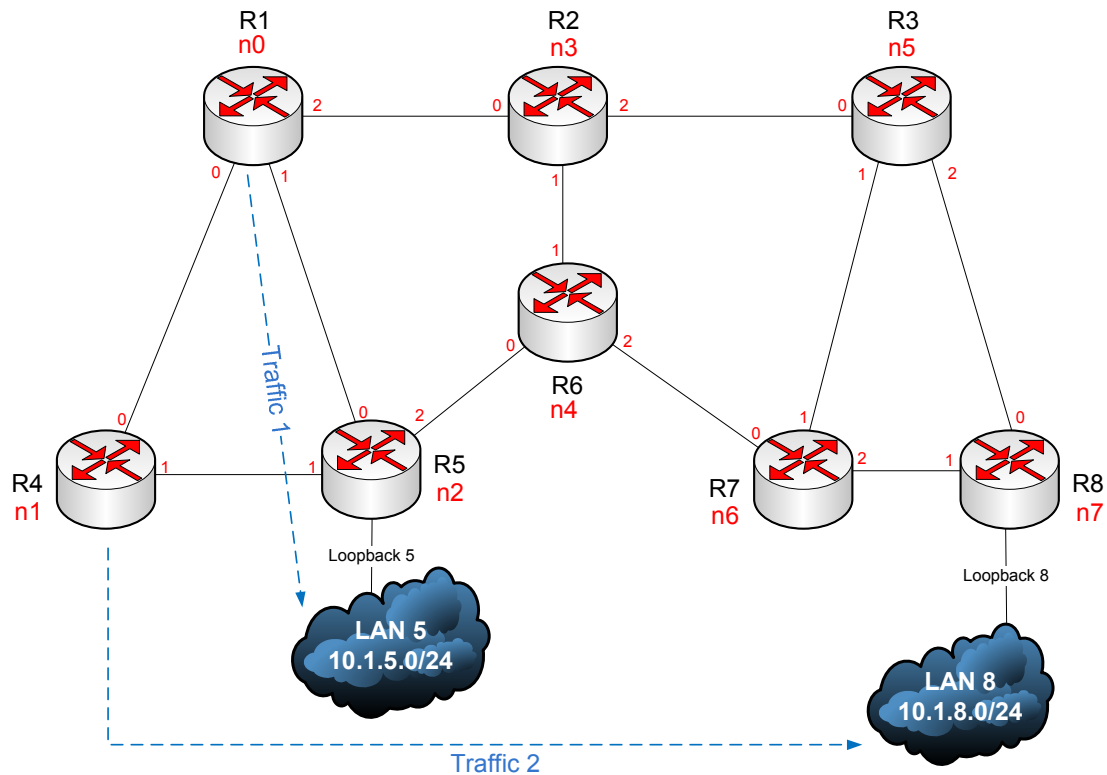
Cost adaptive subsystem also could be implemented using a centralized or a distributed approach. Details of these approaches are same as the monitoring subsystem explained in the previous section, and in fact in this project, monitoring and cost adaptive components are in complete agreement in terms of centralized or distributed approach used.

## Chapter Three: Simulation

### 3.1 Designing Network Topology

The network topology that is designed for the simulation is similar to the OSPF network diagram shown in chapter 1. As shown in figure 3-1, we have eight routers which construct the backbone of our network for the simulation. There are two LANs named as LAN 5 and LAN 8 in the network which are connected to routers R5 and R8 respectively. In the actual code, LAN 5 and LAN 8 have been implemented as loop-back interfaces to routers R5 and R8 and each have been assigned the address blocks 10.1.5.0/24 and 10.1.8.0/24 respectively. Routers in the backbone network have IP addresses from the 10.1.1.0/24 address block, but for neatness of the figure, individual interface addresses have been omitted. There are two traffic flows in the network; traffic-1 from router R1 to a destination in LAN 5 and traffic-2 from router R4 to a destination in LAN 8.

Since the simulation has been coded in ns-3, each router in the network is a node (in ns-3 terminology, every network device is a node). Therefore, each router in the network is also named by the letter “n” and a number. For example, router R1 is node n0 and router R2 is node n3 and so on. The numbers shown on each router’s interface indicate the interface number of that interface in that router. Links are bidirectional and numbered by combining node number with interface number. For example, the link from n3 to n5 is link 32 because it is attached to node 3, interface 2. The link in the reverse direction, from n5 to n3 is link 50 because it is attached to node 5, interface 0.



**Figure 3-1 OSPF network topology designed for simulation**

Here, we aren't using any extra multipliers to differentiate between links of different types (e.g. terrestrial vs. satellite) so what we're going to get are minimum-hop solutions.

### 3.2 A Note on OPNET

This project was first attempted to be simulated by using OPNET simulator. After creating the network topology and writing some significant amount of code inside the `ospf_v2` process of routers, the coding process encountered some difficulties at the implementation stage of monitoring subsystem.

In OPNET, link utilization statistic has two possible meanings, depending on the collection method used:

1. When collected by *statistic probe*, this statistic is a measure of the consumption to date of an available channel bandwidth. This statistic is expressed as a percentage, with 100 indicating full usage.
2. When collected by *statistic wire*, this statistic has one of two values: 0 and 100. Instantaneous utilization is not accumulated. In other words, if we utilize a statistic wire to convey the utilization statistic value to our module/processor, it'll be of value 0 or 100.

Regarding statistic probe, the returned value is an accumulated consumption rate to date. What indeed is our criteria here is that the value which is returned at the current simulation time, let say  $T_i$ , should be an accumulated consumption rate to date which is calculated from the beginning of the simulation to the current simulation time  $T_i$ . To get this, we need the API to read this static probe at the simulation time  $T_i$ .

After some research and consulting with OPNET technical support representatives, it was discovered that the statistic collected via statistic probe could be only viewed during simulation (via Live Stats page of the Simulation Progress dialog box) or using the result viewer at the end of the simulation. There isn't actually any API to read out the statistic value amid the DES.

This obstacle marked the end of OPNET use in this project and it was decided to carry out the simulation part of this project in ns-3 network simulator which was a more powerful choice for what we were looking for. The power of ns-3 lies in its pure C++ structure that actually gives the developer the freedom of coding virtually anything, which is essential to implement those low level concepts that are not otherwise possible with simulators like OPNET.

### **3.3 NS-3 Design**

The ns-3 simulator is a discrete-event network simulator for internet systems targeted primarily for research, development and educational use. It's very popular for its extensibility due to its open source model. Being an open source project written in C++, researchers and developers can design and develop all their requirements by using ns-3's rich set of classes or by deriving new classes from existing classes. In fact, in ns-3 every entity and object is an instance of these classes.

In the following sections, we will step through designing our simulation scenario for the network shown in Figure 3-1. We will only examine the example codes for the subnet R1-R4. The remaining subnets have been coded similarly.

#### **3.3.1 Node**

In ns-3 the basic computing device abstraction is called the node. This abstraction is represented in C++ by the class Node. The Node class provides methods for managing the representations of computing devices in simulations. We should think of a Node as a

computer to which we will add functionality. One adds things like applications, protocol stacks and peripheral cards with their associated drivers to enable the computer to do useful work. The same basic model is used in ns-3.

We start coding our simulation by creating eight Nodes. These Nodes represent our routers R1 to R8 shown in Figure 3-1. The NodeContainer helper will be used for creating our nodes and connecting them as explained in section 3.3.5.

### ***3.3.2 Channel***

In real world, in order to connect our routers, we use communication channels. In the simulated world of ns-3, one connects the Node to an object representing a communication channel. Here the basic communication sub-network abstraction is called the channel and is represented in C++ by the class *Channel*.

The Channel class provides methods for managing communication sub-network objects and connecting nodes to them. Channels may also be specialized by developers in the object oriented programming sense. A Channel specialization may model something as simple as a wire. The specialized Channel can also model things as complicated as a large Ethernet switch, or three-dimensional space full of obstructions in the case of wireless networks. We will use specialized version of the Channel called *PointToPointChannel* in our simulation.

### ***3.3.3 Net Device***

In real world, when we want to connect a computer or a router to network, we have to first install a Network Interface Card (NIC) inside our box. A NIC will not work without a software driver to control the hardware. NICs are controlled using network device drivers collectively known as *net devices*. In Unix and Linux these net devices are referred to by names such as eth0. In ns-3 the net device abstraction covers both the software driver and the simulated hardware. A net device is “installed” in a Node in order to enable the Node to communicate with other Nodes in the simulation via Channels. Just as in a real computer, a Node may be connected to more than one Channel via multiple net devices.

The net device abstraction is represented in C++ by the class NetDevice. The NetDevice class provides methods for managing connections to Node and Channel objects; and may be specialized by developers in the object-oriented programming sense. We will use the specialized version of the NetDevice called PointToPointNetDevice in this simulation. Just as an Ethernet NIC is designed to work with an Ethernet network, the PointToPointNetDevice is designed to work with a PointToPointChannel.

### ***3.3.4 Topology Helpers***

In a real network, we find host computers with added (or built-in) NICs. In ns-3 we would say that Nodes are with attached NetDevices. In a large simulated network we will need to arrange many connections between Nodes, NetDevices and Channels. Since connecting NetDevices to Nodes, NetDevices to Channels, assigning IP addresses, etc.,



are such common tasks in ns-3; topology helpers are provided to make this as easy as possible. For example, it may take many distinct ns-3 core operations to create a NetDevice, add a MAC address, install that net device on a Node, configure the node's protocol stack, and then connect the NetDevice to a Channel. Even more operations would be required to connect multiple devices onto multipoint channels and then to connect individual networks together into internetworks. Topology helper objects are provided that combine those many distinct operations into an easy to use model for convenience. In the next few sections we will see how topology helpers make our life easier to code our simulation.

### ***3.3.5 Node Container***

We use various topology helpers in this simulation. Initially to create our nodes and connecting them via communication channels, we use NodeContainer helper. Let's have a look at the code for creating routers R1 and R4:

```
NodeContainer R14;  
R14.Create (2);
```

The NodeContainer topology helper provides a convenient way to create, manage and access any Node objects that we create in order to run a simulation. The first line above just declares a NodeContainer which we call R14. The second line calls the Create method on the nodes object and asks the container to create two nodes which actually corresponds to our nodes R1 and R4 in Figure 3-1. As described in the ns-3 Doxygen, the



The first line basically instantiates a `PointToPointHelper` object on the stack. From a high-level perspective the second line tells the `PointToPointHelper` object to use the value “100Mbps” as the “DataRate” when it creates a `PointToPointNetDevice` object. From a more detailed perspective, the string “DataRate” corresponds to what is called an Attribute of the `PointToPointNetDevice`. Similar to the “DataRate” on the `PointToPointNetDevice` there is a “Delay” Attribute associated with the `PointToPointChannel`. The final line tells the `PointToPointHelper` to use the value “6560ns” (6560 nanoseconds) as the value of the transmission delay of every point to point channel it subsequently creates.

### ***3.3.7 Net Device Container***

At this point, we have a `NodeContainer` that contains two nodes; R1 and R4. We have a `PointToPointHelper` that is primed and ready to make `PointToPointNetDevices` and wire `PointToPointChannel` objects between them. Just as we used the `NodeContainer` topology helper object to create the Nodes for our simulation, we will ask the `PointToPointHelper` to do the work involved in creating, configuring and installing our devices for us. We need to have a list of all of the `NetDevice` objects that are created, so we use a `NetDeviceContainer` to hold them just as we used a `NodeContainer` to hold the nodes we created. The following two lines of code will finish configuring the devices and channel:

```
NetDeviceContainer R14D;  
R14D = p2p.Install (R14);
```

The first line declares the device container mentioned above and the second does the heavy lifting. The `Install` method of the `PointToPointHelper` takes a `NodeContainer` as a parameter. Internally, a `NetDeviceContainer` is created. For each node in the `NodeContainer` (there must be exactly two for a point-to-point link and here we have R1 and R4) a `PointToPointNetDevice` is created and saved in the device container. A `PointToPointChannel` is created and the two `PointToPointNetDevices` are attached. When objects are created by the `PointToPointHelper`, the `Attributes` previously set in the helper are used to initialize the corresponding `Attributes` in the created objects.

After executing the `p2p.Install (R14)` call we will have two nodes R1 and R4, each with an installed point-to-point net device and a single point-to-point channel between them. Both devices will be configured to transmit data at 100 megabits per second over the channel which has a 6560 nanosecond transmission delay.

### ***3.3.8 Internet Stack***

We now have nodes and devices configured, but we don't have any protocol stacks installed on our nodes. The next two lines of code will take care of that:

```
InternetStackHelper stack;  
stack.Install (R14);
```

The `InternetStackHelper` is a topology helper that is to internet stacks what the `PointToPointHelper` is to point-to-point net devices. The `Install` method takes a

NodeContainer as a parameter. When it is executed, it will install an Internet Stack (TCP, UDP, IP, etc.) on each of the nodes (R1 and R4 here) in the node container.

### ***3.3.9 Assigning IP Addresses***

Next we need to associate the devices on our nodes with IP addresses. We provide a topology helper to manage the allocation of IP addresses. The only user-visible API is to set the base IP address and network mask to use when performing the actual address allocation (which is done at a lower level inside the helper). The next two lines of code from my actual code declare an address helper object and tell it that it should begin allocating IP addresses from the network 10.1.1.0 using the mask 255.255.255.252 to define the allocatable bits:

```
Ipv4AddressHelper address;  
address.SetBase ("10.1.1.0", "255.255.255.252");
```

By default the addresses allocated will start at one and increase monotonically, so the first addresses allocated from this base will be 10.1.1.1/30, followed by 10.1.1.2/30. The low level ns-3 system actually remembers all of the IP addresses allocated and will generate a fatal error if the same address accidentally caused to be generated twice (which is a very hard to debug error, by the way).

The next line of code performs the actual address assignment:

```
Ipv4InterfaceContainer R14I = address.Assign (R14D);
```

In ns-3, the association between an IP address and a device are made using an Ipv4Interface object. Just as we sometimes need a list of net devices created by a helper for future reference, we sometimes need a list of Ipv4Interface objects. The Ipv4InterfaceContainer provides this functionality.

For setting the base address for our next set of NetDevices (in subnets other than R1-R4), we can write a code like this:

```
address.SetBase (address.NewNetwork() ,"255.255.255.252");
```

The address helper allocated IP addresses based on a given network number and initial IP address. In order to separate the network number and IP address parts, SetBase was given an initial network number value, a network mask and an initial address base. NewNetwork() method of Ipv4AddressHelper class increments the network number and resets the IP address counter to the last base value used. Therefore the next call to the Assign method of address object, will assign addresses 10.1.1.5/30 and 10.1.1.6/30 to the two NetDevices passed to it.

Now we have a point-to-point network built, with stacks installed and IP addresses assigned. What we need at this point are applications to generate traffic, but

before going to explanation of traffic generation codes, let's quickly have a look at our codes to implement LANs.

### ***3.3.10 LAN Design***

I mentioned that we design our LANs as loopback addresses. First we define our two loopback addresses:

```
Ipv4InterfaceAddress loo5 ("10.1.5.0", "255.255.255.0");
Ipv4InterfaceAddress loo8 ("10.1.8.0", "255.255.255.0");
```

Ipv4InterfaceAddress is a class to store IPv4 address information on an interface. We create two objects loo5 and loo8 out of this class and send IPv4 address and the mask to its constructor. The class constructor looks like this:

```
Ipv4InterfaceAddress (Ipv4Address local, Ipv4Mask mask)
```

We assigned the address 10.1.5.0/24 and 10.1.8.0/24 to loo5 and loo8 respectively. We then attach LAN5 and LAN8 as loopback addresses to routers R5 and R8 respectively. Here we just show the code for attaching loo5 to R5:

```
(pair[2][1].first)->AddAddress (0, loo5);
```

We're not going to go inside the details of my actual coding strategies in this documentation, but just to have an understanding of the above snippet, `pair[2][1]` is an element of a 2D array, that could be defined by a line of code like this:

```
std::pair<Ptr<Ipv4>,uint32_t> pair[2][1] = R45I.Get(1);
```

The *Get* method of `Ipv4InterfaceContainer` class is actually defined as:

```
std::pair<Ptr<Ipv4>, uint32_t>
ns3::Ipv4InterfaceContainer::Get (uint32_t i) const
```

It returns the `std::pair` of a `Ptr<Ipv4>` and interface stored at the location specified by the index *i*. Therefore, `R45I.Get(1)` gets the mentioned pair corresponding to router R5 (R45I contains R4 and R5 `Ipv4Interface` objects). Now, we can understand that `pair[2][1].first` is actually referring to `Ipv4` object belonging to R5. This `Ipv4` object has a method called `AddAddress` that is defined as below according to ns-3 Doxygen:

```
virtual bool
ns3::Ipv4::AddAddress ( uint32_t interface,
                       Ipv4InterfaceAddress address
                       ) [pure virtual]
```



Interface number 0 of an Ipv4 interface corresponds to a loopback address, therefore, (pair[2][1].first)->AddAddress(0, loo5); is actually assigning the loopback address loo5 (10.1.5.0/24) to interface number 0 of Ipv4 object of router R5.

### ***3.3.11 Traffic Generation***

Another one of the core abstractions of the ns-3 system is the Application. We need applications to generate traffic in our simulated network. In this simulation we use two specializations of the core ns-3 class Application called OnOffApplication and PacketSink to generate traffics. Just as we have in our previous explanations, we use helper objects to help configure and manage the underlying objects. Here, we use OnOffHelper and PacketSinkHelper objects to make our lives easier.

As shown in Figure 3-1, we have two traffic flows in our network. Therefore we need to create two instances of OnOffApplication and two instances of PacketSink. OnOffApplication serves as our source (traffic generator) while PacketSink is our destination (sink). Here, we just go through implementation of traffic-1 which flows from R1 to LAN5. The same strategy is used to implement the second traffic.

We first create an object of OnOffApplication to define our source:

```
OnOffHelper onoff ("ns3::UdpSocketFactory",
                  InetSocketAddress ("10.1.5.0", 8));
```

The first parameter passed to the object constructor is the name of the protocol to use to send traffic by the applications. This string identifies the socket factory type used to create sockets for the applications. The second parameter passed to the object constructor is the address of the remote node to send traffic to. The `ns3::InetSocketAddress` class is an Inet address class which is similar to `inet_sockaddr` in the BSD socket API. i.e., this class holds an `Ipv4Address` and a port number to form an ipv4 transport endpoint. Here our destination is port 8 of any node in LAN5.

The following three lines of code then set the attributes of our source traffic generator:

```
onoff.SetAttribute ("OnTime",
                    RandomVariableValue (ConstantVariable (1)));
onoff.SetAttribute ("OffTime",
                    RandomVariableValue (ConstantVariable (0)));
onoff.SetAttribute ("DataRate", StringValue ("40Mbps"));
```

These codes set up a continuous traffic flow at the data rate of 40Mbps. What we need now is to install this application on our source-1 which is router R1. The following three lines take care of that:

```
ApplicationContainer src = onoff.Install (R15.Get (0));
src.Start (Seconds (start));
src.Stop (Seconds (stop));
```

ApplicationContainer is a class that holds a vector of ns3::Application pointers. The first line actually installs the application that we created and defined previously on R1. The second and third lines set the times that when the application should start and stop. These times are declared in the program by the variables start and stop.

Now following the same analogy, we define and install our sink application on LAN5 which is in fact the loopback interface of router R5:

```
PacketSinkHelper sink ("ns3::UdpSocketFactory",
                      InetAddress ("10.1.5.0", 8));
ApplicationContainer dest = sink.Install (R15.Get (1));
dest.Start (Seconds (start));
dest.Stop (Seconds (stop));
```

The only difference here is that the second parameter passed to the PacketSinkHelper object is the address of the sink itself which is port 8 of any machine in LAN5.

### ***3.3.12 Call Backs***

In this section we are going to see how the power of C++ in ns-3 and its open source nature is beautifully used to overcome the bottleneck that prevented us from using OPNET for our simulation. In section 3.2, we explained how OPNET gave us a hard time to implement and simulate the monitoring component of our hill climbing algorithm.

Here we see how the C++ concept of Callback is used to actually measure the live link utilization of all the interfaces of all the routers in our network in real time.

The goal of the Callback system in ns-3 is to allow one piece of code to call a function (or method in C++) without any specific inter-module dependency. This ultimately means we need some kind of indirection – we treat the address of the called function as a variable. This variable is called a pointer-to-function variable. The relationship between function and pointer-to-function pointer is really no different than that of object and pointer-to-object.

Again we are not going to present my actual coding strategies in this documentation, but for the sake of completeness of this section, we briefly cover the high level design. Also it's worthy to mention that the Callback system in ns-3 is a part of ns-3's Tracing System which by itself is a huge topic. Obviously covering the Tracing System of ns-3 is not in the scope of this documentation. Therefore, readers are advised to make themselves familiar with the concepts of trace source and trace sinks before reading the following paragraphs. Ns-3 documentations and manuals have an extensive coverage on these topics.

In this simulation we are using point-to-point links. Therefore our routers have point-to-point interfaces. On every point-to-point interface, there is a trace source called MacTx which indicates a packet has arrived for transmission by that interface while simulation is under run. We can connect this trace source to a trace sink specially coded

to use the parameters passed by the trace source. We can use the `Config::Connect` subsystem in order to connect `MacTx` trace source to our specialized trace sink. It turns out that the parameter passed by `MacTx` of each interface to our trace sink is a pointer to the `Packet` that is about to be sent out of that particular interface. Please note that each pointer is in fact pointing to an object of class `Packet`. This class has a method called `GetSize()` which can help us to get what we want. Therefore, we code our trace sink `TraceMacTx` something like this:

```
uint64_t p2pBytes = 0;
void TraceMacTx (std::string context,
                 Ptr<const Packet> txPacket) {
p2pBytes += txPacket->GetSize (); }
```

Of course, the way this is presented here, it will add up all bytes received on all point-to-point interfaces. In my actual code there's a strategy to find out which interface was sending the trace request.

Finally, after each time we use `p2pBytes` of each interface to calculate its link utilization, we zero it and then we start accumulating a new count. When the event signalling the end of a monitoring interval fires, we take the totals and divide by the length of the interval to get the byte rates. After zeroing the counts, we use the byte rates in the hill climber to adjust the link costs.

### 3.3.13 CA-OSPF

So far we saw how we designed and implemented our OSPF network. Now we run the OSPF process on our routers by a single line of code:

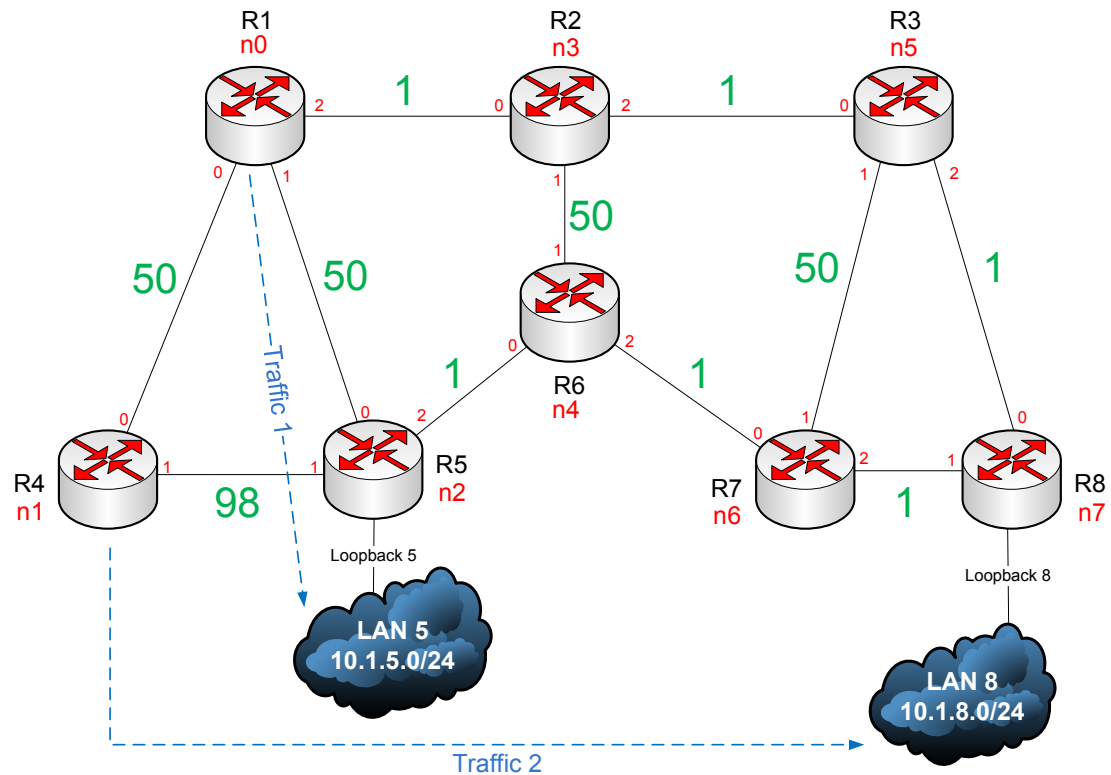
```
Ipv4GlobalRoutingHelper::PopulateRoutingTables();
```

The `PopulateRoutingTables()` method of `Ipv4GlobalRoutingHelper` class builds a routing database and initialize the routing tables of the nodes in the simulation. It also makes all nodes in the simulation into routers. All that actually this function does is call the functions `BuildGlobalRoutingDatabase()` and `InitializeRoutes()` internally.

Now it's time to set up our initial link costs. These are the metrics set by the network manager at the network setup that in our test case actually represents a bad setup. As explained in chapter 1, we are going to assign link costs in such a way to overload the links 02, 32 and 52 (node 5 - interface 2).

Our metric assignment for the OSPF network of Figure 3-1 is shown by green numbers in Figure 3-2. The `Ipv4` objects of all the interfaces in our simulation have a method called `SetMetric`. This method is used to set the initial costs in our OSPF network. After each cost assignment to any interface we execute the following code in order to force ns-3 routers to recompute their routing tables:

```
Ipv4GlobalRoutingHelper::RecomputeRoutingTables();
```



**Figure 3-2 Simulated OSPF network with bad cost (metric) setup**

`RecomputeRoutingTables()` removes all routes that were previously installed in a prior call to either `PopulateRoutingTables()` or `RecomputeRoutingTables()`, and adds a new set of routes. This method does not change the set of nodes over which `GlobalRouting` is being used, but it will dynamically update its representation of the global topology before recomputing routes.

The last part of CA-OSPF which is remaining is our hill climber itself which is actually the core of our cost adaptive OSPF network. We implement it and all its subroutines in a function called `hill()`.

### 3.3.14 Simulator

Finally it's now time to schedule and run our discrete event simulation. We schedule our calls to `hill()` by using the `Schedule` method of class `Simulator`. This method schedules an event to expire at the relative time "t" is reached. This can be thought of as scheduling an event for the current simulation time plus the `Time` passed as a parameter. When the event expires (when it becomes due to be run) the input method will be invoked on the input object. To schedule our hill climber to run on time frequencies indicated by the variable  $f$ , we can write the following snippet:

```
for (int t=start+f; t<=stop; t=t+f)
    simulator::Schedule (Seconds(t),hill);
```

This loop sets `hill()` up to be run after each  $f$  seconds intervals starting at time “*start*” and stopping at time “*stop*” seconds. Now all is remaining is to run our simulation:

```
simulator::Run ();
```

When `Simulator::Run` is called, the system will begin looking through the list of scheduled events and executing them. When these events are all executed, there are no further events to process and `Simulator::Run` returns. The simulation is then complete. All that remains is to clean up. This is done by calling the global function `Simulator::Destroy`. As the helper functions (or low level ns-3 code) executed, they



arranged it so that hooks were inserted in the simulator to destroy all of the objects that were created. We did not have to keep track of any of these objects ourselves — all we need to do is to call `Simulator::Destroy` and exit from our program:

```
simulator::Destroy ();  
return 0;
```

### 3.4 Test Case Results

Now it's time to see our simulated CA-OSPF network in action. In this test case scenario, we set the step size of our hill climber  $\Delta=2$  and the frequency of calls to `hill()` at  $f=2$  seconds i.e. `hill()` is scheduled to be invoked every two seconds during the simulation run. Furthermore, we set the start of our two 40Mbps traffics at *start*=1 second and the stop time at *stop*=30 seconds where simulation ends too. So our simulation is set to run for 30 seconds of simulation time. Please note this indicates the simulation time and is actually different from the real time. The actual time that is taken for the simulation to complete 30 seconds of our scenario is approximately 442 seconds on a 32-bit 1.6 GHz Centrino machine.

Recalling that our links are 100Mbps point-to-point links, we set our first hill climber threshold  $\Theta_1=20\%$  and our second threshold  $\Theta_2=80\%$ . So if the average link utilization of any interface in our network goes above 80Mbps during any monitoring interval, the cost adaptive component will increase the interface's cost by  $\Delta=2$ . If the average link utilization remains between 20Mbps and 80Mbps, then the interface's cost is

kept unchanged and finally if it happens that a interface's average link utilization falls below 20Mbps in any monitoring interval, its cost will be decreased by  $\Delta=2$  provided that it's not already equal to the interface's initial cost  $C_0$  (shown in Figure 3-2).

This setup of threshold values and our two traffic flows at 40Mbps, cause our hill() to flag links 02, 32 and 52 in figure 3-2 as overloaded from the very first monitoring interval. Therefore, the cost adaptive component of hill() starts increasing those interfaces' costs after each monitoring interval as long as their link utilizations are above 80Mbps. This ultimately causes traffic 1 to find a better path than going through the path R1-R2-R3-R8-R7-R6-R5-LAN5. Figure 3-3 (a) to (f) shows the output of our program at different simulation times.

```

ighamari@ubuntu: ~/tarballs/ns-allinone-3.6/ns-3.6
File Edit View Terminal Help
Now time is 3s.
Data rate on link 00 is 0 Mbps.
Data rate on link 01 is 0 Mbps.
Data rate on link 02 is 84.6842 Mbps. (Cost on link 02 changed to 3)
Data rate on link 10 is 42.3432 Mbps.
Data rate on link 11 is 0 Mbps.
Data rate on link 20 is 0 Mbps.
Data rate on link 21 is 0 Mbps.
Data rate on link 22 is 0 Mbps.
Data rate on link 30 is 0 Mbps.
Data rate on link 31 is 0 Mbps.
Data rate on link 32 is 84.6821 Mbps. (Cost on link 32 changed to 3)
Data rate on link 40 is 42.3367 Mbps.
Data rate on link 41 is 0 Mbps.
Data rate on link 42 is 0 Mbps.
Data rate on link 50 is 0 Mbps.
Data rate on link 51 is 0 Mbps.
Data rate on link 52 is 84.6799 Mbps. (Cost on link 52 changed to 3)
Data rate on link 60 is 42.3389 Mbps.
Data rate on link 61 is 0 Mbps.
Data rate on link 62 is 0 Mbps.
Data rate on link 70 is 0 Mbps.
Data rate on link 71 is 42.3389 Mbps.

```

**Figure 3-3 (a) Simulation time 17:00:03**

```

ighamari@ubuntu: ~/tarballs/ns-allinone-3.6/ns-3.6
File Edit View Terminal Help
Now time is 5s.
Data rate on link 00 is 0 Mbps.
Data rate on link 01 is 0 Mbps.
Data rate on link 02 is 84.6886 Mbps. (Cost on link 02 changed to 5)
Data rate on link 10 is 42.3432 Mbps.
Data rate on link 11 is 0 Mbps.
Data rate on link 20 is 0 Mbps.
Data rate on link 21 is 0 Mbps.
Data rate on link 22 is 0 Mbps.
Data rate on link 30 is 0 Mbps.
Data rate on link 31 is 0 Mbps.
Data rate on link 32 is 84.6886 Mbps. (Cost on link 32 changed to 5)
Data rate on link 40 is 42.3454 Mbps.
Data rate on link 41 is 0 Mbps.
Data rate on link 42 is 0 Mbps.
Data rate on link 50 is 0 Mbps.
Data rate on link 51 is 0 Mbps.
Data rate on link 52 is 84.6886 Mbps. (Cost on link 52 changed to 5)
Data rate on link 60 is 42.3432 Mbps.
Data rate on link 61 is 0 Mbps.
Data rate on link 62 is 0 Mbps.
Data rate on link 70 is 0 Mbps.
Data rate on link 71 is 42.3454 Mbps.

```

**(b) Simulation time 17:00:05**

```

ighamari@ubuntu: ~/tarballs/ns-allinone-3.6/ns-3.6
File Edit View Terminal Help
Now time is 7s.
Data rate on link 00 is 0 Mbps.
Data rate on link 01 is 0 Mbps.
Data rate on link 02 is 84.6864 Mbps. (Cost on link 02 changed to 7)
Data rate on link 10 is 42.3432 Mbps.
Data rate on link 11 is 0 Mbps.
Data rate on link 20 is 0 Mbps.
Data rate on link 21 is 0 Mbps.
Data rate on link 22 is 0 Mbps.
Data rate on link 30 is 0 Mbps.
Data rate on link 31 is 0 Mbps.
Data rate on link 32 is 84.6864 Mbps. (Cost on link 32 changed to 7)
Data rate on link 40 is 42.3432 Mbps.
Data rate on link 41 is 0 Mbps.
Data rate on link 42 is 0 Mbps.
Data rate on link 50 is 0 Mbps.
Data rate on link 51 is 0 Mbps.
Data rate on link 52 is 84.6864 Mbps. (Cost on link 52 changed to 7)
Data rate on link 60 is 42.3432 Mbps.
Data rate on link 61 is 0 Mbps.
Data rate on link 62 is 0 Mbps.
Data rate on link 70 is 0 Mbps.
Data rate on link 71 is 42.3432 Mbps.

```

**(c) Simulation time 17:00:07**

```

ighamari@ubuntu: ~/tarballs/ns-allinone-3.6/ns-3.6
File Edit View Terminal Help
Now time is 17s.
Data rate on link 00 is 0 Mbps.
Data rate on link 01 is 0 Mbps.
Data rate on link 02 is 84.6864 Mbps. (Cost on link 02 changed to 17)
Data rate on link 10 is 42.3432 Mbps.
Data rate on link 11 is 0 Mbps.
Data rate on link 20 is 0 Mbps.
Data rate on link 21 is 0 Mbps.
Data rate on link 22 is 0 Mbps.
Data rate on link 30 is 0 Mbps.
Data rate on link 31 is 0 Mbps.
Data rate on link 32 is 84.6886 Mbps. (Cost on link 32 changed to 17)
Data rate on link 40 is 42.3432 Mbps.
Data rate on link 41 is 0 Mbps.
Data rate on link 42 is 0 Mbps.
Data rate on link 50 is 0 Mbps.
Data rate on link 51 is 0 Mbps.
Data rate on link 52 is 84.6886 Mbps. (Cost on link 52 changed to 17)
Data rate on link 60 is 42.3454 Mbps.
Data rate on link 61 is 0 Mbps.
Data rate on link 62 is 0 Mbps.
Data rate on link 70 is 0 Mbps.
Data rate on link 71 is 42.3432 Mbps.

```

(d) Simulation time 17:00:17

```

ighamari@ubuntu: ~/tarballs/ns-allinone-3.6/ns-3.6
File Edit View Terminal Help
Now time is 19s.
Data rate on link 00 is 0 Mbps.
Data rate on link 01 is 42.3454 Mbps.
Data rate on link 02 is 42.3432 Mbps.
Data rate on link 10 is 42.3454 Mbps.
Data rate on link 11 is 0 Mbps.
Data rate on link 20 is 0 Mbps.
Data rate on link 21 is 0 Mbps.
Data rate on link 22 is 0 Mbps.
Data rate on link 30 is 0 Mbps.
Data rate on link 31 is 0 Mbps.
Data rate on link 32 is 42.3432 Mbps.
Data rate on link 40 is 0.004336 Mbps.
Data rate on link 41 is 0 Mbps.
Data rate on link 42 is 0 Mbps.
Data rate on link 50 is 0 Mbps.
Data rate on link 51 is 0 Mbps.
Data rate on link 52 is 42.3432 Mbps.
Data rate on link 60 is 0.002168 Mbps.
Data rate on link 61 is 0 Mbps.
Data rate on link 62 is 0 Mbps.
Data rate on link 70 is 0 Mbps.
Data rate on link 71 is 0.002168 Mbps.

```

(e) Simulation time 17:00:19

```

ighamari@ubuntu: ~/tarballs/ns-allinone-3.6/ns-3.6
File Edit View Terminal Help
Now time is 21s.
Data rate on link 00 is 0 Mbps.
Data rate on link 01 is 42.3432 Mbps.
Data rate on link 02 is 42.3454 Mbps.
Data rate on link 10 is 42.3432 Mbps.
Data rate on link 11 is 0 Mbps.
Data rate on link 20 is 0 Mbps.
Data rate on link 21 is 0 Mbps.
Data rate on link 22 is 0 Mbps.
Data rate on link 30 is 0 Mbps.
Data rate on link 31 is 0 Mbps.
Data rate on link 32 is 42.3432 Mbps.
Data rate on link 40 is 0 Mbps.
Data rate on link 41 is 0 Mbps.
Data rate on link 42 is 0 Mbps.
Data rate on link 50 is 0 Mbps.
Data rate on link 51 is 0 Mbps.
Data rate on link 52 is 42.3454 Mbps.
Data rate on link 60 is 0 Mbps.
Data rate on link 61 is 0 Mbps.
Data rate on link 62 is 0 Mbps.
Data rate on link 70 is 0 Mbps.
Data rate on link 71 is 0 Mbps.

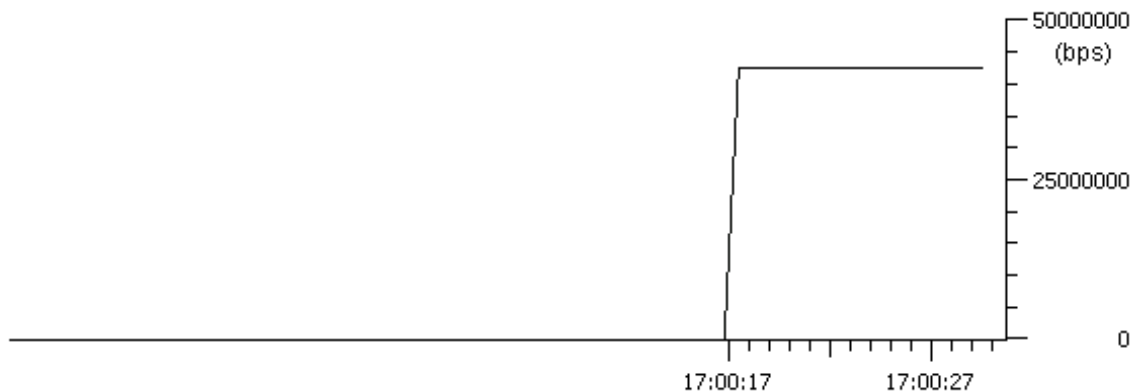
```

**(f) Simulation time 17:00:21**

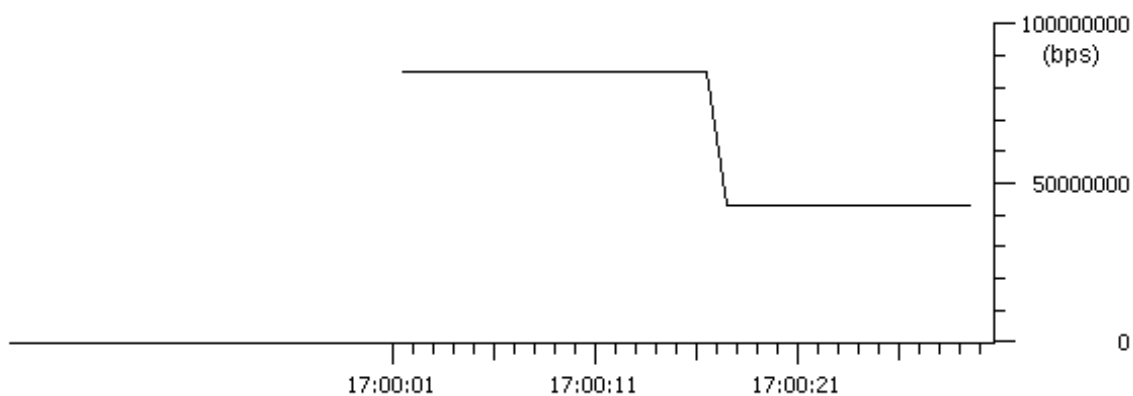
Let say the simulation started at time 17:00:00 (hh:mm:ss). We can see from figure 3-3 (a) that our hill() has noticed the overloaded links 02, 32 and 52 after the very first monitoring interval. The first monitoring interval actually started from the first second to 3s (17:00:03). The hill climber sees that the average link utilizations on these interfaces have been more than 80 Mbps during the first monitoring interval and hence increases the costs on those links by the step size value  $\Delta=2$ . Similarly after subsequent monitoring intervals, those links' costs are further increased (as seen in Figure 3-3 (b), (c) and (d)) because their average link utilizations have been still more than our upper threshold  $\Theta_2=80$  Mbps. Ultimately as seen in Figure 3-3 (d), when the costs on those

links are further increased to 17 at the simulation time 17:00:17, the total cost on path R1-R2-R3-R8-R7-R6-R5 for traffic-1 becomes 54 which is more than the cost of 50 which is for path R1-R5. Therefore, at this stage router R1 notices the better path for traffic-1 and hence starts forwarding all the packets belonging to traffic-1 to its interface 01 instead of interface 02. So, from next monitoring intervals shown in figure 3-3 (e) and (f), we can see that traffic-1 follows path R1-R5-LAN5 and traffic-2 follows path R4-R1-R2-R3-R8-LAN8 as before and therefore no interface in the network is overloaded anymore. So our CA-OSPF network took 17 simulation seconds to converge with  $f=2$  and  $\Delta=2$ . The result is a true stable traffic distribution in the network where no interface is overloaded. The costs on all the interfaces are kept unchanged as long as traffic conditions are same. If there are any changes in traffic patterns, our CA-OSPF network is able to adapt itself again and causes the traffics to be distributed evenly.

Figure 3-4 (a) and (b) show the link utilization graphs for links 01 and 02 respectively. The packets sent from these interfaces have been captured with Wireshark protocol analyzer in order to produce these graphs.



**Figure 3-4 (a) Link utilization on link 01 (node 0 – interface 1)**



**(b) Link utilization on link 02 (node 0 – interface 2)**

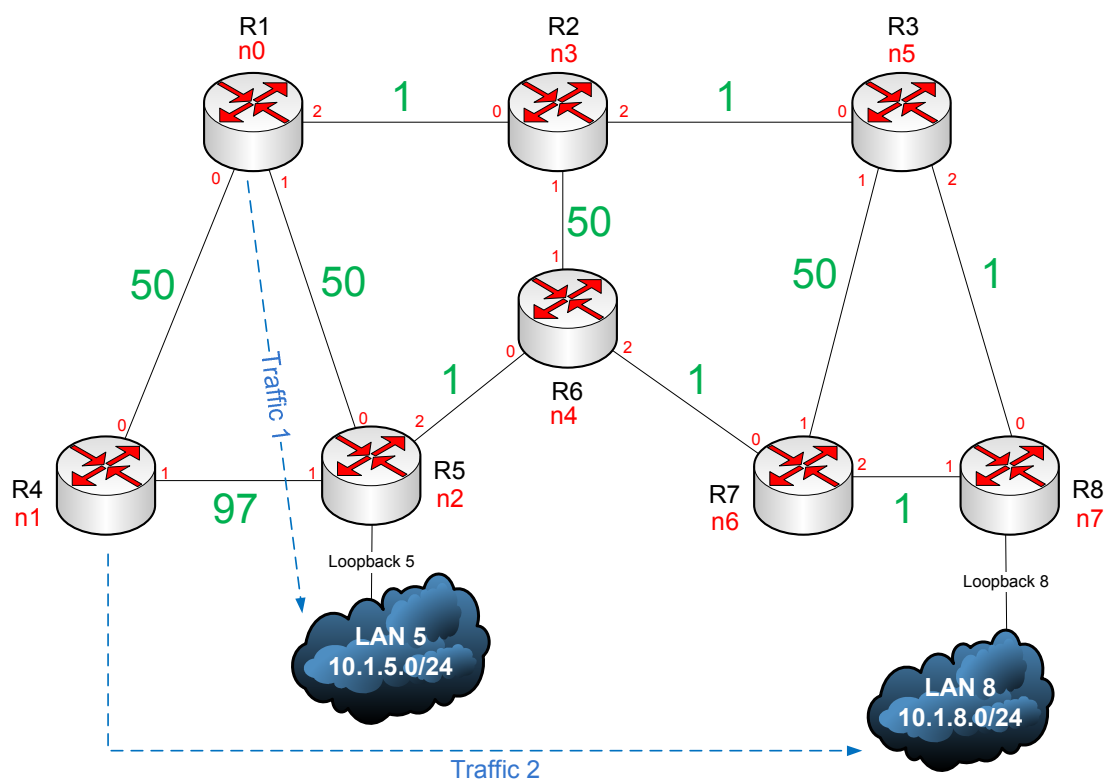
Figure 3-4 (a) shows that no packet was seen on link 01 until the simulation time of 17:00:17. We know that because R1 was forwarding all the packets to its interface 02 until that time. But after the simulation time of 17:00:17, R1 calculated a better path for traffic-1 and started forwarding traffic-1 packets to its interface 01 and hence we see the data rate on graph for link 01 jumps to ~42 Mbps after the simulation time of 17:00:17.

In Figure 3-4 (b) however, interface 2 of R1 (node 0) was overloaded from the beginning of simulation to simulation time 17:00:17 after which its data rate dropped to ~42 Mbps. This is because of the fact that traffic-1's load was removed from it at simulation time of 17:00:17.

These two graphs together show us how our CA-OSPF network relieved overloaded interface and consequently distributed total traffic in our network more evenly.

### 3.5 Routing Oscillations

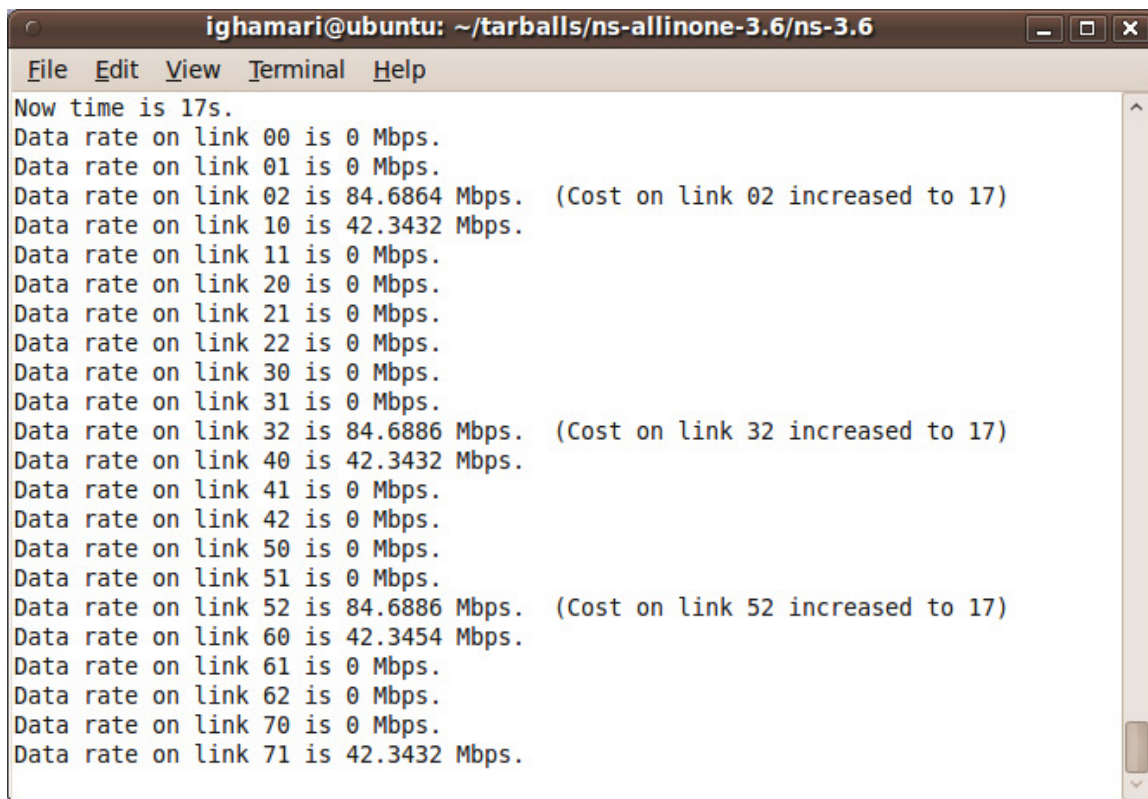
As it was briefly mentioned in introduction of this documentation, the best application of cost adaptive OSPF could be in Internet, but this kind of routing protocol has been largely abandoned in practice due to a problem associated with routing oscillations. Routing oscillation is a situation in which an interface is oscillated between overused and underused states. We can see the effect of this by changing the cost on our link R4-R5 from 98 to 97. The new cost assignment is shown in Figure 3-5.



**Figure 3-5 Metric assignment causing routing oscillation**



When `hill()` increases the cost on links 01, 32 and 52 after each monitoring interval, finally their individual costs reach 17 at simulation time 17:00:17. At this point not only traffic-1 finds a better path but also traffic-2 does. We have already seen the calculation for traffic-1's best path selection. For traffic-2, when the costs on links 02, 32 and 52 reach 17 each, the cost of path R4-R1-R2-R3-R8 becomes 101. At this point router R4 notices the path R4-R5-R6-R7-R8 which has a better total cost (100). Hence R4 starts forwarding traffic-2's packets to its interface 1 rather than interface 0. So, both the traffics are now rerouted and hence the link utilization of links 02, 32 and 52 drop to zero. This can be seen in Figure 3-6 when we are 19s into simulation.

A terminal window titled "ighamari@ubuntu: ~/tarballs/ns-allinone-3.6/ns-3.6" displays the output of a network simulation. The window has a menu bar with "File", "Edit", "View", "Terminal", and "Help". The output text shows the current time as 17s and lists data rates for various links. Three specific links (02, 32, and 52) are highlighted with a light blue background, and their costs are noted as having increased to 17. The data rates for these links are 84.6864 Mbps, 84.6886 Mbps, and 84.6886 Mbps respectively. Other links (00, 01, 10, 11, 20, 21, 22, 30, 31, 40, 41, 42, 50, 51, 60, 61, 62, 70, 71) show data rates of 0 Mbps or approximately 42.34 Mbps.

```
Now time is 17s.
Data rate on link 00 is 0 Mbps.
Data rate on link 01 is 0 Mbps.
Data rate on link 02 is 84.6864 Mbps. (Cost on link 02 increased to 17)
Data rate on link 10 is 42.3432 Mbps.
Data rate on link 11 is 0 Mbps.
Data rate on link 20 is 0 Mbps.
Data rate on link 21 is 0 Mbps.
Data rate on link 22 is 0 Mbps.
Data rate on link 30 is 0 Mbps.
Data rate on link 31 is 0 Mbps.
Data rate on link 32 is 84.6886 Mbps. (Cost on link 32 increased to 17)
Data rate on link 40 is 42.3432 Mbps.
Data rate on link 41 is 0 Mbps.
Data rate on link 42 is 0 Mbps.
Data rate on link 50 is 0 Mbps.
Data rate on link 51 is 0 Mbps.
Data rate on link 52 is 84.6886 Mbps. (Cost on link 52 increased to 17)
Data rate on link 60 is 42.3454 Mbps.
Data rate on link 61 is 0 Mbps.
Data rate on link 62 is 0 Mbps.
Data rate on link 70 is 0 Mbps.
Data rate on link 71 is 42.3432 Mbps.
```

```
ighamari@ubuntu: ~/tarballs/ns-allinone-3.6/ns-3.6
File Edit View Terminal Help
Now time is 19s.
Data rate on link 00 is 0 Mbps.
Data rate on link 01 is 42.3454 Mbps.
Data rate on link 02 is 0 Mbps. (Cost on link 02 decreased to 15)
Data rate on link 10 is 0 Mbps.
Data rate on link 11 is 42.3454 Mbps.
Data rate on link 20 is 0 Mbps.
Data rate on link 21 is 0 Mbps.
Data rate on link 22 is 42.3432 Mbps.
Data rate on link 30 is 0 Mbps.
Data rate on link 31 is 0 Mbps.
Data rate on link 32 is 0 Mbps. (Cost on link 32 decreased to 15)
Data rate on link 40 is 0.004336 Mbps.
Data rate on link 41 is 0 Mbps.
Data rate on link 42 is 42.3432 Mbps.
Data rate on link 50 is 0 Mbps.
Data rate on link 51 is 0 Mbps.
Data rate on link 52 is 0.002168 Mbps. (Cost on link 52 decreased to 15)
Data rate on link 60 is 0.002168 Mbps.
Data rate on link 61 is 0 Mbps.
Data rate on link 62 is 42.341 Mbps.
Data rate on link 70 is 0 Mbps.
Data rate on link 71 is 0.002168 Mbps.
```

```
ighamari@ubuntu: ~/tarballs/ns-allinone-3.6/ns-3.6
File Edit View Terminal Help
Now time is 21s.
Data rate on link 00 is 0 Mbps.
Data rate on link 01 is 0 Mbps.
Data rate on link 02 is 84.6864 Mbps. (Cost on link 02 increased to 17)
Data rate on link 10 is 42.3432 Mbps.
Data rate on link 11 is 0 Mbps.
Data rate on link 20 is 0 Mbps.
Data rate on link 21 is 0 Mbps.
Data rate on link 22 is 0.002168 Mbps.
Data rate on link 30 is 0 Mbps.
Data rate on link 31 is 0 Mbps.
Data rate on link 32 is 84.6842 Mbps. (Cost on link 32 increased to 17)
Data rate on link 40 is 42.3389 Mbps.
Data rate on link 41 is 0 Mbps.
Data rate on link 42 is 0.002168 Mbps.
Data rate on link 50 is 0 Mbps.
Data rate on link 51 is 0 Mbps.
Data rate on link 52 is 84.6821 Mbps. (Cost on link 52 increased to 17)
Data rate on link 60 is 42.3389 Mbps.
Data rate on link 61 is 0 Mbps.
Data rate on link 62 is 0.004336 Mbps.
Data rate on link 70 is 0 Mbps.
Data rate on link 71 is 42.341 Mbps.
```

```

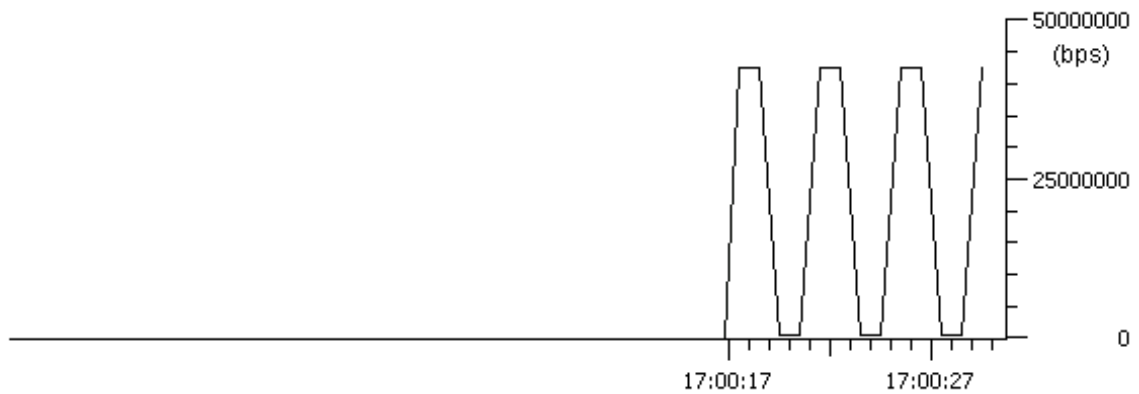
ighamari@ubuntu: ~/tarballs/ns-allinone-3.6/ns-3.6
File Edit View Terminal Help
Now time is 23s.
Data rate on link 00 is 0 Mbps.
Data rate on link 01 is 42.3432 Mbps.
Data rate on link 02 is 0 Mbps. (Cost on link 02 decreased to 15)
Data rate on link 10 is 0 Mbps.
Data rate on link 11 is 42.3432 Mbps.
Data rate on link 20 is 0 Mbps.
Data rate on link 21 is 0 Mbps.
Data rate on link 22 is 42.3432 Mbps.
Data rate on link 30 is 0 Mbps.
Data rate on link 31 is 0 Mbps.
Data rate on link 32 is 0.002168 Mbps. (Cost on link 32 decreased to 15)
Data rate on link 40 is 0.004336 Mbps.
Data rate on link 41 is 0 Mbps.
Data rate on link 42 is 42.341 Mbps.
Data rate on link 50 is 0 Mbps.
Data rate on link 51 is 0 Mbps.
Data rate on link 52 is 0.004336 Mbps. (Cost on link 52 decreased to 15)
Data rate on link 60 is 0.004336 Mbps.
Data rate on link 61 is 0 Mbps.
Data rate on link 62 is 42.341 Mbps.
Data rate on link 70 is 0 Mbps.
Data rate on link 71 is 0.002168 Mbps.

```

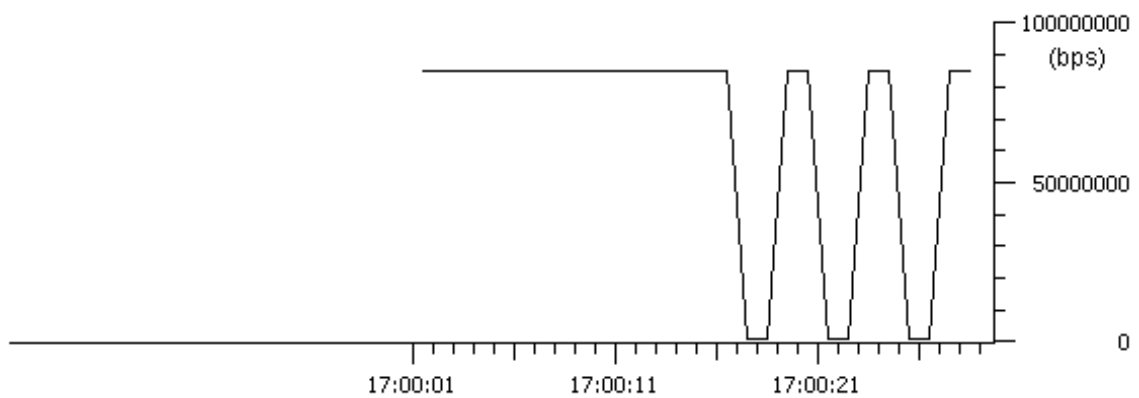
**Figure 3-6 Routing oscillations**

When the link utilization of links 02, 32 and 52 drop to zero, CA-OSPF flags them as underused and hence decrease their cost by  $\Delta=2$ . This causes both of our traffics to be rerouted again to their previous paths. After this point as also seen in Figure 3-6, links 02, 32 and 52 keep oscillating between overused and underused states. This increases the total overhead in network as more LSAs are generated for link costs that change every 2 seconds.

Figure 3-7 (a) and (b) show these oscillations on links 01 and 02 graphically.



**Figure 3-7 (a) Data rate oscillation on link 01**



**(b) Data rate oscillation on link 02**

As mentioned before, routing oscillations are the main issue in CA-OSPF that prevent developers from using it widely in internet systems. But this issue is not something without solution. I have developed an algorithm to change the step size of hill climber dynamically in real time to avoid the routing oscillations. This algorithm is currently under test at the time of this writing and its details are beyond the scope of this documentation.

## **Chapter Four: Lab Implementation**

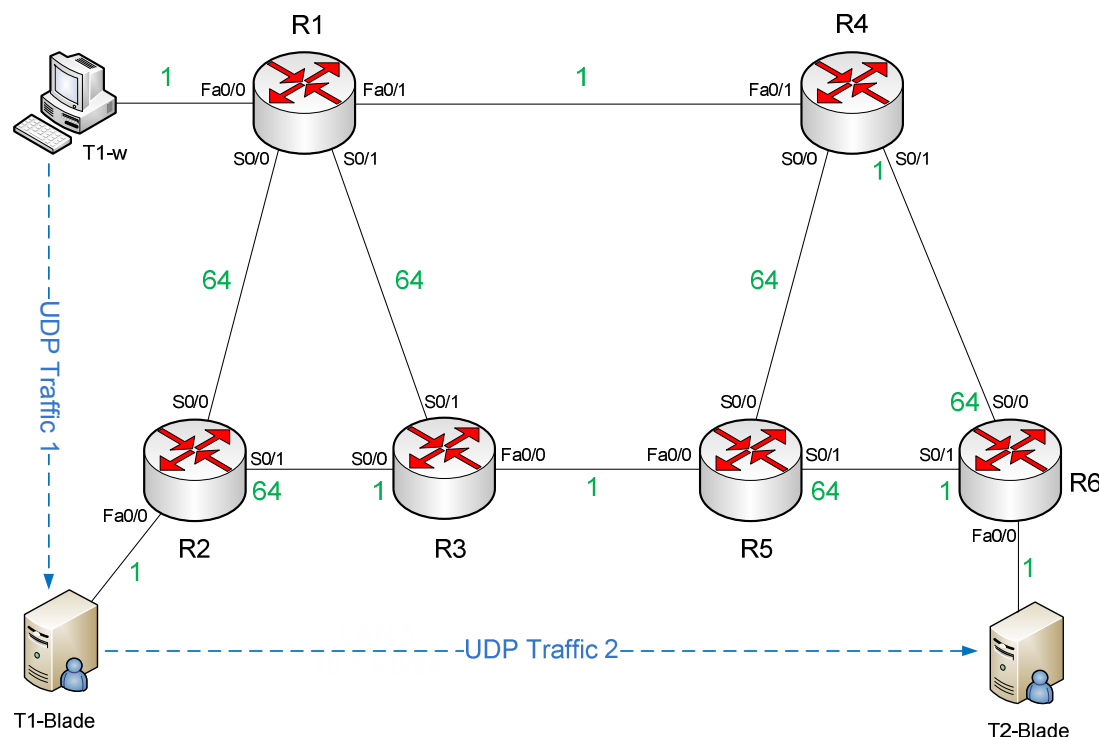
### **4.1 Open Source Routers**

After getting our expected results from the simulation, it's now time to implement our CA-OSPF in internetworking lab to see its action in real world and to measure and analyze its performance against projected optimization targets. To implement CA-OSPF directly inside routers, one is forced to use open source routers like Quagga. In other words, to modify the routing process inside the routers, there is no choice other than using these open source softwares. This is due to the fact that real routers like Cisco devices are closed source and hence do not allow programmers to modify their IOS. Therefore we can not directly modify router's OPSF process to convert it to CA-OSPF.

Since the initial plan of this project was to finally test our developed CA-OSPF with real routers, it was decided to use real Cisco boxes instead of open source routing suite softwares like Quagga. To use real Cisco routers however, we have to develop a method to implement our CA-OSPF inside routers indirectly. We will see the details of this method in section 4.3.

### **4.2 Designing OSPF Network with Cisco Routers**

The network topology that was designed in internetworking lab for our hill climber implementation is shown in Figure 4-1.



**Figure 4-1 OSPF network with Cisco 2600 series routers**

There are six Cisco 2600 series routers interconnected with the combination of Fast Ethernet and serial links as shown in the figure. Fast Ethernet interfaces operate at 100 Mbps and serial interfaces are configured to operate at the clock data rate of 8 Mbps. Fa0/0 of router R1 is connected to the workstation T1-w which serves as the source of our first UDP traffic. Workstation T1-w is also where we implement our CA-OSPF core algorithms. Fa0/0 of R2 is connected to a Sun Blade server (T1-Blade) which acts as the destination for traffic-1 as well as the source for our second UDP traffic. Fa0/0 of R6 is connected to our second Sun Blade T2-Blade which is the sink for traffic-2. Hence, traffic-1 flows from T1-w to T1-Blade and traffic-2 flows from T1-Blade to T2-Blade.

In Cisco routers by default the OSPF costs on Fast Ethernet and serial interfaces are 1 and 64 respectively, but in our network design shown in Figure 4-1, we have reduced the costs on some serial interfaces to 1 purposely in order to implement the bad cost scheme we have talked about in previous chapters. Please note that only the costs on serial interfaces R4 (S0/1), R6 (S0/1) and R3 (S0/0) are reduced to 1 and all the other serial interfaces in the network have the OSPF cost of 64. Now, with this bad cost assignment shown in the figure, traffic-1 takes the path (T1-W)-R1-R4-R6-R5-R3-R2-(T1-Blade) instead of physically shorter path of (T1-W)-R1-R2-(T1-Blade) and traffic-2 takes the path (T1-Blade)-R2-R1-R4-R6-(T2-Blade). This scheme therefore causes the links R1-R4 and R4-R6 to get overloaded by both the traffic flows.

All the network interfaces are addressed from 10.10.10.0/24 address block and OSPF routing process is configured for this network block on all the routers in the network.

### **4.3 Implementing CA-OSPF with Automated Scripts**

In section 4.1 we explained that we cannot directly implement our developed hill climber in Cisco routers. The IOS of Cisco routers are not open source and hence programmers are not allowed to modify its internal structure in any way. For our purpose what that means is that we cannot implement our CA-OSPF in IOS simply by modifying its OSPF process. Therefore what we need to do is to develop a method to implement our CA-OSPF somehow indirectly.

We can do this by writing some automated scripts. If we could write some scripts that run in a node attached to our network and if these scripts could remotely check the link utilization of all the links in our network via a network management protocol like SNMP then we have already implemented the monitoring subsystem of our CA-OSPF. Furthermore, if our scripts are also able to modify the OSPF costs on all the interfaces in the network then we have accomplished the task of cost adaptive subsystem as well.

This is the art of scripting where we have two distinct scripts; one is responsible for the monitoring subsystem of our hill climber and the other is responsible for the cost adaptive subsystem. I have developed the first script with Perl and the second script with Expect both of which are powerful scripting languages. Both scripts are run on workstation T1-w which is attached to our network via router R1. The Perl script responsible for the monitoring subsystem connects to each router in our network via SNMP, checks the link utilization of its interfaces and if required calls the Expect script to change their OSPF cost. The Expect script (if called) connects to the flagged routers remotely to change the OSPF cost on its flagged interface(s).

We will see the designing details of these two scripts in sections 4.5 and 4.6 but before that some background in SNMP and MIB-II are necessary. Therefore, we first go through these topics briefly in the next section.



#### 4.4 SNMP, OIDs and MIB-II

Simple Network Management Protocol is a simple method of interacting with networked devices. The standard is defined by IETF RFC 1157. SNMP can often seem quite confusing and overly complicated, its available APIs tend to put a lot of wrapping around what should be very simple.

A network device runs an SNMP agent as a daemon process which answers requests from the network. The agent provides a large number of Object Identifiers (OIDs). An OID is a unique key-value pair. The agent populates these values and makes them available. An SNMP manager (client) can then query the agent's key-value pairs for specific information. From a programming standpoint it's not much different than importing a ton of global variables. SNMP OIDs can be read or written.

OIDs are numerical and global. An OID looks similar to an IPv6 address and different vendors have different prefixes and so forth. The OIDs are long enough that it's complicated for a human to remember or make sense of them, so a method was devised for translating a numeric OID into a human readable form. This translation mapping is kept in a portable flat text file called a Management Information Base or MIB.

IETF RFC 1213 defines the second version of the Management Information Base (MIB-II) for use with network management protocols in TCP/IP-based internets. All SNMP agent and tool distributions should include MIBs that will comply with MIB-II and all devices should at the very least return values that comply with the MIB-II

standard. For our monitoring script, we use the 'Interfaces' group of MIB-II which is documented in RFC 2863.

SNMP can be used in 2 ways: polling and traps. Polling just means that we write an application that sets an SNMP GET request to an agent looking some value. This method is useful because if the device responds we get the information we want and if the device does not respond we know there is a problem. Polling is an active form of monitoring and we actually use it to implement the monitoring component of our hill climber in section 4.5. On the other hand, SNMP traps can be used for passive monitoring by configuring an agent to contact another SNMP agent when some action occurs.

#### **4.5 Scripting the Monitoring Subsystem with Perl**

The Simple Network Management Protocol (SNMP) offers a general way to remotely monitor and configure network devices and networked computers. One way we can use SNMP from Perl is to use a Perl SNMP module. There are at least three separate but similar modules available: `Net::SNMP`, by David M. Town; `SNMP_Session.pm`, by Simon Leinen; and a module that has had several names, including `NetSNMP`, `Perl/SNMP`, and “The Perl5 ‘SNMP’ Extension Module v5.0 for the Net-SNMP Library,” originally written by G. S. Marzot and now maintained by the Net-SNMP Project. The most significant difference between these three modules (other than their level of SNMP support) is their reliance on libraries external to the core Perl distribution. I chose `Net::SNMP` for our work because it's largely implemented in Perl alone. The `Net::SNMP` module implements an object oriented interface to the Simple Network Management

Protocol. Therefore, Perl applications can use the module to retrieve or update information on a remote host using the SNMP protocol. Each Net::SNMP object provides a one-to-one mapping between a Perl object and a remote SNMP agent or manager. Once an object is created, it can be used to perform the basic protocol exchange actions defined by SNMP.

Before starting to write our monitoring script, we first need to configure and run SNMP agent (process) on all the routers in our network. We use the following command to configure SNMP agent on all of our routers:

```
snmp-server community iman RW
```

This command sets the SNMP community name to 'iman' and enables the SNMP agent for both read and write access, but we will only use the read access.

Now we can start writing our Perl script to implement the monitoring subsystem of our CA-OSPF. Please note since a centralized approach is used to implement our monitoring subsystem, my actual Perl script includes many lines of code to solve the time synchronization issues between routers and the monitoring workstation. Here, we will only have a look at the high level design and will mostly pay attention to how Net::SNMP Perl module is used to connect to our routers to get the information we need. We will see the example codes for monitoring the interface Fa0/1 of router R1.

First we need to connect to our SNMP router:

```
my ($session,$error) = Net::SNMP->session(Hostname => $R1,  
                                           Community => 'iman');  
die "session error: $error" unless ($session);
```

`session()` is the constructor for `Net::SNMP` objects. In list context, a reference to a new `Net::SNMP` object (`$session`) and an empty error message string is returned. If a failure occurs, the object reference is returned as the undefined value. The error string (`$error`) may be used to determine the cause of the error. The IP address of the destination SNMP device can be specified using the `Hostname` argument. `$R1` is the variable that holds the IP address of router R1. Since the Security Model is Community-based, the only argument available is the `Community` argument. This argument expects a string that is to be used as the SNMP community name. Since the destination port number is not specified, the module uses the well-known SNMP port number 161.

Now that we are connected to our SNMP manager, the next step is to read the average data rate on its Fa0/1 interface over the past monitoring window. Inside the Interfaces Group of MIB-II (RFC 2863), there is a variable called `ifOutOctets`. Its description says:

`ifOutOctets` OBJECT-TYPE

SYNTAX Counter32

MAX-ACCESS read-only

STATUS current

DESCRIPTION

"The total number of octets transmitted out of the interface, including framing characters. Discontinuities in the value of this counter can occur at reinitialization of the management system, and at other times as indicated by the value of ifCounterDiscontinuityTime."

::= { ifEntry 16 }

So it's a Counter32 type variable with read-only access and according to its description, sounds promising for what we are looking for. The numerical OID for ifOutOctets is “.1.3.6.1.2.1.2.2.1.16”. Therefore to read ifOutOctets of interface Fa0/1 (IFindex 2), we can write:

```
my $result = $session->get_request ('.1.3.6.1.2.1.2.2.1.16.2');
die 'request error: '.$session->error unless (defined $result);
```

The `get_request()` method performs a SNMP get-request query to gather data from the remote agent on the host associated with the `Net::SNMP` object. Therefore `$result` gets the ifOutOctets of Fa0/1 of R1. Let say `$R1_fa01_OldMB` holds the total megabytes transmitted out of the interface, which was read last time the interface was monitored. Now after sleeping for time `t` (defined by frequency of calls to monitoring subsystem), we read the current ifOutOctets of the interface to calculate `$R1_fa01_NewMB` in order to

finally calculate the average data rate on the interface over the past monitoring interval

i.e. from the last time we read it up until now:

```
$R1_fa01_NewMB = $result->{'1.3.6.1.2.1.2.2.1.16.2'}/10**6;
  if ($R1_fa01_NewMB<$R1_fa01_OldMB) {
$MB=(4294967295-$R1_fa01_OldMB+$R1_fa01_NewMB)*8/tv_interval($time);
}
    else {
      $MB=($R1_fa01_NewMB-$R1_fa01_OldMB)*8/tv_interval($time); }
```

The IF statement in above snippet is to handle the situation where ifOutOctets had reached its maximum 32-bit value and wrapped around during the last monitoring interval. tv\_interval() is a method of Time::HiRes Perl module. We use this module in our script to implements a Perl interface to the usleep, ualarm, and gettimeofday system calls in order to achieve high resolution time synchronization for our monitoring subsystem. tv\_interval() returns the floating seconds between the two times. In our code the second argument is omitted, therefore the current time is used. \$time is a variable that holds the accurate time of last time the interface was monitored and it was set by the following statement:

```
$time=[gettimeofday()]; #Returns a floating seconds since the epoch
```

Finally, we reset \$time to current time again to be used in the next call to the monitoring subsystem and also we put \$R1\_fa01\_NewMB into \$R1\_fa01\_OldMB.

As seen so far, after each call to the monitoring subsystem we get the average data rate on the interface over the past monitoring interval. Now we need to pass this information to the cost adaptive subsystem. The cost adaptive subsystem determines whether the cost on the interface needs to be changed or not. Our Expect script (explained in next section) will be called if the interface cost has to be changed. For the Expect script to work in this way, our Perl script needs to pass the following information to it:

1. The console IP address of the destination router
2. Interface name of the interface whose cost needs to be changed
3. The new cost value

Last, before wrapping up our Perl script, we need to clear the Transport Domain and any errors associated with our Net::SNMP object. Once closed, the Net::SNMP object can no longer be used to send or receive SNMP messages.

```
$session->close();
```

#### **4.6 Scripting the Cost Adaptive Subsystem with Expect**

Expect is a UNIX automation and testing tool, written by Don Libes as an extension to the Tcl scripting language, for interactive applications. It uses UNIX pseudo terminals to wrap up subprocesses transparently, allowing the automation of arbitrary applications that are accessed over a terminal. Expect has regular expression pattern matching and general program capabilities, allowing simple scripts to intelligently control programs such as telnet, ftp and ssh, all of which lack a programming language,

macros, or any other program mechanism. The result is that Expect scripts provide old tools with significant new power and flexibility. In fact Expect serves as a "glue" to link existing utilities together.

All of these mean that we can actually use Expect to write a script to implement our cost adaptive subsystem. The idea is very simple: if the Expect script is called by our Perl script to change the cost on an interface, the Expect script logs in to the flagged router automatically, changes the cost on the flagged interface and exits. That's it!

We first write the following command in the beginning of our script to disable the program's output to the terminal in order to hide all the cost changing activities in the back and to keep telnet outputs out of our display.

```
log_user 0
```

The script starts by logging to the flagged router using the *spawn* command:

```
spawn telnet [!index $argv 0]
```

The *spawn* command starts another program. The first argument of the *spawn* command is the name of a program to start. The remaining arguments are passed to the program. So in our above code, *telnet* is started and the first argument that was passed to Expect is passed to it. Recall that this argument was actually passed by the Perl script and



is the console IP address of the flagged router. This directs telnet to open a connection to that host just as if the full command had been typed to the shell. We can now send commands using *send* and read prompts and responses using *expect*.

The first thing router will ask us after telnetting to it, is the IOS password. We can write the following codes in our script to interact with it:

```
expect "Password:"  
send "\nletmein\r"  
expect "OK"
```

The script waits until it matches the exact phrase "Password:" and after matching this, the script sends the password followed by a carriage return. Again script waits to receive "OK" indicating that the password we sent was correct.

Of course we also use regular expressions in expect commands to make our script more efficient. In the following lines we first send a carriage return after getting the above "OK" and then we wait to match a string starting with "Router" followed by any character including no character at all (given that the router's host name starts with the word "Router"). Then we send enable command to router to enter the privileged mode. Again we use regular expressions in the next expect command to match any character followed by the # sign, since we know that in router's privileged mode prompt ends with the number sign.

```
send "\r"
expect "Router*"
send "en\r"
expect "*#"
```

In the same way we can continue writing our script to interact with the router to finally change the cost on the requested interface. Recalling that the flagged interface name and its new cost were passed to the Expect script as its second and third arguments respectively, the remaining of the script looks something like this:

```
send "conf t\r"
expect "*#"
send "int [lindex $argv 1]\r"
expect "*#"
send "ip ospf cost [lindex $argv 2]\r"
expect "*#"
send "exit\r"
expect "*#"
send "exit\r"
```

When the Expect script is done with the interface cost change, the control will be back to the Perl script to continue with the next monitoring window.

## 4.7 Test Case Results

Now it's time to see our automated scripts in action and see how they convert our OSPF network to a CA-OSPF network. Let's first set our hill climber parameters in our Perl script. We set the step size of hill climber  $\delta=5$  and the frequency of calls to our hill climber at  $t=10$  seconds. This means each monitoring window is 10 seconds long and therefore Perl script sleeps for 10 seconds between calls to monitoring subsystem.

Each traffic flow is set at 3Mbps considering the presence of serial links in our network. Since we have a combination of Fast Ethernet and serial links in our test network (figure 4-1), we set our threshold values with respect to the slower links. Setting  $\Theta_1=25\%$  and  $\Theta_2=75\%$  of serial links' speed, if the average data rate on any interface in our network goes above 6Mbps during any monitoring interval, the cost adaptive component will increase the interface's cost by  $\delta=5$ . If the average data rate remains between 2Mbps and 6Mbps, then the interface's cost is kept unchanged and finally if it happens that a interface's average link utilization falls below 2Mbps in any monitoring interval, its cost will be decreased by  $\delta=5$  provided that it's not already equal to the interface's initial cost  $C_0$  (shown in figure 4-1).

This setup causes the R1-Fa0/1 and R4-S0/1 interfaces to be flagged as overloaded during those monitoring intervals that both traffics flow in the network. Figures 4-2 to 4-6 show the output of our program on T1-w terminal at different important moments.

```

Router R1:
*****
Avg. data-rate transmitted out of R1-Fa0/1: 3.09 Mbps
Avg. data-rate transmitted out of R1-S0/0: 0.00 Mbps
Avg. data-rate transmitted out of R1-S0/1: 0.00 Mbps

Router R2:
*****
Avg. data-rate transmitted out of R2-S0/0: 0.00 Mbps
Avg. data-rate transmitted out of R2-S0/1: 0.00 Mbps

Router R3:
*****
Avg. data-rate transmitted out of R3-Fa0/0: 0.00 Mbps
Avg. data-rate transmitted out of R3-S0/0: 3.06 Mbps
Avg. data-rate transmitted out of R3-S0/1: 0.00 Mbps

Router R4:
*****
Avg. data-rate transmitted out of R4-Fa0/1: 0.00 Mbps
Avg. data-rate transmitted out of R4-S0/0: 0.00 Mbps
Avg. data-rate transmitted out of R4-S0/1: 3.07 Mbps

Router R5:
*****
Avg. data-rate transmitted out of R5-Fa0/0: 3.08 Mbps
Avg. data-rate transmitted out of R5-S0/0: 0.00 Mbps
Avg. data-rate transmitted out of R5-S0/1: 0.00 Mbps

Router R6:
*****
Avg. data-rate transmitted out of R6-S0/0: 0.00 Mbps
Avg. data-rate transmitted out of R6-S0/1: 3.07 Mbps

#####
##### Sleeping for 10.00 s #####
#####
- codename [ pwnsauce ]

```

**Figure 4-2** This figure indicates that during the past monitoring window, only traffic-1 was flowing in network. Due to the bad cost scheme used in the design of OSPF network, traffic-1 is following the path R1-R4-R6-R5-R3-R2 instead of physically shorter path of R1-R2.



```

Router R1:
*****
Avg. data-rate transmitted out of R1-Fa0/1: 6.17 Mbps -----> The cost on R1-Fa0/1 increased to 31
Avg. data-rate transmitted out of R1-S0/0: 0.00 Mbps
Avg. data-rate transmitted out of R1-S0/1: 0.00 Mbps

Router R2:
*****
Avg. data-rate transmitted out of R2-S0/0: 3.06 Mbps
Avg. data-rate transmitted out of R2-S0/1: 0.00 Mbps

Router R3:
*****
Avg. data-rate transmitted out of R3-Fa0/0: 0.00 Mbps
Avg. data-rate transmitted out of R3-S0/0: 3.07 Mbps
Avg. data-rate transmitted out of R3-S0/1: 0.00 Mbps

Router R4:
*****
Avg. data-rate transmitted out of R4-Fa0/1: 0.00 Mbps
Avg. data-rate transmitted out of R4-S0/0: 0.00 Mbps
Avg. data-rate transmitted out of R4-S0/1: 6.10 Mbps -----> The cost on R4-S0/1 increased to 31

Router R5:
*****
Avg. data-rate transmitted out of R5-Fa0/0: 3.09 Mbps
Avg. data-rate transmitted out of R5-S0/0: 0.00 Mbps
Avg. data-rate transmitted out of R5-S0/1: 0.00 Mbps

Router R6:
*****
Avg. data-rate transmitted out of R6-S0/0: 0.00 Mbps
Avg. data-rate transmitted out of R6-S0/1: 3.07 Mbps

#####
##### Sleeping for 10.00 s #####
#####
#####

```

- codename [ pwnsaucе ]  
(From here network converged)

**Figure 4-4** The process of cost increment on R1-Fa0/1 and R4-S0/1 interfaces continued until the OSPF cost on these two interfaces reached 31 each. At this point, according to the initial cost plan seen in figure 4-1, the total cost for traffic-1 to flow through its current path from R1 has reached 66 which is higher compared to the physically shorter path of R1-R2-(T1-Blade) which has the cost of 65. Therefore at this point, router R1 updates its routing table and from here onwards it will forward all the packets destined for T1-Blade to its S0/0 interface instead of Fa0/1.



```

Router R1:
*****
Avg. data-rate transmitted out of R1-Fa0/1: 1.77 Mbps ----> The cost on R1-Fa0/1 decreased to 26
Avg. data-rate transmitted out of R1-S0/0: 0.00 Mbps
Avg. data-rate transmitted out of R1-S0/1: 0.00 Mbps

Router R2:
*****
Avg. data-rate transmitted out of R2-S0/0: 1.44 Mbps
Avg. data-rate transmitted out of R2-S0/1: 0.00 Mbps

Router R3:
*****
Avg. data-rate transmitted out of R3-Fa0/0: 0.00 Mbps
Avg. data-rate transmitted out of R3-S0/0: 0.00 Mbps
Avg. data-rate transmitted out of R3-S0/1: 0.00 Mbps

Router R4:
*****
Avg. data-rate transmitted out of R4-Fa0/1: 0.00 Mbps
Avg. data-rate transmitted out of R4-S0/0: 0.00 Mbps
Avg. data-rate transmitted out of R4-S0/1: 1.43 Mbps ----> The cost on R4-S0/1 decreased to 26

Router R5:
*****
Avg. data-rate transmitted out of R5-Fa0/0: 0.00 Mbps
Avg. data-rate transmitted out of R5-S0/0: 0.00 Mbps
Avg. data-rate transmitted out of R5-S0/1: 0.00 Mbps

Router R6:
*****
Avg. data-rate transmitted out of R6-S0/0: 0.00 Mbps
Avg. data-rate transmitted out of R6-S0/1: 0.00 Mbps

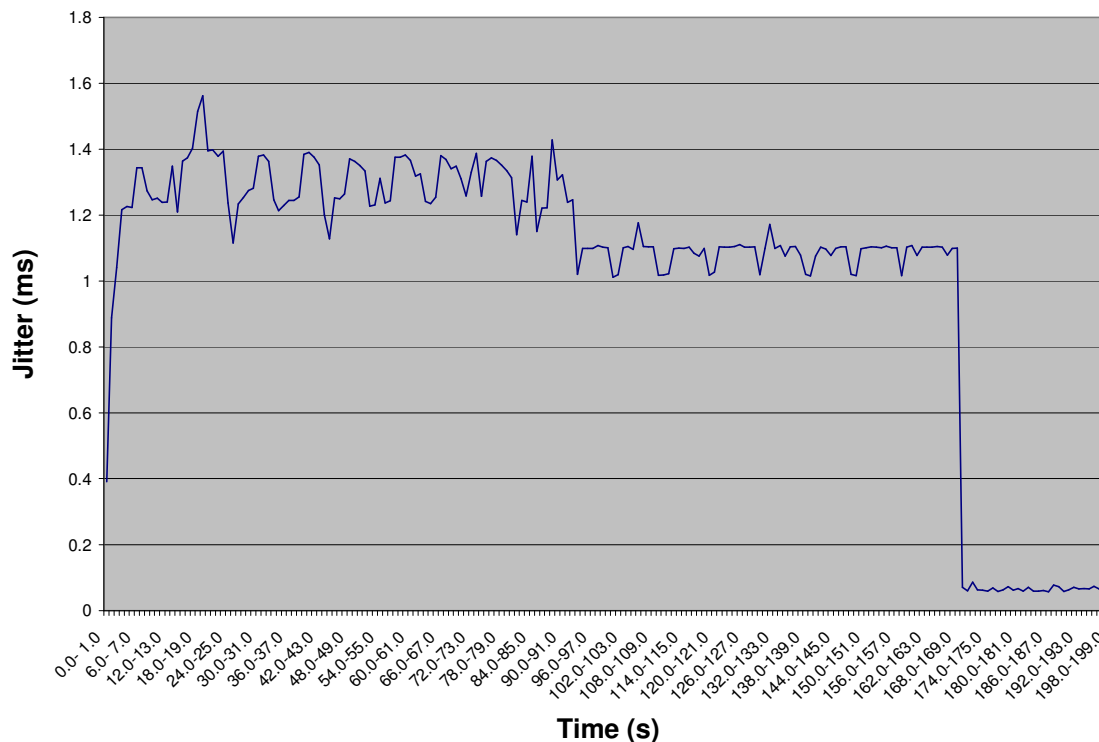
#####
##### Sleeping for 10.00 s #####
#####
- codename [ pwnsauc3 ]

```

**Figure 4-6** Finally in this figure that shows the output of the past monitoring window, we can see that traffic-1 had been stopped completely and traffic-2 was going down. Now since the average data rate on R1-Fa0/1 and R4-S0/1 interfaces are below 2 Mbps, our Perl script flags them as under-used and since their OSPF cost are higher than their initial values (1), the Expect script is called to decrease their cost by  $\$delta=5$ . Therefore their costs drop to 26. Again, this process continues as long as the costs on under-used interfaces are higher than their initial values.



Now let's see the performance boost that we get from CA-OSPF in comparison with OSPF in terms of packet jitter. We manage our traffics to start at almost the same time and to stop after 200 seconds.



**Figure 4-7 Average packet jitter over time for traffic2 packets received on T2-Blade**

Figure 4-7 shows the average jitter over time for packets (belonging to traffic-2) received on T2-Blade machine. We can see that during first few seconds when traffic-1 starts too, the average jitter of traffic-2 packets gets shot to above 1.2 ms. The packets jitter remain relatively high until time=169 seconds. At this time network converges and traffic on overloaded links are distributed more evenly (this corresponds to figure 4-4).

Therefore traffic congestion is greatly reduced on the links along the traffic-2 path. The result can be seen in the figure 4-7 where the packet jitter drops drastically at time=169 seconds and remains low from there on.

If we imagine that our network was an OSPF network without traffic management capabilities, we can see how it could suffer from high packet jitters as those seen in figure 4-7 before time=169 seconds. But CA-OSPF which is an OSPF routing protocol with added traffic management capabilities enhanced our network performance.

## Chapter Five: Conclusion

In this project one of the shortcomings of the existing OSPF routing protocol was identified in terms of calculating the *real* best paths, and on the basis of this identification, a cost adaptive version of the routing protocol was developed to include the extra traffic management capabilities to overcome the identified shortcoming.

The cost adaptive OSPF was first researched and then developed using a variation of hill climbing algorithm. The developed algorithm was then coded (in C++) and simulated in ns-3 network simulator to analyze its performance in a simulation world. After seeing the expected results from the simulation, the final milestone was to implement the cost adaptive OSPF in real world and to measure and analyze its performance against projected optimization targets.

The cost adaptive OSPF proved to overcome the shortcoming of traditional OSPF in terms of even traffic distribution and in author's view it can definitely replace OSPF in internet systems provided that the associated routing oscillation issue is solved. As it was mentioned in section 3.5, the routing oscillation problem associated with the cost adaptive OSPF could have some solutions; one of which is currently under test by the author as a potential patch to the developed cost adaptive OSPF.

## References

- [1] ns-3 Tutorial, 28 January 2010
- [2] ns-3 Reference Manual, ns-3-dev 29 April 2010
- [3] Zhou Haijun , Pan Jin & Shen Pubing , “Cost adaptive OSPF”, Lab of Network Eng., Xi'an Commun. Inst., China
- [4] Tatiana B. Pereira and Lee L. Ling, “Network Performance Analysis of an Adaptive OSPF Routing Strategy – Effective Bandwidth Estimation”, International Telecommunication Symposium – ITS 2002, Natal, Brazil
- [5] Tatiana Brito Pereira, Lee Luan Ling, “An OPNET Modeler Based Simulation Platform for Adaptive Routing Evaluation”, *FEEC, UNICAMP Campinas, S.P., Brazil*
- [6] J. Moy, “OSPF Version 2”, RFC 2328, April 1998.
- [7] K. McCloghrie, F. Kastenholz, “The Interfaces Group MIB”, RFC 2863, June 2000.
- [8] Automating System Administration with Perl, David N. Blank-Edelman, May 2009
- [9] The Net-SNMP Programming Guide, Ben Rockwood. Updated: Nov 17th, 2004
- [10] Net::SNMP by David M. Town (<http://search.cpan.org/~dtown/Net-SNMP-v6.0.1/lib/Net/SNMP.pm>)
- [11] Exploring Expect - A Tcl-based Toolkit for Automating Interactive Programs, Don Libes, December 1994
- [12] S. Savage & A. Collins & E. Hoffman & J. Snell & T. Anderson., The End-to-End Effects of Internet Path Selection, 1999 ACM. SIGCOMM Conference, pp. 289-299, September 1999.

- [13] D. W. Glazer & C. Tropper, A New Metric for Dynamic Routing. Algorithms, IEEE Transactions on Communications, Vol.38 No.3, March 1990.
- [14] A. Khanna & J. Zinky, The Revised ARPANET Routing Metric, In. Proceedings of the ACM SIGCOMM, pp. 45-56, 1989.
- [15] D. Edelson. Smart pointers: They're smart, but they're not pointers. University of California, Santa Cruz, Computer Research Laboratory, 1992.
- [16] Bernard Fortz, Mikkel Thorup, "Internet Traffic Engineering by Optimizing OSPF Weights", IEEE INFOCOM 2000, pp. 519-528.
- [17] D. Frigioni, M. Ioffreda, U. Nanni, and G. Pasqualone, "Experimental analysis of dynamic algorithms for the single-source shortest path problems", ACM Journal of Experimental Algorithmics, vol 3, article 5, 1998.