University of Alberta
Master of Science Internetworking

CAPSTONE PROJECT

"TELUS MPLS QoS enabled Converged Network Simulation"

Student: Elshan Hasanov

Supervisor: Mr. Pete Nanda

Submission date: April 14, 2014

CONTENTS

Abstract……..2
Solution……..3
Acknowledgement…….4
Background and Challenges……5
Problem Statement…….8

Solution and Simulation Design

Network Design……10
Network Components……...12
Solution for MPLS……...16
Solution for DifferServ QoS……17
Simulation Parameters……..22

Simulation and Lab Results Analysis

Queuing Time Performance……..24
End-to-End Delay Result……..27
Drop Precedence…….....34
Statistics at MPLS node…….....35
Test Results done in Laboratory……..…….35
Link Failure……...……...37
Conclusion…….....….....38
Future Work……..39
References…….……40

APPENDIX

Simulation Background…….41
Configuration of description (ned) file…….….43
Configuration of omnet.ini file……..49

ABSTRACT

Internet delivered individually each packet, and delivery time was not guaranteed, moreover every packet might be dropped due to congestion at the routers in the early days of internetworks by default. This solution was suitable until file transfer and email protocols were the main application of internet. Recently, we have witnessed a vast deployment of delay sensitive and real-time applications on internet. New generation technologies have different requirements, specific needs for their services, as a result QoS becomes more important on WAN. Providing QoS and MPLS capabilities in the internet is very essential to support the requirements of current and future generation Internet services.

Many researches and projects have been done and many are in progress to measure, compare delay performance of different traffic classes as it becomes necessary to provide different services to different classes and customers in different network implementations. The goal of this project is to evaluate the MPLS QoS performance of real-time applications such as voice and video conferencing operating over MPLS/DiffServ and compare the results against best effort traffic

SOLUTION

We will study and measure behaviour of Expedited Forwarding, Assured Forwarding , Best Effort, traffic aggregation, link congestion and the link failure/recovery affect to end-to-end performance and delay variation, throughput and packet loss. Qualitative and quantitative analysis of QoS over MPLS backbone will be implemented using OMNET simulation tool. Omnet++ is an open source network simulator which allows us to simulate any network scenario, change behaviour of models and the output of simulations overlaps with test results done in lab environment using devices of different vendors (Cisco, Alcatel). The main simulation objective is to calculate results, compare them against each other and real network results and answer the following questions.

- How much time difference is observed in the queuing time and queuing data among voice, video and traditional traffic?

- To what extend do the performances of end-to-end delay, delay variation for EF, AF11 and BE per hob behaviours vary from each other over MPLS/DifferServ network?

- How link failure and recovery will affect to the delay and throughput and solution offers to overcome the failure effect?

ACKNOWLEDGEMENT

I want to thank my supervisor Mr. Pete Nanda for his encouraging guidance and help during the project. Regular meetings with him helped me better understand importance of the project and make real network topology for the simulation. The simulation tool (OMNET++) which he recommended me to use is an important tool for network engineers to measure and to test network performance.

Secondly, I want to use this this opportunity to thank Professor Mike MacGregor, the director of MINT (Master of Science Internetworking) department, for making this course available for us. MINT is an industry oriented networking course where theory and practical knowledge come together and move hand-in-hand. Thank to him we are in touch with supervisors, experienced specialists from industry, we have chance to understand study and research needs of environment. Obtaining master degree from MINT at University of Alberta will play huge role in our future successes.

MINT laboratory is certainly one of the best one's across the Canada, where we have sufficient network devices from different vendors and servers for our project needs. Mir Shahnawaz - Program Coordinator and lab instructor is a high qualified network professional who is always available for guidance, kind help for projects or studies.

BACKGROUND AND CHALLANGES

Internet Protocol (IP) has enabled global network between an immense variety of systems and transmission media. One of the tremendous successes of IP in internet is its simplicity. The fundamental architecture of IP networks is derived from end-to-end communication - source and destination devices sending traffic to each other throw next hops. Routers at intersections throughout the network check the destination address against forwarding table to find the next hop for an IP packet and if the queue for the next hop is long, the packet may be delayed or dropped. Depending of the datagram some packets can be discarded, assembled if the queue is full or unavailable. Some IP packets carry discard eligibility information to notify next hops in case of congestion and nodes can support explicit congestion notification which provides a method for an intermediate router to notify end hosts of impending network congestion. The benefit of ECN is to reduce retransmission and packet loss in in data transmission. The overall result is that IP provides a best effort service which is the subject to unpredictable delays and data loss.

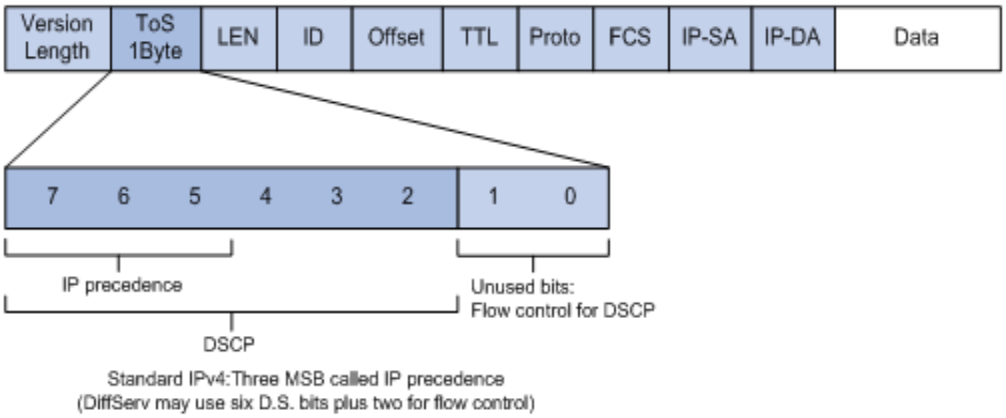
This structure of the IP was not a problem until file transfer and email was the main application of networks. Traditional internet applications like file transfer, web, email do not suffer from the best effort nature of IP. The new breed of real-time applications such as audio and video streaming applications demand high data throughput, bandwidth and low –latency requirements. WAN, MAN, LAN IP networks all are also being used increasingly for delivery of mission critical, delay-sensitive information that cannot tolerate unpredictable delay, queue and losses. It becomes increasingly important to manage and utilize network efficiently to accomplish the requirements of various Internet services or customers. In this case, QoS comes to help by providing consistent, predictable data delivery service. QoS is the ability of a network element to have some level of assurance that is traffic flow and service requirements can be satisfied according to service layer agreements. The other goal of the QoS is to provide priority to different class of traffic flow.

Besides the default behaviour of the QoS (best effort), two other QoS architectures used in IP networks when designing a QoS solution: integrated services and differentiated services. Integrated services - also known as hard QoS, is an absolute reservation of services. In other words, the IntServ model implies that traffic flows are reserved explicitly by all intermediate systems and resources. In this model the application requests a specific kind of service from the network before it sends data. The Resource Reservation Protocol (RSVP) can be used by applications to signal their QoS requirements to the router. However, the resource reservation requires a setup phase, where all involved network nodes receive a reservation request, check the availability and if possible reserve the resources (queue space, queuing management strategy, scheduling strategy) accordingly. Such a mechanism is a stateful procedure for the nodes and requires a connection-oriented session management. That is, there is a setup phase (which involves the reservation), a working phase (the actual communication session) and a tear down phase which also includes the release of reserved resources, bandwidth, waiting queue space, dropping mechanism. In order to avoid wasted reservations due to broken sessions, which have not gone

through a tear down, a soft-state approach with reservation timeouts and periodic reservation refreshes are used. Reservation also requires session admission control. That is, setup requests, which cannot provide the requested resources will block the request and to turn down the admission to the reserved network. Integrated services is designed for per flow traffic such as the bunch of the traffic using the same destination and source address are treated on the same way. Other drawback of the Integrated Services is that the reservation must be periodically refreshed, so there is an overhead during the data transmission too.

Differentiated services - also known as soft QoS, is class-based, which some classes of traffic receive preferential handling over other traffic classes. Differentiated services use statistical preferences, not a hard guarantee like integrated services. Instead of making per-flow reservations Differentiated Services divides the traffic into a small number of forwarding classes. In other words, DiffServ categorizes traffic and then sorts it into queues of various efficiencies, unlike in the integrated service model. An application using differentiated service does not explicitly signal the router before sending data. This model is flexible for aggregate flows because it performs a relatively coarse level of traffic classification.

The differentiated services architecture provides the most extended and attractive solution service quality support in IP networks. Scalability and traffic classification are main concerns for DiffServ as it can handle large number of data networks very efficiently. The network tries to deliver a particular kind of service based on the QoS descriptor specified in each IP packet header on a per-hop basis, rather than per-traffic flow. We can use the DiffServ model for the same mission-critical applications as IntServ and to provide end-to-end QoS. Network devices forward each according to priorities designated within the packet on a per-device, per-hop basis. Typically, this DiffServ model is the preferable model within the multilayer switched network because of its scalability. Differentiated Services redefines this IP header field, TOS field. The new usage of those header bits is called "Differentiated Service Code Point - DSCP" and defines six bits for priority selection and two unused bits of the header byte.

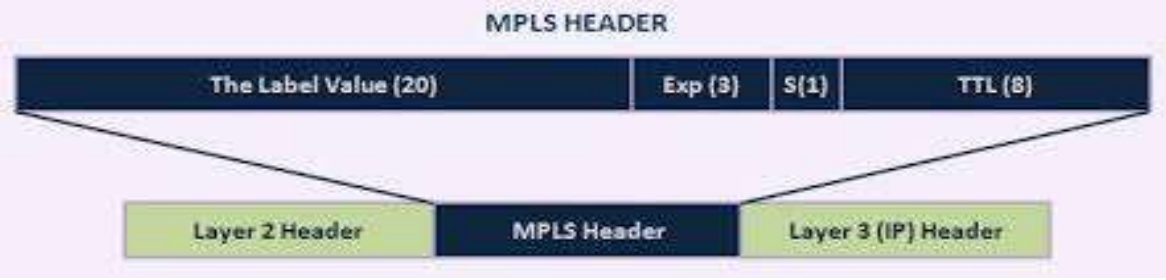


Six bits allow for 64 classes to be distinguished. There are only a few recommendations on how to select the number of supported classes, their encoding and resulting forwarding behaviour (Per Hop Behaviour - PHB) in the DiffServ enabled router devices. However, the following three DSCP groups are defined and commonly used: "Default PHB", "Class Selector PHB", "Expedited Forwarding PHB" and "Assured Forwarding PHB". "Default PHB" is used for general internet traffic as it is right now. "Class Selector PHB" is there to support the original precedence encoding, "Expedited Forwarding PHB" (RFC 2598) is the single encoding for highly prioritized traffic with strictly limited resources' (10% of traffic), which is the basis for services like pseudo wire emulation. "Assured Forwarding PHB" (RFC 2597) is the classification that marks relative priority within four classes and three drop precedence's within each AF class.

Although DiffServ model solves the scalability problem of IntServ but still there has some drawback like resource stealing. Flows share a common class compete inside the class for resources available to all members and in some occasions might reduce the performance of their competitors in term of QoS measures by stealing the resources that were initially used by their rivals. Hence, IETF has suggested using the MPLS architecture to solve the drawbacks of DiffServ on IP networks.

MPLS is a switching technology that brings up a new dimension for the traditional IP networks. The basic idea of MPLS network is to provide facilities of individual packets, which belong to individual Forwarding Equivalent Class (FEC), needed to be labeled before they are forwarded. In MPLS network no further packet header analysis is required once packet is labeled and assigned to a FEC before they are forwarded to the next hope and this is done only in ingress and egress nodes of MPLS domain. Label swapping is done only by mapping incoming label to determine outgoing interface.

The packet is assigned by the FEC encoded in a MPLS shim header as a short fixed – 32 bits length value which is called Label. Label is added between transport and IP headers.



DiffServ introduces scalable edge-to-edge QoS over MPLS as MPLS performs fast switching, traffic reservation, traffic engineering to distribute traffic load on available links and fast rerouting. Implementing IS-IS routing protocol with fast-rerouting feature will improve convergence time on failure which will result better performance on QoS. Moreover, MPLS can be

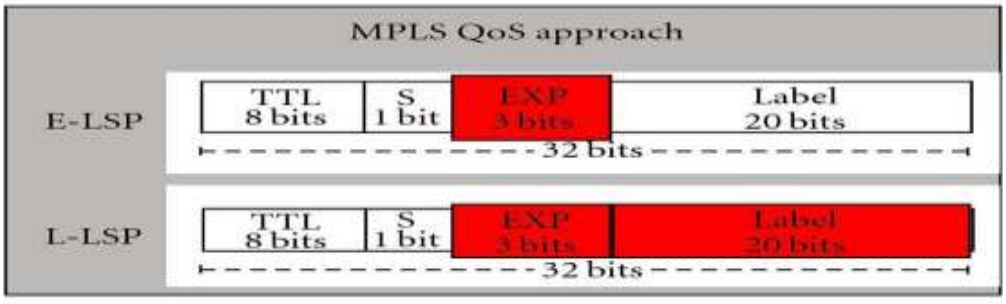
deployed over wide variety link layer technologies. The combination of DiffServ and MPLS presents better strategy to core network service providers with scalable and flexible QoS.

PROBLEMS ASSOCIATED WITH MPLS/DIFFERSERV

We have two issues in providing MPLS support for DiffServ. The DSCP (Differentiated Service Code Point) is carried in the IP header, but MPLS nodes only examine the label header while making forwarding decision. IP header has one byte for Type of service which two bits are used for flow control other six for DSCP mapping. DS code point uses six bits in IP header to map QoS classes while the EXP field has only three bits for label to per hob behaviour mapping.

One solution for this issue is applicable when the network can support less than eight per hob behaviour in MPLS core nodes. Three experimental (EXP) bits of shim header are used for mapping the DSCPs. Label indicates where the packet to be forwarded, while the EXP bits indicate how to treat the PHB. The EXP bits are value that needs to be configured based on the DSCP bits of the IP packets or by the network operator. LSPs are often called E-LSPs. The LSR determines the behaviour type to be applied to the incoming packet by looking up the EXP-to-PHB mapping.

First solution is not satisfying if network requires more than eight PHBs. EXP bits aloe are not able to hold all the necessary information to differentiate more than eight PHBs so other field of shim header label itself is used to provide more services. The label indicates where packet is to be forwarded and what scheduling behaviour should be made to grant it while EXP bits carries information about the drop priority allocated to a packet. As a result hop behaviours is specified from the label and EXP bits. Label is implicitly tied to a PHB and this information should be known as soon as the LSP is signaled. LSPs which use the label to convey information about the PHB are called L-LSPs. An arbitrary large number of PHBs can be supported in L-LSP method. Below described picture visualises MPLS QoS approach.



As a result packet marked with high service quality defined from layer to changes its code point while transiting from Layer 2 to Layer 3 and to MPLS domain. In each layer the QoS marker

value is translated to a new value and treated the same as original. Below added table describes DS, DSCP, MPLS EXP translations values of QoS pointers.

DS class	DSCP value	MPLS EXP value
EF (voice traffic high priority)	46	4
AF11 (video, low priority)	10	1
AF12 (video, medium priority)	12	2
AF13 (video, high priority)	14	3
BE (best effort)	0	0

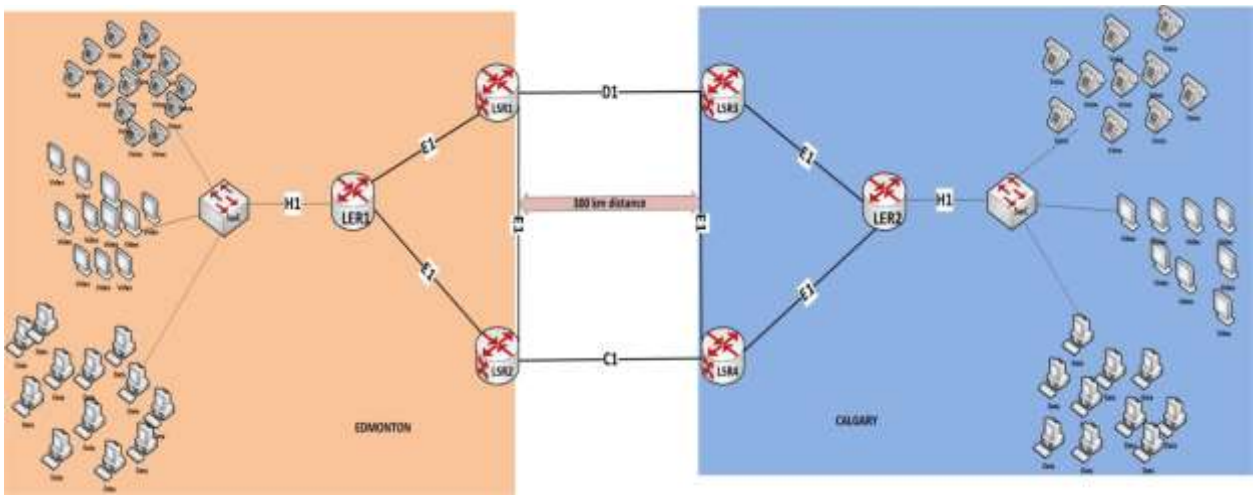
SOLUTION AND SIMULATION DESGIN

NETWORK DESIGN

OMNET++ (version 4.3) has been used in this project for MPLS/QoS simulation. OMNET++ is free object-oriented network simulation software used to simulate in modular from discrete events establishing hierarchical structures between each component.

For MPLS QoS demonstration we will simulate backbone network of service provider, providing MPLS service to the customers located in Edmonton and Calgary cities. Users located in these two cities send voice, video and data traffic to each other. In order to make analyse easy and clear many of the hosts intercommunicate only with one host during simulation by transmitting only voice, data or video traffic. Some hosts (Multi1&2) receive all three type of traffic simultaneously, sent from different hosts. We will measure end-to-end delay, queue time and length, packet loss value variation among voice over IP, digital video and data traffic transferred between the cities. Logical network diagram of scenario is described below:

Figure 1: logical topology.



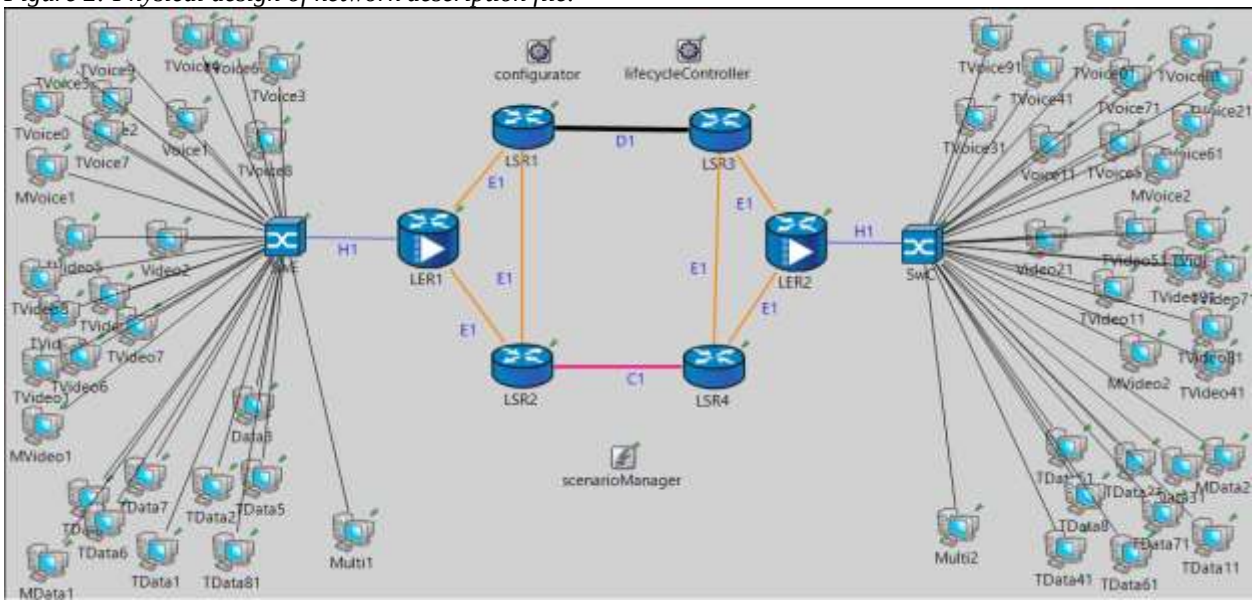
Distance between core network devices located in Edmonton and Calgary cities is calculated as 300 km and devices between cities are interconnected via single mode fiber optical media. As typical single mode optical fiber has a core diameter between 8 and 10.5 µm, we are assuming it 9 µm and value of refractive index of the light in glass is 1.4. Using these values and wavelength as 1320nm, propagation delay in fiber optic in 300 km distance is equal to 1.47ms (milliseconds). Considering processing delay at middle multiplexors and amplifiers delay value between the cities is taken as 1.5ms in this simulation. Primary link between the cities is link named D1 (LSR1←→LSR3) with 1.5ms and alternate backup link is C1 (LSR2←→LSR4) which

has a bit higher delay (2ms) than D1. Data rate of the both channels is 5mbps. Link among the network devices of provider inside the city is marked as E1 with data rate of 2Mbps and delay 0.5ms. SwE and SwC are edge devices of customers which are connected to the label edge routers of provider in both cities via H1 links with data rate of 2Mpbs and 1ms propagation delay. Workstations transmit data with 100gbps speed without delay.

Delay and data rate values are configured under network description file according to simulation requirements in order to create congestion in ingress and egress interfaces of transmitting routers. Group of voice, video and data host start transmit simultaneously using the same send-interval value and as a result links gets congested with different class type of traffic.

The nodes having IPv4 network layer should be configured at the beginning of the simulation. This configuration can be manual or automatic. IPv4NetworkConfigurator module of INET framework has been used to automatically configure IP addresses and gateway on customer hosts and edge devices. Configurator module supports both fully manual and fully automatic configuration. It can also be used with partially specified manual configurations and fills the gaps automatically. It assigns per-interface IP address, strives to take subnets into account and can also optimize the generated routing tables by merging routing entries. This module can be partially or fully turned of particular compound modules. Only IP addresses and default routes of customer side have been configured automatically in simulation, configurator module is disabled on provider's backbone network (LER1, LER2, LSR1, LSR2, LSR3, LSR4). Interface addresses of this devices assigned manually using routing table (*.rt*) files which are written in XML language and remote routes are learned by running interior gateway routing protocol. Physical diagram of the network design built in *CAPSTONE.ned* file is described below.

Figure 2: Physical design of network description file.



NETWORK COMPONENTS

This section summarizes network components used in the demonstrated simulation. Some modules from INET framework are used in simulation and many of them have been changed significantly to meet the requirements of the project.

The INET Framework is an open-source communication network simulation package for the OMNeT++ simulation environment**.** It introduces routing protocols, Ethernet, ppp, qos, mpls implementations and support for simulating node/link shutdowns, auto-configuration, crash and restart, adds TCP, IdealWirelessNic, and contains many other improvements. Ethernet, autoconfigurator, mpls, qos, bgp, linkStateRouting modules of INET framework have been used extensively in building our simulation. Some compound modules of MPLS and QoS simulation, linkStateRouting, routingTable and interfaceTable simple modules have been modified according to simulation aims and objectives.

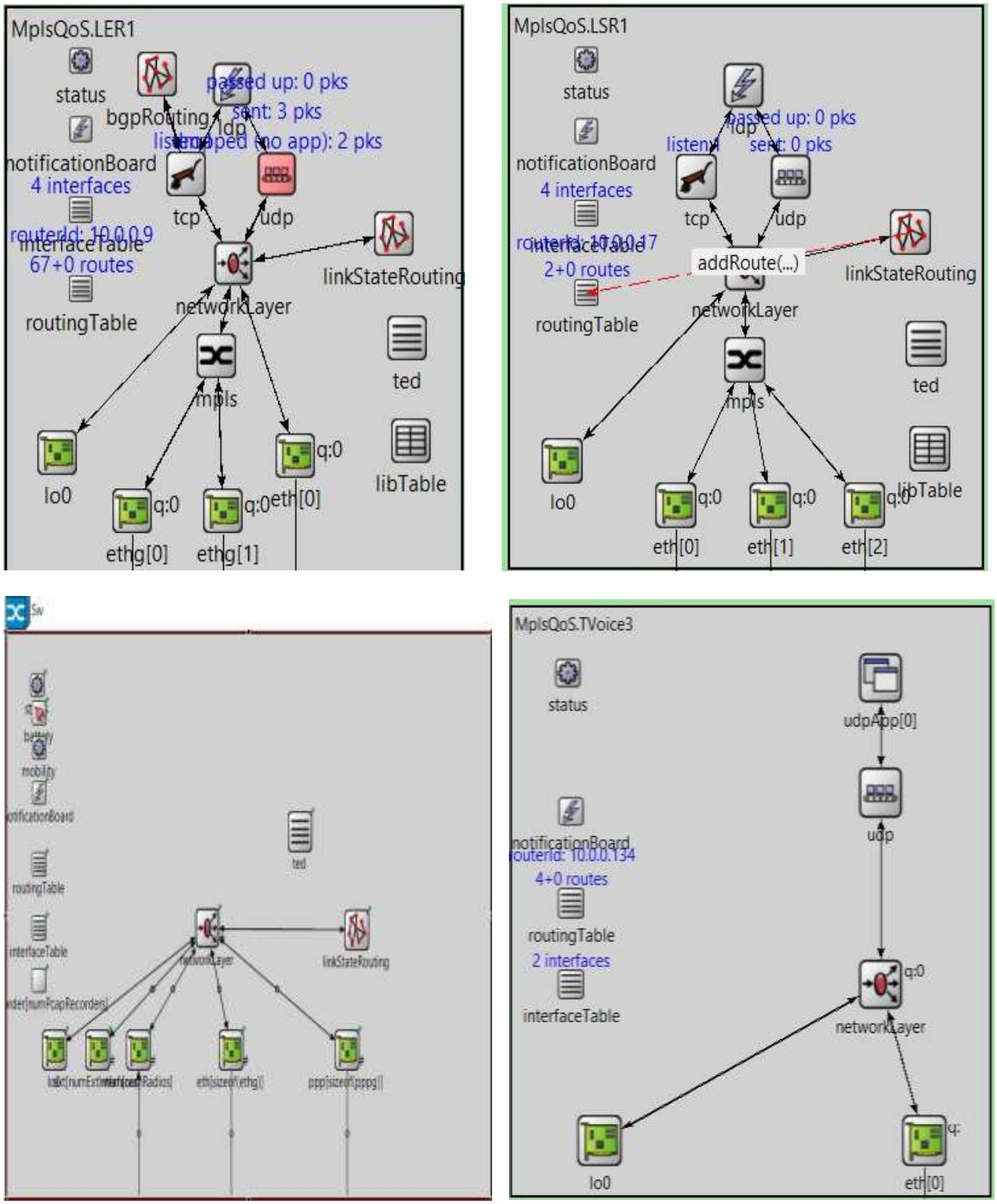
LER1, LER2 – (Label Edge Router) are LDP capable ingress and egress routers of MPLS domain. They label packets on ingress interface, create FEC and remove MPLS shim header while packet is exiting the domain. Both routers are the instances of the same (LER) compound module. LER compound module is modified from the basis of LDP_LSR node module which is the part of MPLS/LDP simulation of INET framework. Ethernet, MP-BGP and modified link-state routing modules have been added to LDP_LSR module. Internal BGP routing module is configured in only edge devices of MPLS domain in order to exchange customer routes with each other throw VPN. Linkstaterouting protocol is running between MPLS nodes to exchange internal routes. Another change is that, MPLS is not enabled in all interfaces of the LER routers, as the port of Ethernet module connected to non-mpls nodes are connected directly to network layer module but not to MPLS simple module.

LSR – An LDP capable router - the main building blocks for an LDP MPLS network. They act as intermediate nodes of MPLS domain forwarding packets based on labels. This module also is modified version of LDP_LSR module with minor change in Ethernet module. In contrast to LER, all interfaces are MPLS/LDP enabled and BGP is not running in these routers. Ipv4NetworkConfigutor module is turned off for both LER and LSR routers. Both modules have MPLS, Linkstaterouting, ldp, tcp, udp, ted and lib modules.

SwE, SwC – are customer edge routers advertising customer routes to label edge routers. These routers are based on NodeBase compound module with minor changes in routing algorithm.

All customer *hosts* are instances of StandardHost module from INET framework without any changes as this module has satisfactory features for our simulation needs.

Figure 3: Grpahical structure of network nodes (LER, LSR, SW, Host).



Ethernet - IWiredNIC module which is extension of NIC module of INET framework is used in simulation for Ethernet data link layer implementation. It contains a MAC, LLC models as well as a bus (for coaxial cable) and a hub (EthernetHub) model. The Ethernet model operates according to the following standards:

- Ethernet: IEEE 802.3
- Fast Ethernet: IEEE 802.3u
- Full-Duplex: IEEE 802.3x CSMA/CD
- Gigabit Ethernet: IEEE 802.3z

The nodes of the Ethernet networks are connected by coaxial, twisted pair or fibre cables. These cables are represented by connections in the INET framework. The connections used in Ethernet LANs should be derived from *DatarateConnection* channel and can have their delay and data-rate parameters set. By default data-rate parameter can have only four values: 10Mbps, 100Mbps, 1Gbps and 10Gbps. MAC implementation allows Ethernet module to operate only in full-duplex mode by default this parameter is true in its NED definition, so there is no collision detection, retransmission with exponential back-off, carrier extension and frame bursting.

Ethernet module with default settings is not suitable for our simulation, because simulation requires creation of congestion in all interfaces of nodes which supports minimum 10Mbps data-rate. Generating 10Mbps in each site requires hundreds of host transmitting real packets or few hosts sending and receiving simulated unrealistic packets. In order to keep simulation hosts realistic, some changes have been done in C++ library of Ethernet NIC module so it supports 2Mbps data-rate which we are using in simulation. Modification has been done in *ETHERNET_TXRATE* function of *Ethernet.h* library file. Parameters of channels and host will be demonstrated in "Simulation Parameters" section of this paper.

ScenarioManager - ScenarioManager is used for setting up and controlling simulation experiments. We can schedule certain events to take place at specified times, like disconnecting link, changing a parameter value, putting node to shutdown, changing the bit error rate of a connection, removing or adding connections, removing or adding routes in a routing table, etc, so that we can observe the transient behaviour. ScenarioManager executes a script specified in XML. It has a few built-in commands, while other commands are dispatched to be carried out by given simple modules. I have used ScenarioManager module to fail the link between LSR1 and LSR3 at certain time of simulation (2s) in order to measure recovery time and impact of the failure to the delay. ScenarioManager has also been used to change link attributes and parameters after the link failure.

LifecycleController - Manages operations like shutdown/restart, suspend/resume, crash/recover and similar operations for nodes (routers, hosts, etc), interfaces, and protocols. Mainly used by ScenarioManager for making changes and interrupting simulation while it is in progress.

LinkStateRouting – this module introduced with the INET framework MPLS implementation. This routing protocol has been used as an IGP in MPLS backbone network in our simulation.

The module implements a very minimalistic link state routing protocol. Apart from the basic topology information, the current link usage is distributed to all participants in the network. When a router receives a link state packet, it merges the packet contents into its own link state database (ted). If the packet contained new information (ted got updated), the router broadcasts the ted contents to all its other neighbours; otherwise (when the packet didn't contain any new info), nothing happens. Announce messages are used to build neighborship and exchange routing topology with neighbors. When the announceMsg timer expires, linkStateRouting sends out an initial link state message. The "request" bit in the message is set then, asking neighbours to send back their link state databases. By default it happens only once at the beginning of the simulation which is not suitable for our scenario, as we want our routing protocol to recover after the link failure. I have done some minor changes in the handle message function of *linkstaterouting.cc* library file to exchange announce messages after the link failure occurs. Link-state messages updated regularly and when changes occur after the change in code. Similar code changes have been made in *RoutingTable.cc* and *InterfaceTable.cc* library files as they initialize based on self-refresh messages at predefined time intervals to recover after link failure. Change done in LinkStateRouting module is attached below:

```
void LinkStateRouting::handleMessage(cMessage * msg)
    if (msg == announceMsg)
   {
      initialize(4);        // Modified to initialize protocol after trigged messages are received
{
      delete announceMsg;
      announceMsg = LINKSTATE_UPDATE_PACKET;

          //Has been added to create another announce messages after initial announce messages processed

      announceMsg = new cMessage("announce");
      scheduleAt(simTime(), announceMsg);
      sendToPeers(tedmod->ted, true, IPv4Address());
```

Below given example is from RoutingTable module to call initialize at 2.009ms. InterfaceTable module also has been changed similarly but initialize function is called at 2.005ms.

```
refreshTopologyChangeMsg = new cMessage("Refresh");
scheduleAt(simTime() + 2.009, refreshTopologyChangeMsg);
```

void RoutingTable::handleMessage(cMessage *msg)

```
if(msg->isSelfMessage()) {
      initialize(0);
      initialize(1);
      initialize(2);
      NodeStatus *nodeStatus = dynamic_cast<NodeStatus *>(findContainingNode(this)->getSubmodule("status"));
          bool isNodeUp = !nodeStatus || nodeStatus->getState() == NodeStatus::UP;
          if (isNodeUp)
             configureRouterId();
```

Based on network diagram and node configuration, traffic flow from all hosts follows the primary - SwE←→LER1←→LSR1←→LSR3←→LER2←→SwC path

SOLUTION FOR MPLS

MPLS module is placed between the network layer and network interface cards (NIC) and implements label switching. Interfaces connected to customers should connect network layer module directly, while gates connected to MPLS nodes should connect to MPLS module. MPLS module is connected to NetworkLayer simple module. MPLS packets are represented by the MPLSPacket class in OMNET++. As MPLS module is required to interact with L2 (Link Layer) and L3 (Network Layer) in the OSI model, it should be transparent to signalling protocols. Signalling protocol currently implemented in this simulation is label distribution protocol (LDP). LDP module handles process messages. Label distributing protocol automatically generates labels and announces them to all neighbours after MPLS and LDP modules are initialized. The LDP protocol is used by one mpls node to inform another node of the label bindings it has made. The router uses this protocol to establish label switched paths through a network by mapping network layer routing information directly to data-link layer switched paths.

LDP and LIB modules together with MPLS module has been implemented in this simulation. Default value of LDP hold and hello time is used as default, 15s and 5s respectively in OMNET++. MPLS and LIB modules are used with their default setting while LDP hold and hello intervals has been changed to 3s and 2s respectively because of simulation time limit. All LDP message types are sub-classed from LDP packet, and include hello, notify, label request, and LDP address.

The local LIB (Label Information Base) is stored in a LIBTable module. LIBTable module stores the LIB (Label Information Base), accessed by MPLS and its associated control protocols via direct C++ method calls. For most of the time, the MPLS module will do the label swapping and message forwarding. Upon receiving a labelled packet from another LSR, the MPLS first extract the incoming interface and incoming label pair, and then look up the local LIB table. If an outgoing label and an outgoing interface can be found, the module will perform appropriate label operations (PUSH, POP, SWAP) based on the "out-label" vector containing label and operation pairs.

MPLS needs to obtain label information from the LIB component and label query result from the LDP module. Messages in the model communicate with Data Link and Network layers are L2 packets and IP native packets. Specifically, MPLS module encapsulates IP packet and is encapsulated in L2 packet. Different L2 protocols may require different methods of encapsulation to inherit the L2's QoS. This implementation follows a generic approach; it assumes no information of QoS from the link layer. QoS information is added in layer 3 interface towards to customer in MPLS domain an in customer edge devices itself while packets transmitted throw IP interface to overcome this issue.

SOLUTION FOR DIFFERSERV QoS

The goal of QoS is to provide guarantees on the ability of a network to deliver predictable results. Elements of network performance within the scope of QoS often include bandwidth, delay, jitter, availability and drop rate. The differentiated service approach to providing quality of service in networks employs a small, well-defined set of building blocks from which a variety of aggregate behaviors may be built. A small bit-pattern in each packet, in the IPv4 ToS octet or the IPv6 traffic class octet, is used to mark a packet with DSCP, layer 3 value to receive a particular forwarding treatment, or per-hop behavior, at each network node.

Collection of packets that have the same DSCP value in them, and crossing in a particular direction is called a Behavior Aggregate (BA). Packets from multiple applications or sources can belong to the same BA. Based on that BA, each class of forwarded packet is treated the same at all nodes of the network which is defined as a per hob behaviour (PHB). In more concrete terms, a PHB refers to the packet scheduling, queuing, policing, or shaping behavior of a node on any given packet belonging to a BA. Four standard PHBs are available to construct a DiffServ enabled network.

- Default PHB (Best effort) - specifies that a packet marked with a DSCP value of `000000' gets the traditional best effort service from a DS-compliant node. This PHB is applied by default also if packet DSCP value is not mapped to any behaviour class.
- Class-Selector PHB - the DS field values of xxx000 are reserved as a set of Class Selector code-points. The PHBs which are mapped to by these code-points must satisfy the class Selector PHB requirements in addition to preserving the default PHB requirements on code-point 000000.
- Expedited Forwarding PHB - the key ingredient in DiffServ for providing a low-loss, low-latency, low-jitter, and assured bandwidth service. Applications such as VoIP, video, and online trading programs require a robust network-treatment. EF can be implemented using priority queuing, along with rate limiting on the class (formally, a BA). Although EF PHB when implemented in a DiffServ network provides a premium service, it should be specifically targeted toward the most critical applications, because if congestion exists, it is not possible to treat all or most traffic as high priority. EF PHB is especially suitable for applications (like VoIP) that require very low packet loss, guaranteed bandwidth, low delay and low jitter. The recommended DSCP value for EF is `101110'.
- Assured Forwarding PHB - defines a method by which BAs can be given different forwarding assurances. Traffic can be divided into 4 groups, with different priority or bandwidth percentage. The AFxy PHB defines four AFx classes: AF1, AF2, AF3, and AF4. Each class is assigned a certain amount of buffer space and interface bandwidth. AFx class

can be denoted by the DSCP `xyzab0,' where `xyz' is 001/010/011/100, and `ab' represents the drop precedence bits.

DifferServ QoS accomplish its tasks by using these four techniques.

- Classification and Marking
- Policing and Shaping
- Queuing
- Congestion Avoidance

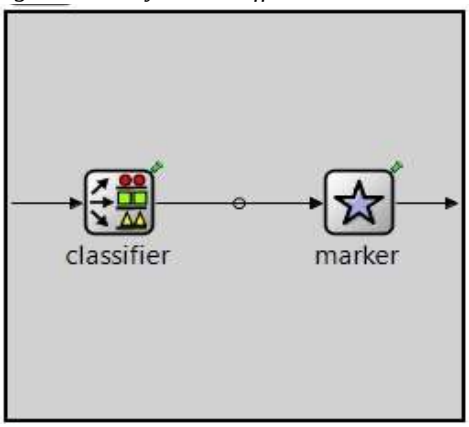
Classification - is defining packet type by checking DSCP values and determining application type to ensure network devices know how to identify and prioritize data as it traverses a network.

Marking - DSCP values matching PHB needs are marked with appropriate class and sent to next hop over egress interface with defined mark so that packet will get the treatmet for its class.

INET provides classfier, meter and marker modules which can be composed to build a traffic conditioner as a compound module to do classifing and marking. Traffic conditioners have one input and one output gate whic is defined in the ItrafficConditioner interface in OMNET++. They classify the received packets according to their content (soruce/destination adress, port, protocol, DSCP filed, label) and forward them to the corresponding output gate.

MultiFieldClassifier and marker simple modules have been used for trafic classification and marking in this simulation. *MultiFieldclassifier* contains a list of filters that identifies the flows and determines their classes. The first matching filter determines the index of the out gate. If no matching filter is found, then the packet will be sent through the default output gate. *Marker* simple module marks packets by setting their fields to control their futher processing. *EntryMark* compound module from One-domain simulation of INET is used for trafic marking in the edge network nodes in our simulation, as it meets requires needs.

Figure 4: EntryMark traffic MultiFieldclassifier and marker.



This classifier is enabled in the ingress interfaces of SwE, SwC, LER1 and LER2 routers to mark IP packets leaving IP domain and entering to MPLS domain. Classifier marks IP packets based on port numbers not based on IP DSCP values in our simulation for simplicity. All hosts are

using the same application “UDPBasicBurst” to send traffic, as end-to-end delay can be measured only if data is sent using this application in OMNET++. IP DSCP field of all application will be same in this case, so *Filters.xml* is used by MultiFieldclassifier to differentiate incoming packet’s QoS classes. Voice, video and data hosts are using different source and destination port number on packet transmission: voice hosts use port 2000, video host use 1000, and best effort traffic uses 2020 and 2021 ports to create sockets. *Filters.xml* file classifies traffic by checking the destination port number and forcing MultiFieldclassifier to transmit defined behaviour class from appropriate output gate on egress side. Gate “0” is used by behaviour class for EF marked traffic, gate “1” is used for AFxy marked traffic while rest of the traffic, undefined in filters file is send throw gate “3” which is for BE service. In other words if packet has destination port 2000, it will be classified and treated as EF forwarding class. Packets with destination port number 1000 will be marked and treated as AF11, rest of the data traveling in network is considered to be BE traffic, which will be forwarded throw gate “3”

Figure 5: EntryMark parameters and filters.xml file.

```
**.LER?.eth[*].ingressTC.classifier.filters = xmldoc("filters.xml", "//experiment[@id='1']")
**.Sw?.eth[*].ingressTC.classifier.filters = xmldoc("filters.xml", "//experiment[@id='1']")#
**.LSR?.eth[*].ingressTC.classifier.filters = xmldoc("filters.xml", "//experiment[@id='1']")
**.LSR?.eth[*].egressTC.classifier.filters = xmldoc("filters.xml", "//experiment[@id='1']")#
**.LER?.eth[*].egressTC.classifier.filters = xmldoc("filters.xml", "//experiment[@id='1']")#
**.Sw?.eth[*].egressTC.classifier.filters = xmldoc("filters.xml", "//experiment[@id='1']")#
**.ingressTC.numClasses = 4
**.egressTC.numClasses = 4
**.ingressTC.marker.dscps = "EF AF11 AF21 BE"
**.egressTC.marker.dscps = "EF AF11 AF21 BE"
```

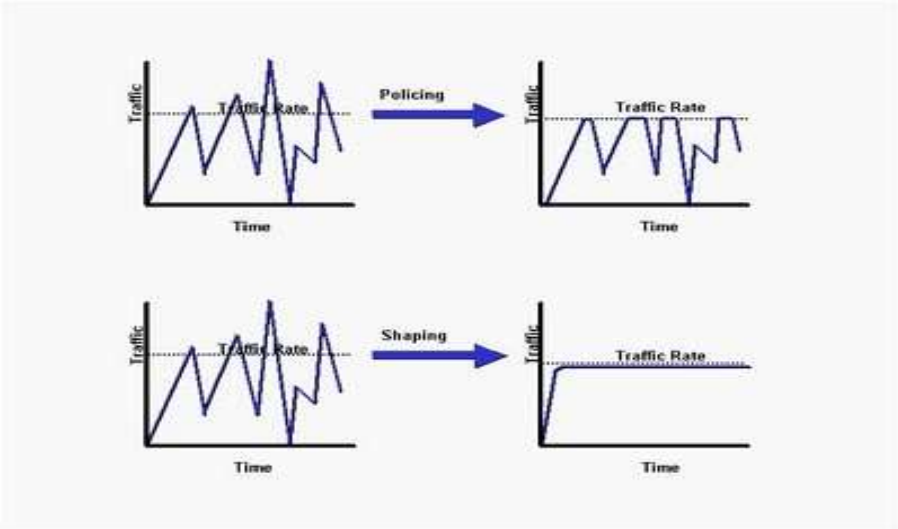
```
<filters>
  <experiment id="1">
    <filter destPort='2000' gate='0'/>
    <filter destPort='1000' gate='1'/>
  </experiment>
</filters>
```

Policing and *shaping* are commonly used QoS technologies to limit the bandwidth utilized by administratively defined traffic types. Traffic *policing* propagates bursts. It enforces bandwidth to a specified limit. If applications try to use more bandwidth than they are allocated, their traffic will be remarked or dropped. In other words, when the traffic rate reaches the configured maximum rate, excess traffic is dropped (or remarked) in order to conform to that specified rate. The result is an output rate that appears as a saw-tooth with crests and troughs.

Shaping involves delaying packets that exceed a software defined threshold. If more data is trying to be sent than the shaped threshold, packets are held in the buffer until a later time when they can be sent out. This has the effect of smoothing outbursts, but unlike policers, shaping will attempt to not drop packets in the process. Queuing polices can be assigned to the shaped buffer to prioritize applications as they leave the buffer. If there is congestion or a large amount of traffic, however, the buffer may overflow, causing dropped packets. In contrast to policing, shaping is a

smoothed packet output rate. Below diagram demonstrates differences between shaping and policing matching time and traffic.

Figure 6: Diagram Policing and shaping



Algorithmic droppers selectively drop received packets based on some condition in OMNET++. The condition can be either deterministic (bound the queue length) or probabilistic. Other kind of droppers are absolute droppers; they drop each received packet. They can be used in traffic policing. *Sink* module is absolute dropper implemented in INET framework.

The algorithmic droppers are ThresholdDropper, DropTailQueue, REDDropper (Random Early Detection) in INET framework. These modules have multiple input and output gates. Packets that arrive on gate input are forwarded to gate out unless they are dropped. These droppers are connected to schedulers to decide whether drop the packet or not. Schedulers have control over queue length.

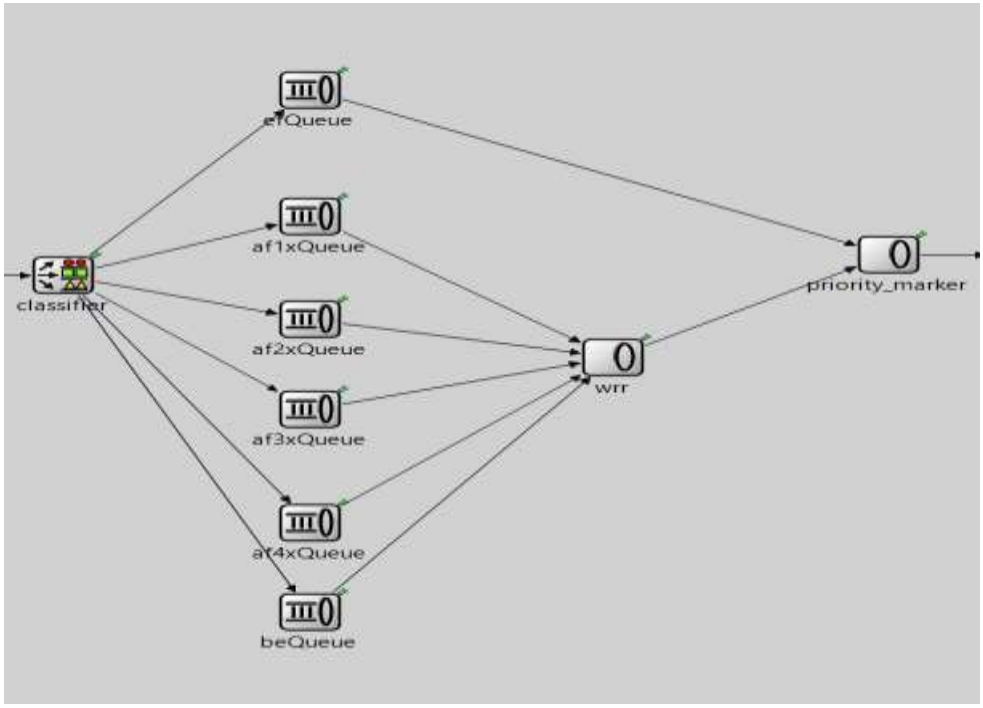
There are several types of *queues* built into network devices. *Priority queues* are designed for use with packets that require low-latency and low-jitter treatment. Each queue can specify a reserved bandwidth that is guaranteed to be available for use by that class during congestion time. Queue depth can also be set to ensure traffic bursts can be handled by a queue.

The WRED (Weighted Random Early Discard) technology provides a congestion avoidance mechanism that will drop lower priority data to attempt to protect higher priority data from the adverse effects of congestion. This deterministic dropping of traffic for class-based queues can reduce the possibility of congestion for all applications. These settings determine the priority of packets to be dropped before the queue becomes full. INET implemented WRR scheduler visits the input gates in turn and selects the number of packets for transmission based on their prioritized weights. If the module has three input gates and weights are 3, 2 and 1, then packets are transmitted in this order: A A A B B C A A A B B C ……. It also efficiently uses the channel – if some traffic is smaller than its share of bandwidth, then rest is allocated to other flows. The WRR Scheduler module is used in this simulation to implement prioritization in between AF and BE traffic.

Another module used for queuing, is DropTailQueue module in current simulation. Its capacity is specified by the frameCapacity in *omnet.ini* or module's network description file. When the number of stored packet reaches the capacity of the queue, further packets will be dropped. Output of this module is connected to WRRScheduler. This module is applied mainly to BE traffic class and partially to AF marked behaviour aggregate. I have limited frame capacity for each traffic classes using network descriptor.

Voice traffic is forwarded directly to output gate without queuing and scheduling in Per Hop Behaviour module of the simulation. Video and Data traffic scheduled and queued before forwarded throw output gate. Queue time and drop precedence is calculated using preconfigured weights of WRRScheduler.

Figure 7: Per Hop Behaviour and its configuration




```
# QoS
**.Sw?.eth[*].ingressTCType = "EntryMark"
**.LER?.eth[*].ingressTCType = "EntryMark
**.LSR?.eth[*].ingressTCType = "EntryMark
**.LER?.eth[*].egressTCType = "EntryMark"
**.Sw?.eth[*].egressTCType = "EntryMark"
**.LSR?.eth[*].egressTCType = "EntryMark"
**.wrr.weights = "5 0 0 0 1"
**.LER*.eth[*].queueType = "PHB"
**.LSR*.eth[*].queueType = "PHB"
**.Sw?.eth[*].queueType = "PHB"
```

SIMULATION PARAMETERS

The network composed of 60 hosts and 8 routers. 22 hosts are devices which send only voice, 18 hosts take part only in video conference and 18 hosts use best effort traffic class in data transmission. Simulated network has two multi hosts; Multi1 and Multi2 which receives data from multiple sources marked with all three type of forwarding classes.

Hosts are connected to the customer edge switch via 100Gbit Ethernet links. The routers are connected by Ethernet links with lower data rates and different delay values. Configuration of link data rate and delay is given below. Traffic shaping performed at the ingress interfaces of edge routers (SwE, SwC, LER1, and LER2). PHB has applied to egress interfaces of all routers.

```
    channel H1 extends ned.DatarateChannel
{
    datarate = 10Mbps;
    delay = 1ms;
}
channel E1 extends ned.DatarateChannel
{
    datarate = 2Mbps;
    delay = 0.5ms;
}
channel C1 extends ned.DatarateChannel
{
    datarate = 5Mbps;
    delay = 2ms;
}
channel D1 extends ned.DatarateChannel
{
    datarate = 5Mbps;
    delay = 1.5ms;
}
```

The voice is sent by an *UDPBacisBurst* application, with mean burst duration 380ms, and sleep duration 240ms (milliseconds). Payload message length is 172 bytes. (160 bytes voice packet payload + 12 RTP headers). After application message is sent down, 20 bytes IP header and 8 bytes UDP header bytes are added at transport and network layers which makes packet 200 bytes long. Data link layer adds its Ethernet frame overhead at least 18 bytes for Ethernet, so minimum packet size becomes 218 bytes when data is transmitted over media. Packets are sent in every 20ms in burst duration which is approximately equal to 64kbps. This data-rate value is the same as G711 and G.722 encoders' bit rate value. Voice encoder is the device that translates analog and digital voice signals. , The G.711 algorithm encodes non-compressed speech streams running at 64 Kbps. G.711 encoded voice is already in the correct format for digital voice delivery in the public phone network or through private branch exchanges.

Video traffic uses encoding compression format, Quarter Common Intermediate Format (QCIF) with data rate of 117kbps. As message length of the host sending video traffic is 600 bytes Packets are sent in every 20ms with host transmitting 252ms burst duration and 496ms sleep duration. QCIF is video format used in videoconferencing systems, which supports signals with

a data rate of 30 frames per second (fps), with each frame containing 288 lines and 352 luminance pixels per line. CIF is part of the ITU H.261 videoconferencing standard. CIF is also known as Full CIF (FCIF) to distinguish it from Quarter CIF (QCIF), a related video format standard that transfers one fourth as much data as CIF.

Data traffic also uses *UDPBacisBurst* application in order to measure end-to-end delay of best effort service. Message length of packet sent by data hosts is configured as 1200bytes in *omnet.ini* in order to create congestion in network. Based on configured parameters, data is sent with 300kbps rate in simulation.

Eleven voice, nine video and ten data hosts begin to transmit data simultaneously at the 25ms of the simulation time after LinkStateRouting converges. Other hosts begin to transmit at 30ms and 35ms simulation time. As a result of transmission on the same time, we will have 2.4 Mbps traffic at each side of the network (Edmonton and Calgary). Hosts are connected to edge node via higher data rate media which will forward the packet to customer edge device with no delay. Interface connecting customer edge router to service provider supports traffic maximum with 2Mbps data rate, so we will have congestion in network from the beginning of the simulation. Congestion will increase linearly according the simulation time as send-interval time of hosts is very short (20ms). In the end will should see delay in all forwarding class as competition will not be only among different forwarding classes, sometimes there will be same type of traffic creating congestion.

DifferServ QoS domain is expected to do traffic shaping in the ingress interfaces of routers and drop data packets and video packets if necessary, but we should not have drops in EF marked voice packets. Internal nodes of the MPLS network will not mark packets once they entered to domain, they will treat each packets according their marked classes. Delay, queuing time queue length observed in data traffic should be higher than video and voice traffic. These variations should be negligible between voice and video traffic as voice packets should have higher bandwidth and less delay and queue time. Main link between cities D1 (LSR1 ←→ LSR3) will fail when simulation reaches 2s. This will be implemented using ScenarioManager simple module.

```
<at t="2s">
    <disconnect src-module="LSR1" src-gate="eth[2]" />
    <disconnect src-module="LSR3" src-gate="eth[0]" />
    </at>
```

Simulation time is configured as 300s.Total sent and received packets, end-to-end delay, queue length, queue time, drop precedence, dropped packets during link failure will be measured after simulation ends.

SIMULATION AND LAB RESULT ANALYSIS

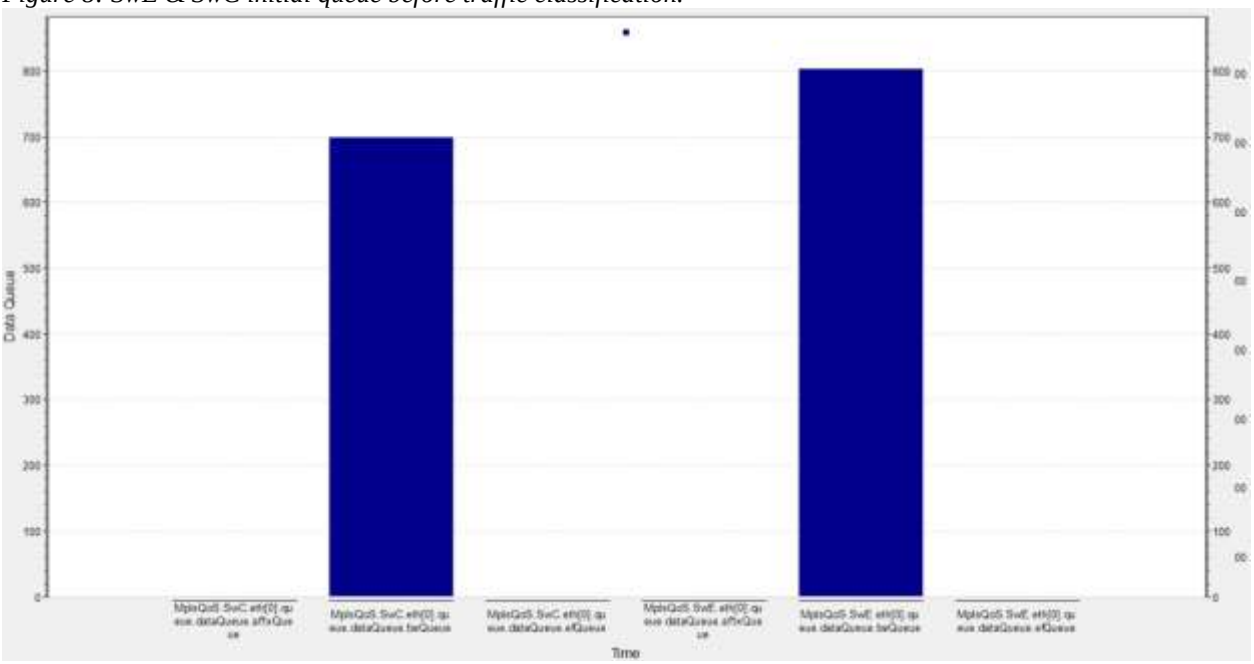
Simulation results will be analysed and discussed in this section. In OMNeT++ 4.3 version the statistical analysis tool is integrated into the Eclipse environment. The results are recorded as scalar values, vector values and histograms. The analysis tool lets us load several result files at once and presents their contents like a database.

Queuing Time Performance

The performance analysis of QoS in term of queue data has been carried out in this part. In DifferServ MPLS and IP network link bandwidth is shared by multiple traffic classes and dividing these classes to prioritized groups is accomplished by queues. Queue refers to the process of buffering incoming packets at the entrance of a communication link. The statistics represents instantaneous measurement of packet waiting time in the transmitter channel queue. Measurement is calculated from the time a packet enters the transmitter channel queue to the last bit of the packet is transmitted.

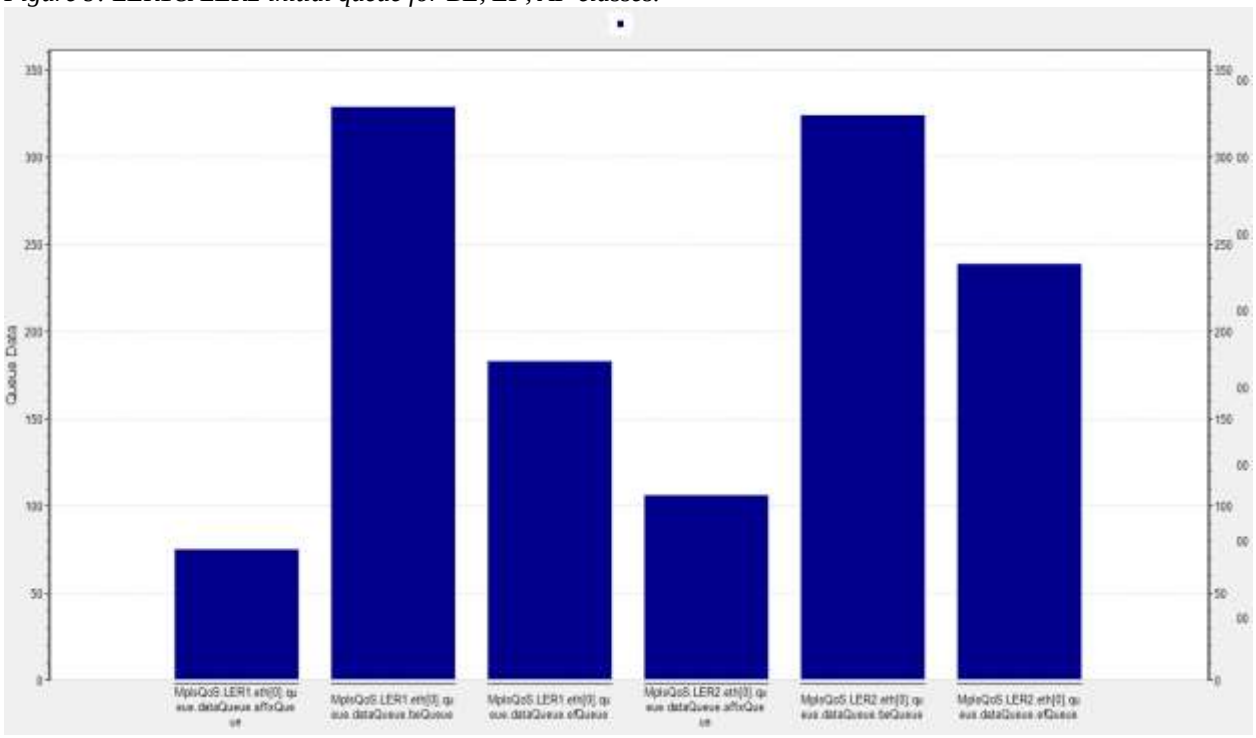
As all the hosts start transmit simultaneously and the data-rate of the link between customer and MPLS edge network link is lower than provider internal links, the highest value of the queuing time is observed in the SwE, SwC, LER1 and LER2 router's ingress interfaces. All packets reaching the SwE and SwC customer edge routers treated as default forwarding class (best effort) before they are marked and sent out throw eth[0] gate. For this reason SwE and SwC routers have only *be-queue* type while packets are waiting in a queue and treated the same. Below we can see queue data value in scalar output file which are received by customer edges. Value of EF and AF traffic is equal to zero

Figure 8: SwE & SwC initial queue before traffic classification.



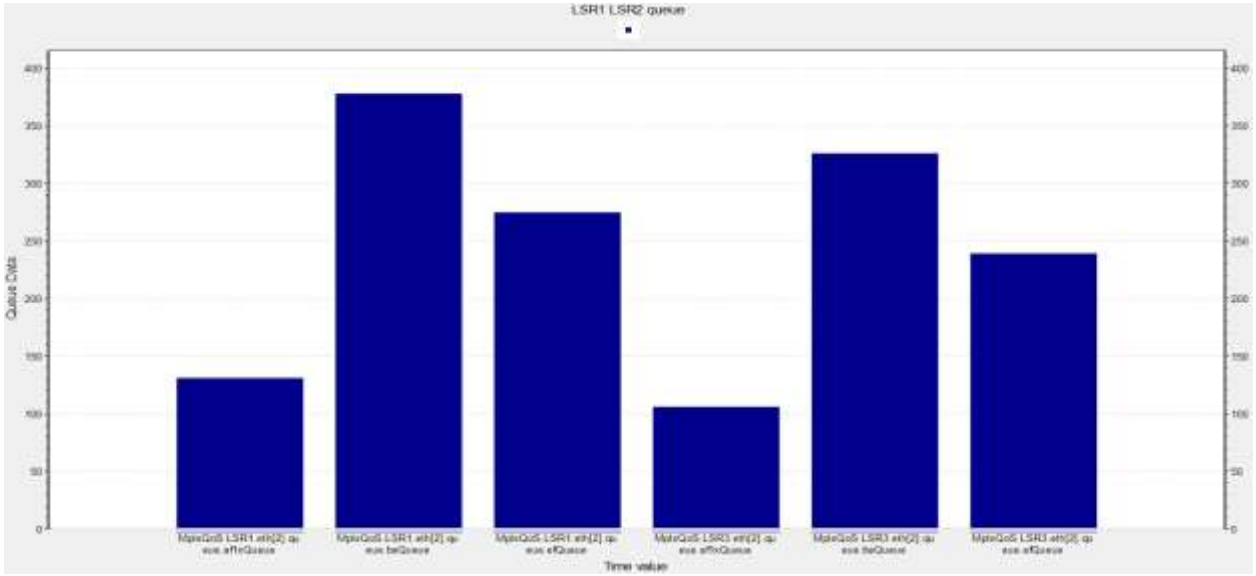
We observe that packets are marked according their forwarding class after they enter to customer edge devices. LER's queue now is classified and both provider edge devices have all three types of forwarding class queue. Best-effort queue is longer than EF and AF because its bigger message length, while EF queue is slightly longer than AF because some video hosts will join later and number of voice host is more than video number of video hosts

Figure 9: LER1& LER2 initial queue for BE, EF, AF classes.



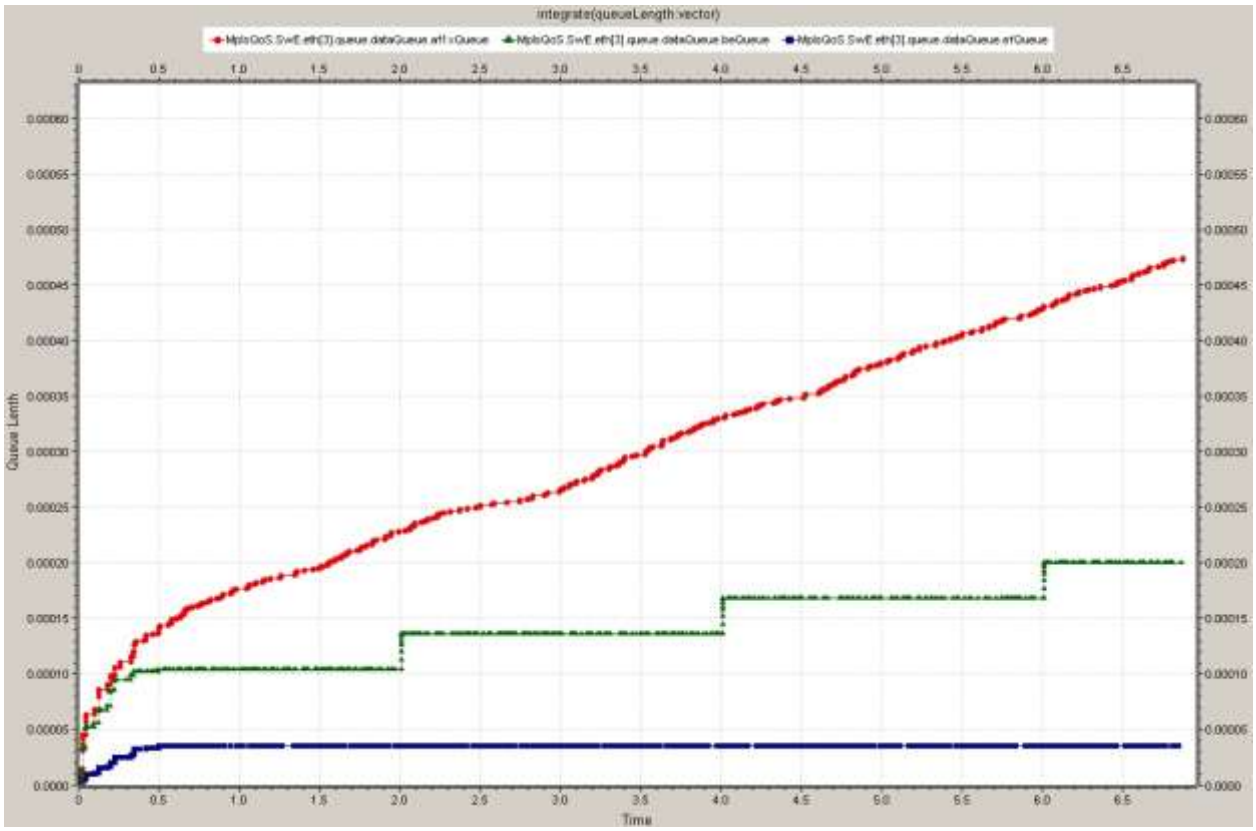
The same value is seen in internal MPLS nodes LSR1 and LSR2 ingress and egress interface queues. Differences between BE, EF and AF traffic remains the same only count value is decreased because data-rate of internal links is higher than SwE→LER1 and SwC→LER2 (H1)

Figure 10: LSR1& LSR3 initial queue for BE, EF, AF classes.



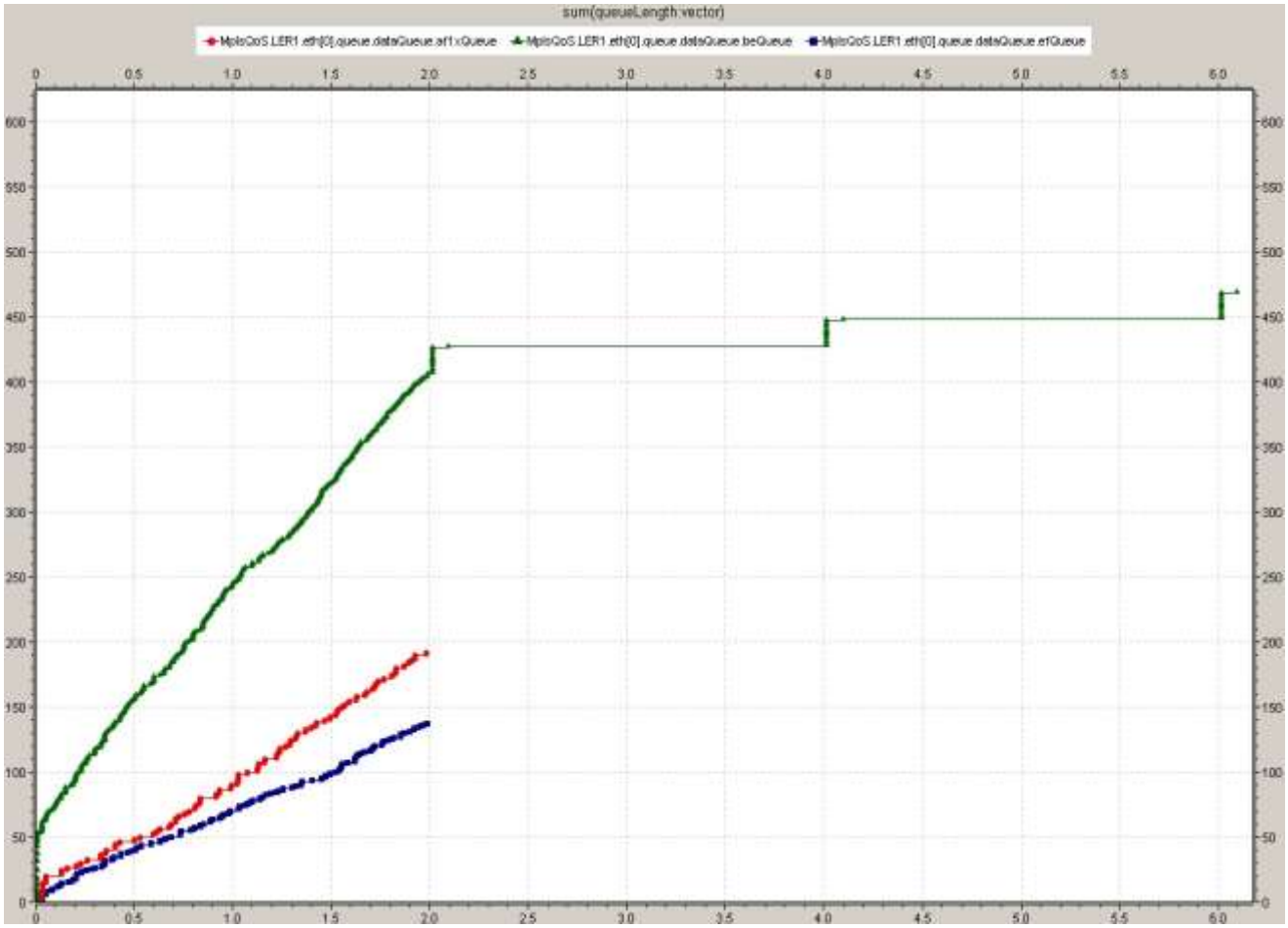
Queuing Time vector results graphically demonstrates the waiting time of each class type in queue before packets are forwarded. We can see that vector indicator of packets waiting in the queue for EF class is minimum while BE class is maximum. The queue length is the same 0.2ms after packets are sent among all three types of classes. After 2ms stabilization is observed in EF queue length while comparing it with others, which means that, it takes first priority and length of other two types of classes increases. AF and BE classes queue length remains the same, 4ms long after transmission until link is congested, then we can see that AF queue takes priority over BE in IP network. The maximum waiting value of EF traffic reaches 0.00005 while this value is 0.0002 for AF and 0.005 for BE behaviour aggregate. The queue length increases after each 20ms in AF and BE forwarding classes while queue length of EF traffic is constant after 5ms.

Figure 11: Queue Length comparison in IP network. SwE



The MPLS DifferServ implementation causes slightly different result than IP network in the output of the queue length value indicator. MPLS QoS immediately gives priority to EF and AF behaviour aggregates and differentiates them in the beginning of the transmission while it happens after 4ms in IP network. In MPLE nodes BE traffic length has the highest value, AF traffic value is very much lower than BE and slightly higher than EF traffic. Queue length value is 400 for EF, 190 for AF and 140 for EF behaviour aggregate at the 2ms of burst duration. Below given vector analyses graph will demonstrates differences. We can see 25bytes increase in queue of BE traffic after each 20ms in MPLS nodes as well.

Figure 12: Queue Length comparison in MPLS network. LER1



End-To-End Delay Results

Measuring end-to-end delay between voice, video and data end users is one of the main objectives of the simulation. According to the result outputs we have observed low end-to-end delay in voice traffic which is caused by competition among voice traffics with each other. Most of the voice hosts send traffic simultaneously and the queue fills with EF traffic. In this case delay becomes inevitable for EF forwarding class, as the traffic causing congestion has the same priority value. End-to-end delay is observed in the video traffic, which is slightly higher than voice traffic. This result has two main reasons: video traffic competes with BE traffic as well as EF and other AF traffic, so traffic shaping implementation in PHB module will prefer voice traffic when links are congested. Another reason is the video traffic message length is significantly higher than voice traffic which makes them to wait in a queue more than voice packets before traffic classification occurs at the edge devices. Below given figures 13 and 14 illustrate the end-to-end delay for the data, voice and video hosts 1 and hosts 6 sending data simultaneously. In figure 13 results are presented as histogram graphs while figure 14 is a vector output from analysis file.

Figure 13: End-to-end delay in host 1 instances (histogram)

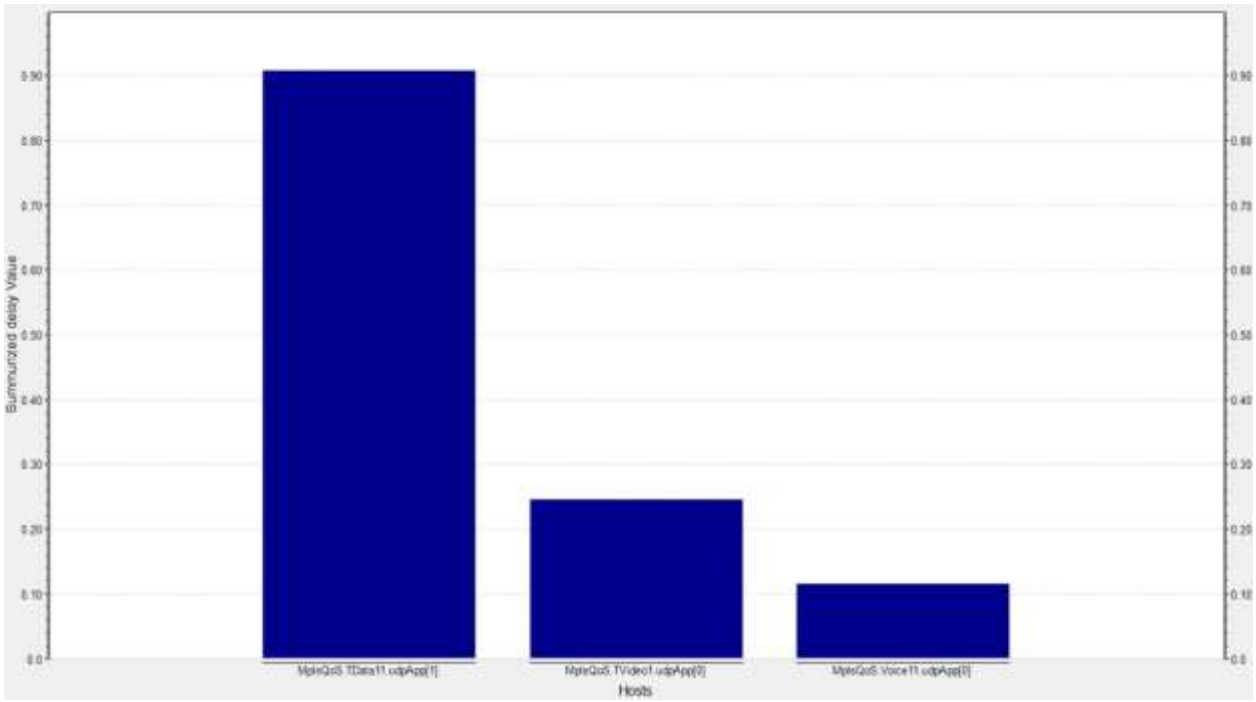
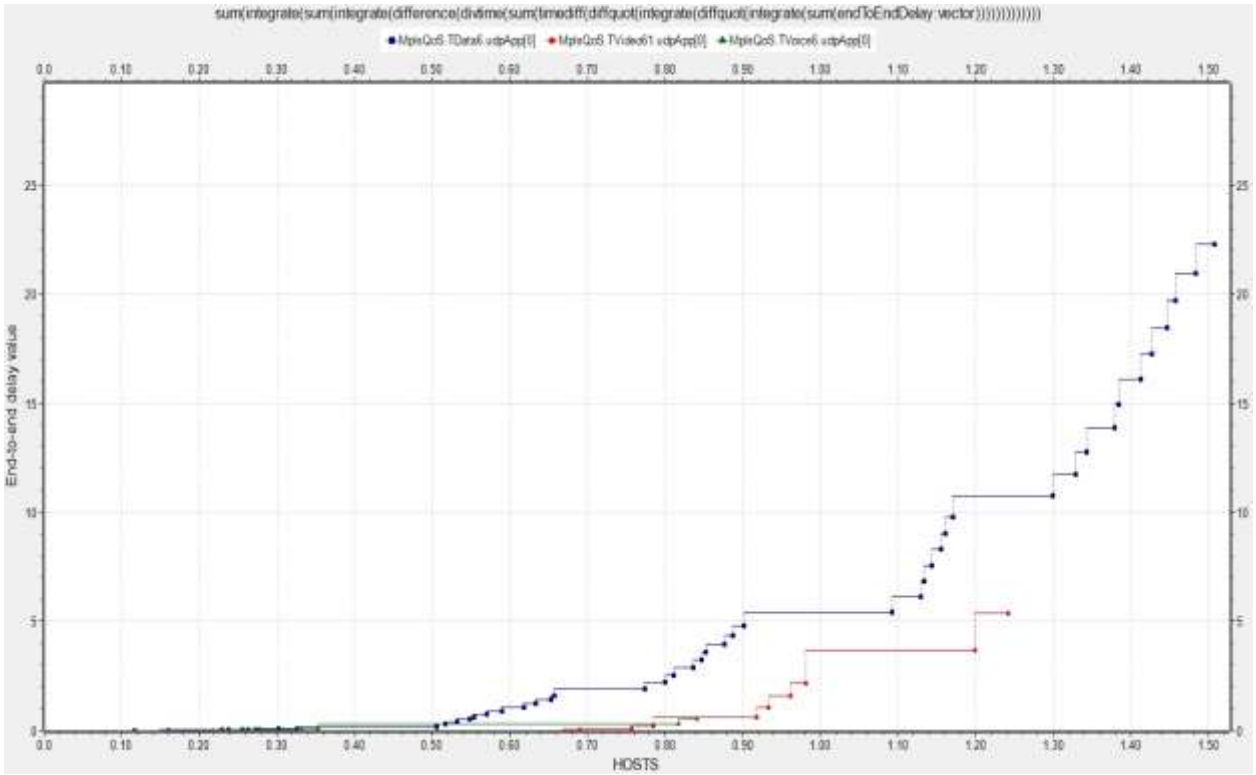


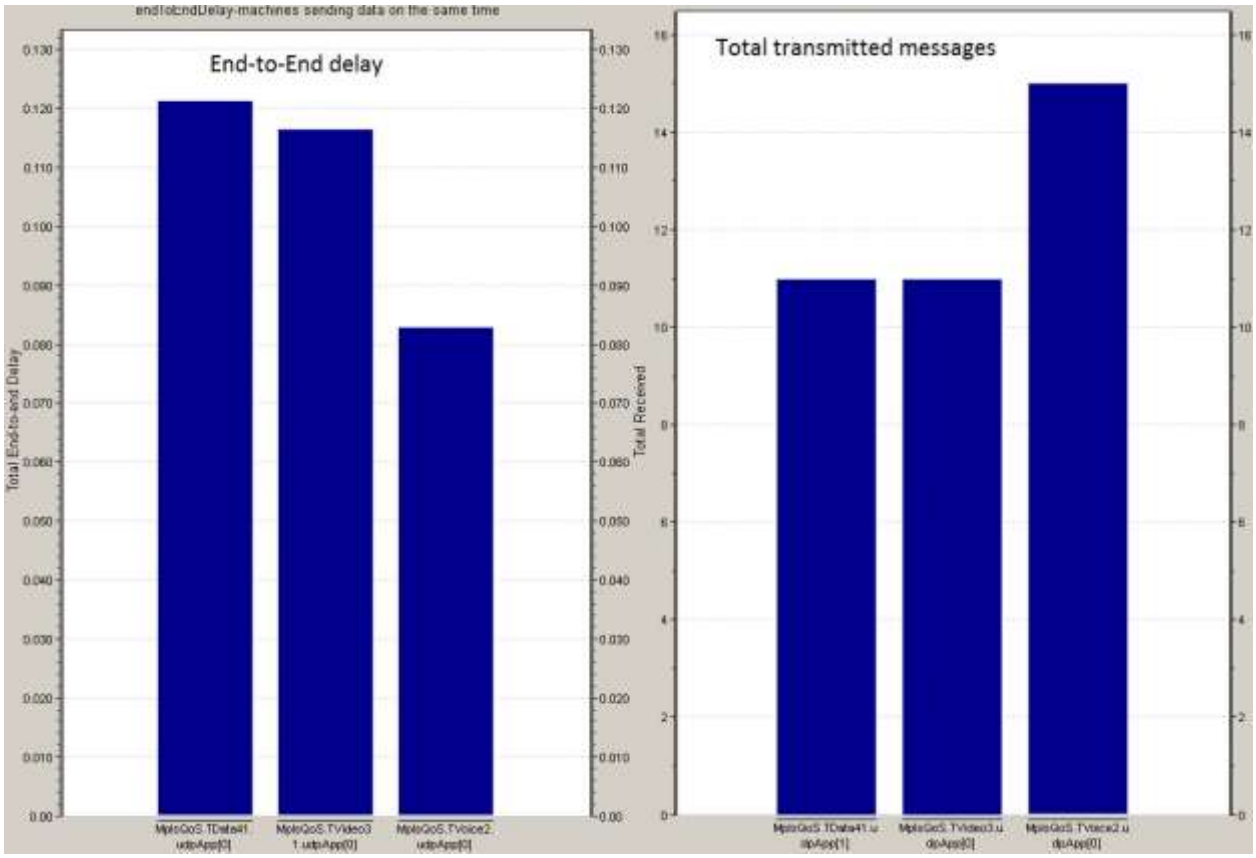
Figure 14: End-to-end delay in host 6 instances (vector)



Blue colored indicator in figure 14 represents end-to-end delay of data host6, red colored line represents video host61 end-to-end delay while green color is used for voice host6 indicator. Delay observed at data host6 reaches 25ms value at congestion time, maximum value of delay at video host6 is 5ms while this value is very negligible in voice host. End-to-end delay value differs among different hosts because of their start time and congestion level of the links.

Figure 15 illustrates graphical histogram comparison of the total transmitted messages against final end-to-end histogram count values for the same hosts. Hosts TData41, TVideo3 and TVoice2 have transmitted different number of messages. Transmitted messages comparison value is the highest at TVoice2, which is equal to 16 while message number of TData41 and TVideo3 is the 11. Low end-to-end delay observed in TVoice2 host comparing to other hosts even it has higher value of transmitted messages. Value of end-to-end delay is equal to 0.8ms at TVoice2, 0.112ms in TVideo3 and the highest value of 0.121 is observed at TData41 host.

Figure 15: End-to-end delay compared with total transmission



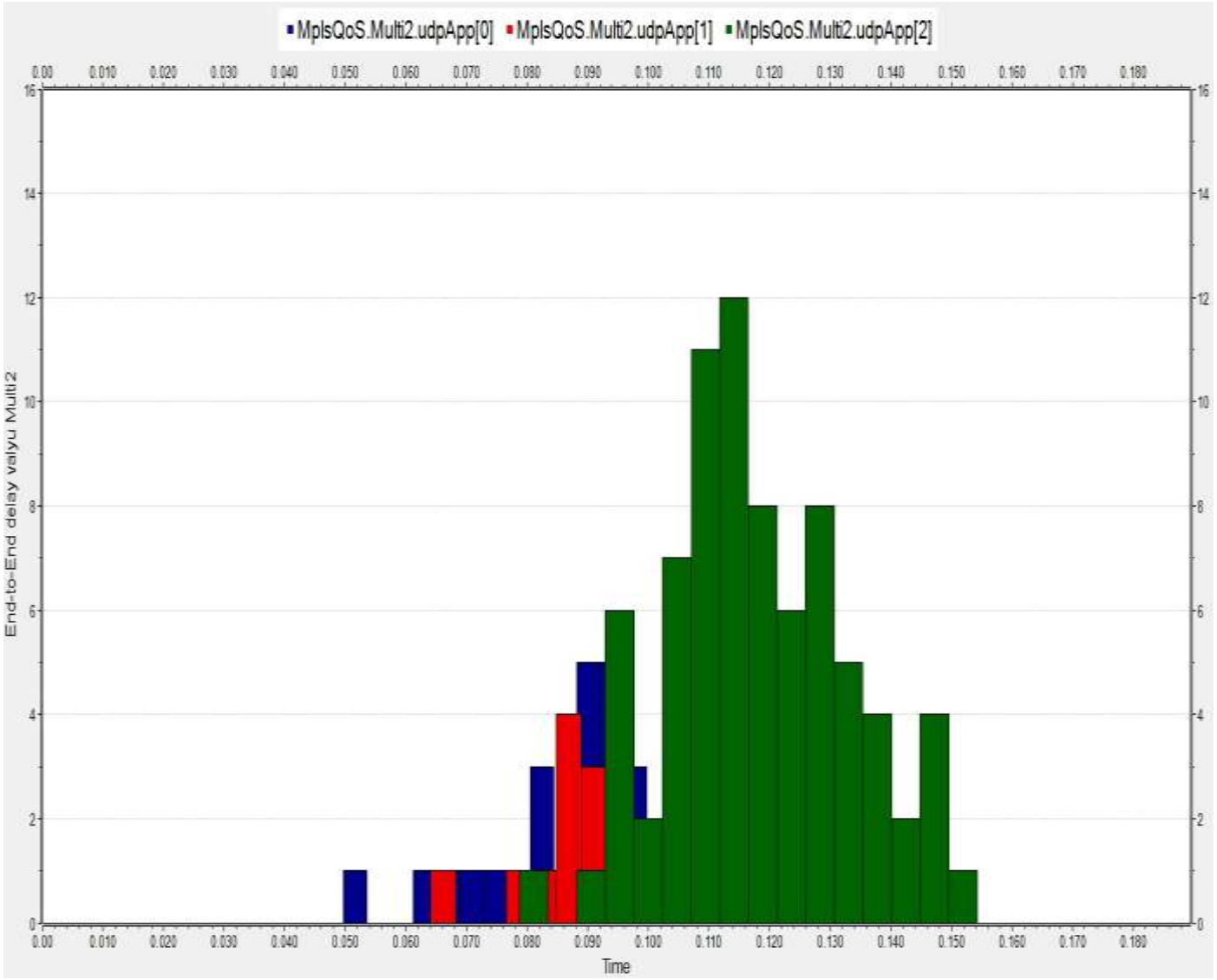
Hosts Multi1 and Multi2 are configured to receive packets in all three forwarding class types. These hosts are not transmitting, they are receivers which receive data from voice, video and data hosts transmitting simultaneously. Three instances of *UDPBACISBURST* application are

running in Multi hosts. UDPApp[0] is voice, UDPApp[1] is video, UDPApp[2] is date receiver application.

```
**.Multi*.udpApp[0].localPort = 2000
**.Multi*.udpApp[1].localPort = 1000
**.Multi*.udpApp[2].localPort = 2020
```

Jitter value for each packet type is measured for comparison in Multi2 host, as the host clients generates traffic on the same time. From output we see that vector value of jitter in voice traffic is minimal, approximately 0.088, this value is observed 0.103 in voice traffic and 0.117 is data traffic. Below illustrated diagram visually reflects the result of jitter in Multi2 host.

Figure 16: Multi2 host Jitter value



Usually VOIP call follows a single path from end to end. That path may include LAN or WAN links, the slowest link presents the available bandwidth for the complete path – which is often referred as a bottleneck of the congestion caused by the slow link. Delay can be measured in either one-way or round-trip delay. To get a general measurement of one-way delay, measure round-trip delay and divide the result by two. One way delay is calculated in simulation results.

VoIP typically tolerates delays up to 150ms before the quality of the call is unacceptable. Voice traffic is predictable in nature as compared to data traffic; apart from bandwidth requirement voice traffic has some additional one way requirements:

1. End to end delay: 150ms or less
2. Jitter: 30ms or less
3. Packet loss: 1% or less

International telecommunication standardization sector (ITU-T) recommends network delays for voice applications in G.114 standard. We observe maximum 50ms end-to-end delay between two voice host instances in this simulation at the congestion time which is acceptable according the standards. The highest delay is equal to convergence time when the primary link fails (132ms). Average delay value is <=10ms for EF marked traffic class.

Table1: ITU-T voice delay standard G.114

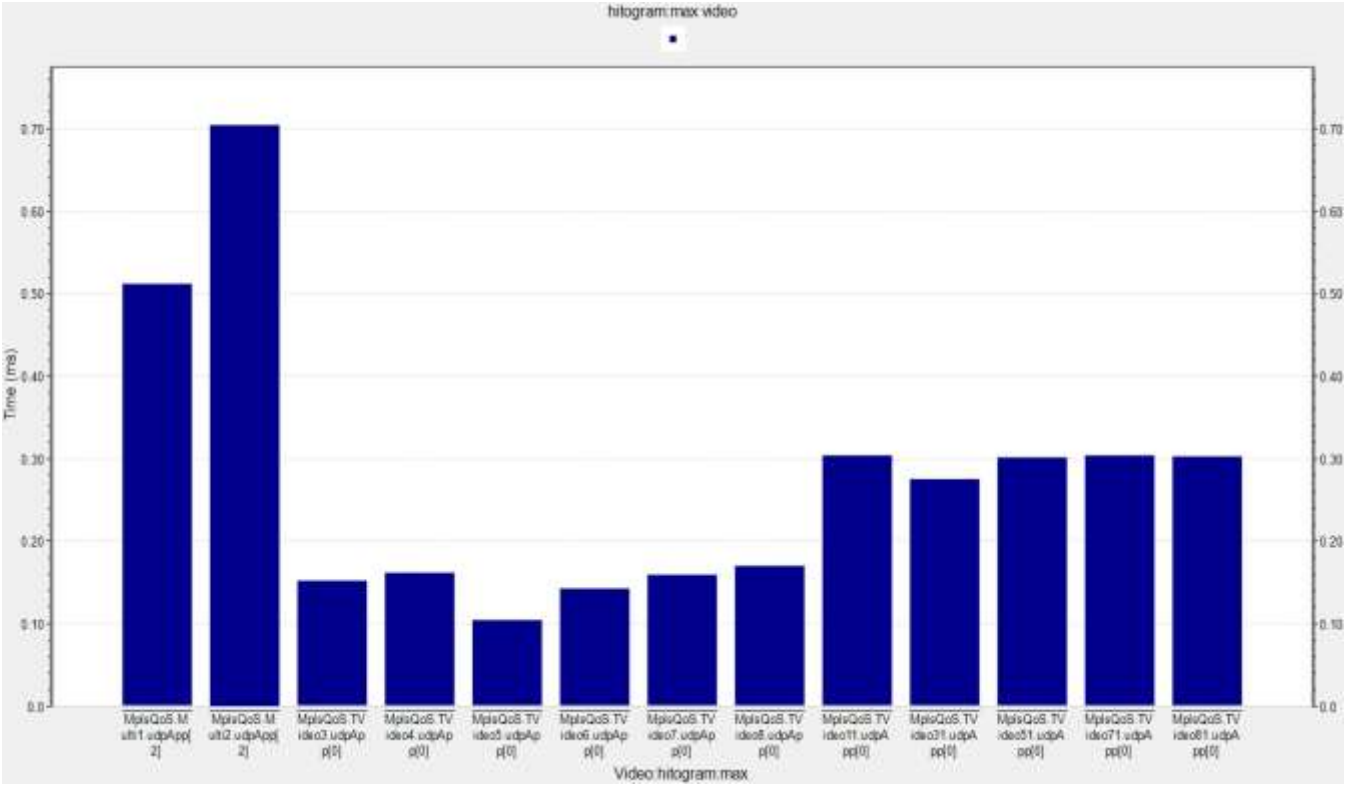
Range - Milliseconds	Description
0-150	Acceptable for user applications
150-400	Acceptable when users are aware of delay impact
Above 400	Unacceptable for general network planning

Video traffic has almost identical requirements of delay as the voice traffic but it consumes a bit more bandwidth and the requirements may vary on how much movement is there in the video. Audio packets tend to be small (480 bytes or less), while video packets are typically large (600-1500 bytes). MPLS intermediate routers may prioritize the two packet sizes differently, creating differing transit times so the audio and video packets become out of sync. A typical rule of thumb for delay is < 300ms round trip between endpoints before users in an interactive call start to notice a delay between the sender and the receipt by the far end participants. Video packets themselves may be differentiated to different groups and treated differently. In some providers, for instance Telus, video traffic originated from different sources (Netflix) is treated as internet traffic and delay value for this traffic can reach up to 100ms. Some video traffic originated from provider itself usually defined higher priority and delay value usually is very low. For example, certain features of the IPTV Middleware environments such as Mediaroom Instant Channel Change (ICC) feature that TELUS uses, requires a one way delay < 30ms for the feature to work reliably.

According the tests done on “Effect of Delay/Delay variable in video streaming” by Blekinge Institute of Technology, delay with 100ms ±4ms delay variance is acceptable by users.

Highest delay value of AF traffic is observed in Multi2 video host with delay value of 70ms. Average delay for video hosts approximately is <=30ms and this value is equal to requirements of instant channel change video streaming technique. Some hosts do not have delay value.

Figure 17: Maximum and average value of delay in video hosts.

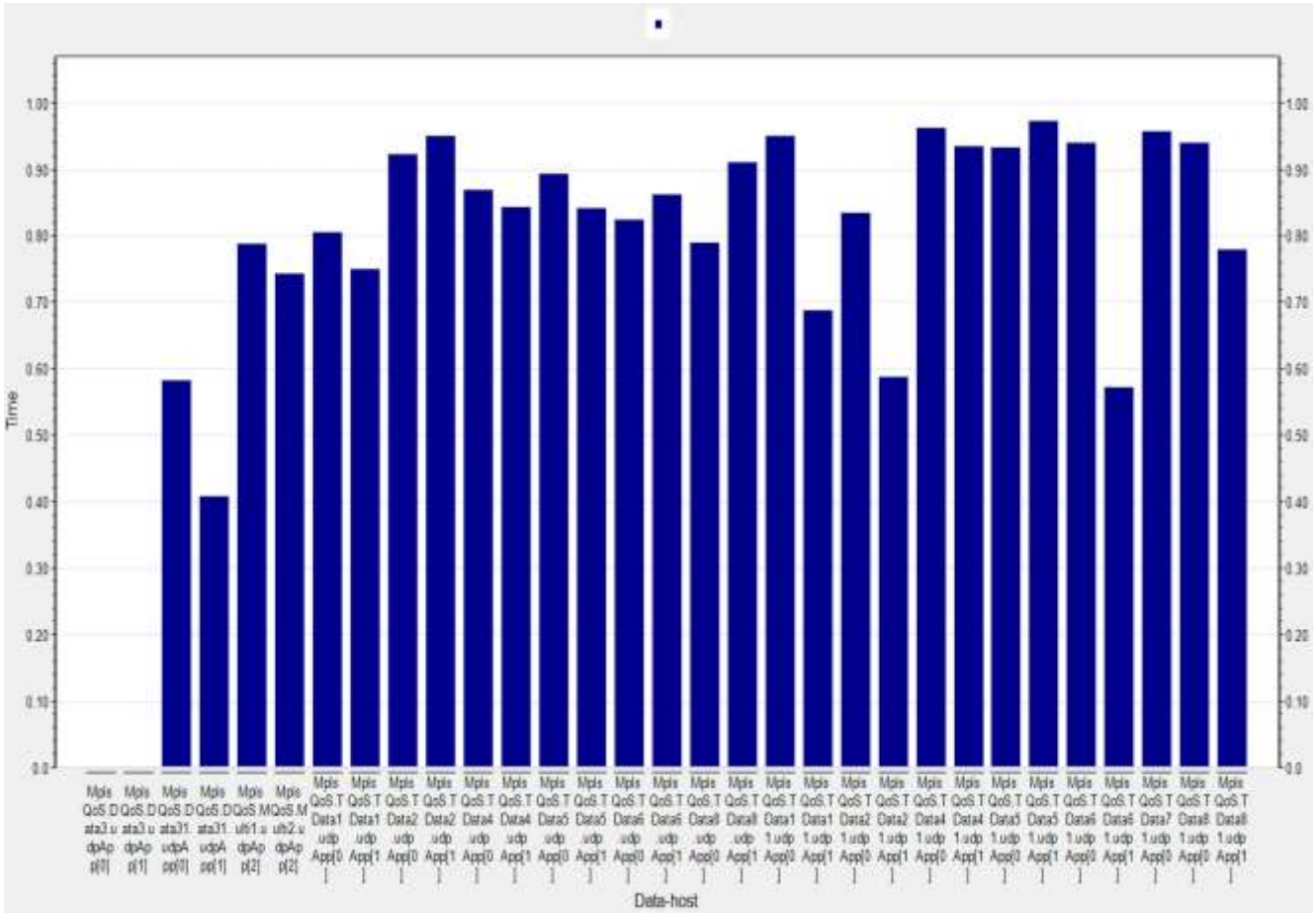


The standards for data traffic is different and most of the time it depends on the user needs and application types. Users on the Internet feel that responses are "instant" when delays are less than 100ms from click to response. Latency and throughput together affect the perceived speed of a connection. However, the perceived performance of a connection can still vary widely, depending on the type of information transmitted and how it is used. According to new research from HTTP Archive, which regularly scans the internet's most popular destinations, the average size of a single web page is now 965 kilobytes. In the past five years from 2008 to late 2012 the average web page grew from 312K to 1114K. Round-trip packet latency over the Internet is fairly low – typically less than a tenth of a second across North America – and an average web page of 30-100 kilobytes would normally transfer fully in 10–30 seconds, over a 56-64kbit/s modem, which yields a 3 KB/s transfer rate. If a user had to wait 10–30 seconds to see anything, after every web-page click, it would be intolerable.

File transport protocol (FTP) can be one example of data transmission. FTP applications use TCP as transmission protocol for encountering packet loss and drops. If TCP will be used with default maximum segment size (MSS), delay friendliness of applications is 360ms for round trip value. This value can grow depending on the type of the transfer. Nowadays number of virtual private network (VPN) users is growing significantly. VPN applications require lower delay than file transport protocol applications for security or speed aspects. One way delay requirement of these applications is =>100ms for some vendors in IP networks.

The highest value of the delay in best effort traffic reaches 98ms in several hosts when links are maximum congested in our simulation. The average delay for data traffic is 75ms while the minimum observed delay value is 40ms.

Figure 18: Maximum and average value of delay in data hosts.

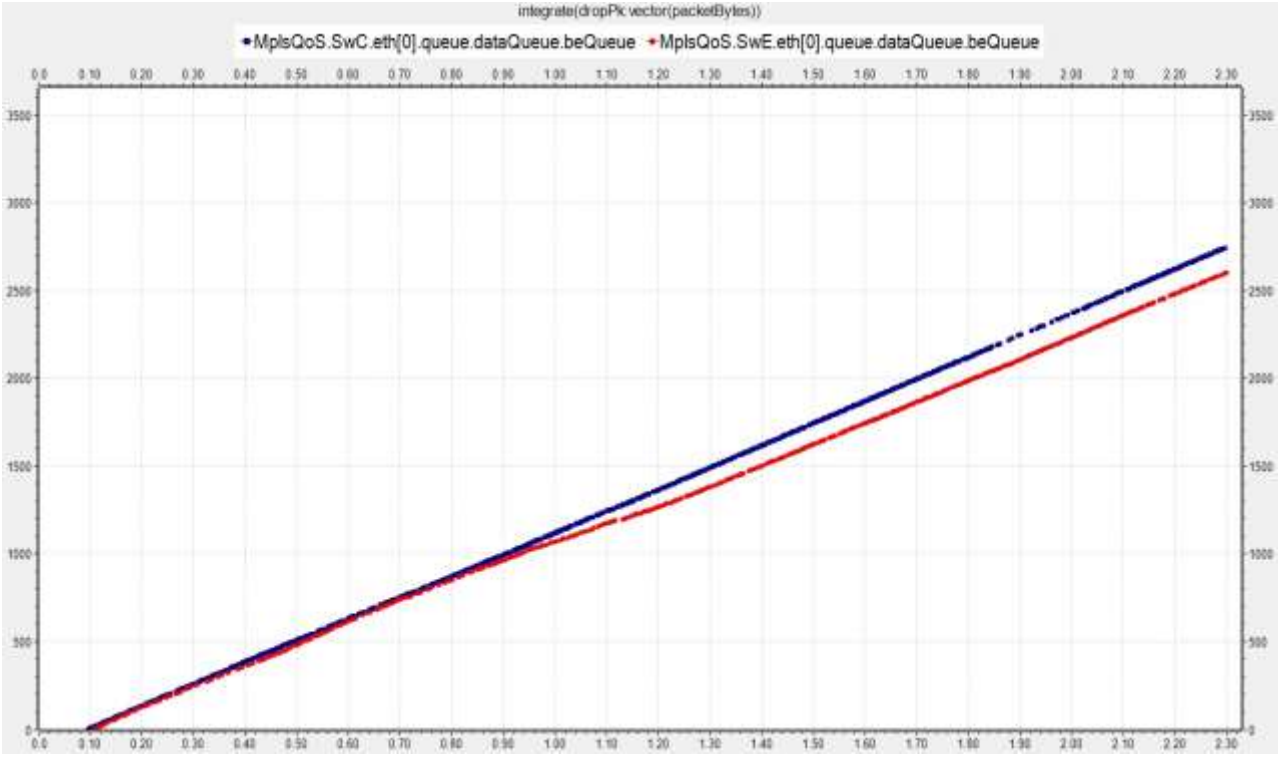


We do not have packet loss in voice and video traffic in the simulation with current parameter configuration. Packet lost is occurred only in data host due to the drops in congestion time. Below discussed drop precedence result reflects packet loss in best effort traffic.

Drop Precedence

Although traffic shaping has implemented to avoid drops on the ingress and egress interfaces of network nodes, packet drop is observed in congestion time. This drop count increases according the simulation time, as hosts continue sending and receiving data all the time. Only BE traffic is dropped while queue length is full and this behaviour is predicted and meets the project needs. Because of its lower forwarding class priority packets having best effort aggregate mark will be dropped to ensure service for voice and data traffic. Packets which belong to EF or AF forwarding class are not dropped in MPLS/DifferServ QoS domain. Drop illustration is presented in scalar line graph in figure 19. Drop in best effort traffic is observed 10ms after all hosts begin to transmit. The indicator of dropped packed number grows according to the simulation time as links becomes congested linearly. We observe 1186 packets drop on SwC out of 163255 packets and this result makes 0.7% of received packets on BE queue. This percentage value is approximately the same in SwE, as 1099 packets have been dropped out of 147198 received packets. We get this result when we define *wrr.weights=5.0.0.0.1* in priority scheduler. It means five packet will be video and sixth one will be data packet in queue. *(V V V V V D V V V V V D V V V V V ...)*

Figure 19: Drops in best effort behaviour aggregate (wrr.weights = 5.0.0.0.1 is used in simulation)



Only 1 packet out of 233534 AF packets gets dropped if scheduler weight is defined as 4.0.0.0.1 in *omnet.ini* file, which makes very negligible percentage of the video packets received from SwE node. Drops in BE traffic changes slightly after this change. These changes in drop value are not reflected in graph.

Statistics at MPLS node LSR1

LSR1 eth [2] gate is transmitting MPLS labeled packets between two sites. Below given table summarizes statistics results on this interface for three forwarding classes

Table 2: Statistics at MPLS node LSR1 eth [2] gate (connected to LSR3).

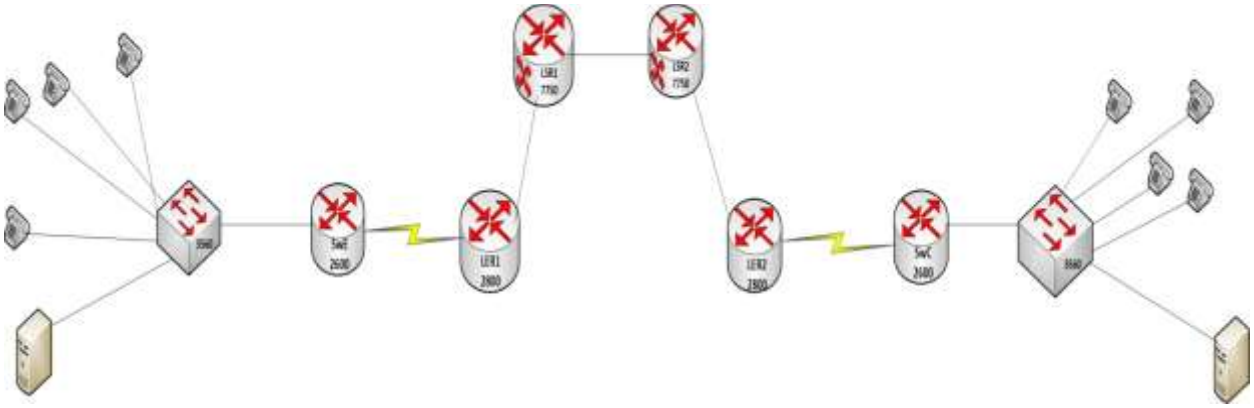
LSR1 eth[2]	EF	AF	BE
Queue Time	5.379	131	378
Queue Length	0.05	1	7
Queue length vector	0.02	0.49	0.59
Received packets sum	55000	84698	431326

We can calculate from this table that, total received bytes in EF class makes 64% of AF traffic and 12% of BE traffic because of message length differences of hosts. When we compare this value against queue time value, EF forwarding class contains only 4.1% queue time of AF traffic and 1.4 % of BE forwarding class. Summary queue length of voice traffic is 95% less than video and 98% less than default data traffic. As we see differences between received packets is significantly lower than queue length and time differences for three forwarding classes.

Test results done in laboratory with real devices

MPLS QoS simulation is repeated in lab using real network devices (Cisco 3560, 2800, 2600 and Alcatel Lucent 7750) for comparison purposes between voice and data traffic. Two Cisco 2900 devices are acting as LER1 and LER2 while other 2600 series router as customer edges. Core of the MPLS domain consists of two directly connected LSR1 and LSR2. Customer edge devices are connected to LER1&2 using serial cable with 64kbps data rate while rest of the network is connected using gigabit Ethernets. Traffic classification is configured on provider edges (LER1 and LER2) and MPLS nodes forward labeled packets applying per hob behaviour class. Below given diagram describes network scenario.

Figure 20: Network diagram built in LAB.



Four Cisco 79411 phones and one computer with fastethernet port are connected to the access switch –catalyst 3560, at each side of the diagram. Cisco phones use G.722 codec by default with 64kbps data-rate. The G.722 ITU Standard is a wideband audio codec that operates at a data sampling rate of. The higher sampling rate allows the G.722 codec to provide higher clarity of audio signals thanG.711. The G.722 codec can provide 7 kHz wideband speech at transmission rates of 48Kbps, 56Kbps and 64Kbps.

Class maps have been configured at LER and Customer edge devices to match traffic and apply policy giving priority to voice packets. For demonstration of the voice traffic taking priority over default best effort traffic, first I transfer data from one server to another while links are empty no call is in progress. We can see that this traffic takes all bandwidth and speed, no drops are observed

```
LER1#sh policy-map int s0/0
  Service-policy output: Project

    Class-map: RTP (match-all)
      0 packets, 0 bytes
      5 minute offered rate 0 bps, drop rate 0 bps
      Match: ip rtp 16384 16383
      Queueing
        Strict Priority
        Output Queue: Conversation 40
        Bandwidth 56 (kbps) Burst 1400 (Bytes)
        (pkts matched/bytes matched) 0/0 (total
        drops/bytes drops) 0/0

    Class-map: FTP (match-all)
      6889 packets, 10317445 bytes
      5 minute offered rate 64000 bps, drop rate 0 bps
      Match: protocol ftp
      Queueing
        Output Queue: Conversation 43
        Bandwidth 20 (%)
        Bandwidth 25 (kbps) Max Threshold 64 (packets)
        (pkts matched/bytes matched) 93/138096
        (depth/total drops/no-buffer drops) 0/0/0
```

Then I make two phone calls between sites while data transfer is in progress. We can see that, data bandwidth and data rate of best effort data transfer has been reduced significantly and drops are observed in data transfer traffic after we place call between sites.

```
LER1#sh policy-map int s0/0
  Service-policy output: Project


      Class-map: RTP (match-all)
      37343 packets, 2389952 bytes
      5 minute offered rate 50000 bps, drop rate 0 bps
      Match: ip rtp 16384 16383
      Queueing
        Strict Priority
```

Output Queue: Conversation 40
Bandwidth 56 (kbps) Burst 1400 (Bytes)
(pkts matched/bytes matched) 37311/2387904
(total drops/bytes drops) 0/0

Class-map: FTP (match-all)
10656 packets, 15975009 bytes
5 minute offered rate 11000 bps, drop rate 15000 bps
Match: protocol ftp
Queueing
Output Queue: Conversation 43
Bandwidth 20 (%)
Bandwidth 25 (kbps) Max Threshold 64 (packets)
(pkts matched/bytes matched)
2287/3433772 (depth/total drops/no-
buffer drops) 14/0/0

We observe significant delay in data transfer at the servers and other site of the domain after phone calls in progress.

Link Failure

After link failure at 2s of simulation time first InterfaceTable module detects the failure and initializes, adds new connected routes (removed) to forwarding table. LinkStateRouting module sends triggered topology change messages advertising link failure and nodes rebuild their routing table. Link-state announce messages are sent at 2,018ms of simulation time, after 0.018ms, when link between LSR1 and LSR2 fails. After 132ms at 2.15ms of simulation time network converges and the change is being distributed to all network nodes. At this time network nodes are aware of the link failure. Regardless the code modification have been done in several modules (InterfaceTable, RoutingTable, LinkStateRouting) and awareness of nodes of failure, they still continue send the traffic throw primary link.

During the convergence time, (132ms) 8.5kb (kilo bit) traffic is initiated per voice host, 15.44kb traffic is generated by video host and 39.6kb data generated per data host. It makes approximately 170 kb traffic for all voice, 277 kb for video and 720 for data hosts, that has been dropped until IGP is supposed to recover and select alternate route.

CONCLUSION

The combined use of MPLS and DifferServ QoS is demonstrated to provide end to end guaranteed quality of service for multi traffic in MPLS networks in this paper. This behavioral analyze begins with the initial major problems on implementing MPLS DifferServ, packet format of MPLS shim header and two solution offered by IETF. After theoretical background about the MPLS and QoS and issues, analyze followed by presenting an approach in OMNET++ version 4.3 to estimate the capabilities and results of QoS over MPLS domain implementation. The analysis is made by focusing on the end-to-end delay, queue time and length variation, load and drop statistics.

During the project, we also have studied voice and video encoders and compression techniques used for them, their packet format and message length of each codec. Per hob behaviour models and traffic classification modules of OMNET++ modified to fit the project needs. Routing algorithm for MPLS nodes has been studied and few minor changes are done in built-in INET modules.

Parallel to simulation, similar diagram has been implemented in laboratory environment using real multi-vendor network devices (Cisco, Alcatel). Bandwidth variation between voice and data traffic has been measured in laboratory and compared results overlaps with the outcomes measured in simulation.

Finally, we have concluded on the basis of considered network topology, configurations and simulated results that, VoIP over QoS enabled MPLS domain provides 80 %better throughput, 66% less delay and 95 % less buffering than default traffic and video, while video traffic gets second priority over data flow. The variation between classes is especially significant in MPLS network because of its label switching technology. Delay in voice and video traffic transmission occurs as result of the queue becomes full with the same priority packets and delay becomes inevitable in this case. Drop is observed only in best effort traffic is *5* is assigned as weight to video traffic, only negligible drop (0.003%) detected when weight is *4*. I suggest using different path for high critical-mission traffic and demanding customers or reserve amount of bandwidth for them to avoid this case in future studies and industry implementation.

FUTURE WORK

Simulation of link failure and recovery while simulation is in progress and delay variance measurement during convergence time initially was part of part of the project. Simulation runs using primary link for transmission between two cities until primary link between LSR1 and LSR3 nodes fails. Interior Gateway Protocol – LinkStateRouting in our simulation should detect the failure and forward traffic using alternate path LER1←→LSR2←→LSR4←→LER2. Regardless of the several attempts (behaviour changes of modules, changing routing table manually using ScenarioManager, initializing forwarding and routing table) alternate path is not selected and nodes still try to use the same path for transmission. Using IS-IS routing protocol with fast re-routing feature implementation could accomplish this task, as this protocol extensively is being used in providers backbone networks. Fast-rerouting technique in IS-IS algorithm calculates alternate path before failure occurs and keeps this loop-free alternate path in each node's local topology. There is no need for link state recalculation when failure occurs. Routing protocol immediately shifts to alternate path after failure detection.IS-IS routing protocol has not been implemented in OMNET++ and INET framework, instead LinkStateRouting protocol has been designed in INET especially for MPLS networks as it has minimal overhead and fast convergence time. Developing LinkStateRouting (used in our simulation) to IS-IS protocol and adding fast-reroute technique is a better solution for performance developing and failure recovery. Replacing currently used routing protocol with IS-IS, will decrease end-to-end delay, queue time and length measurements in all traffic forwarding classes and this change can be reflected in voice traffic. Future work on this task in this simulation can bring cause better outcomes.

R E F E R E N C E S

RFC 5462: Multiprotocol Label Switching (MPLS) Label Stack Entry

RFC 3270: Multi-Protocol Label Switching (MPLS) Support of Differentiated Services

RFC 5777: Traffic Classification and Quality of Service

RFC 2475: An Architecture for Differentiated Services", December 1998

RFC 3290: An Informal Management Model for Diffserv Routers

OMNET++ User Manual Version 4.2.2.2

INET Framework for OMNET++ manual

OMNET++ User IDE Guide

Thomas Chamberlain: "Learning OMNET++"

Abdul-Rahman Elshafei: "Introduction to Omnet++ and its Services"

Yao Wang: Video Coding Standards

SANS Institute InfoSec Reading Room: Latency and QoS for Voice over IP

Johan Martin Olof Petersson: Master thesis-"MPLS Based Recovery Mechanisms"

Md. Tariq Aziz Mohammad Saiful Islam: "Performance Evalution of Real-Time Application over DiffServ/MPLS in IPv4/IPv6 Networks"

Anu Rathi, Amirta Ticku, Niti Gupta: "Development Of Test-Bed For Performance Evolution Multiprotocol Label Switching"

Amardeep Singh, Gagan Mittal: "QoS and Traffic Engineering: MPLS, DifferServ and Constraint Based Routing"

Shahid Ali Bilal Zahid Rana: "OPNET Analysis of VoIP over MPLS VPN with IP QoS"

Xavier Lleixa Rillo: "Analysis of MPLS-TE with OPNET"

Marcelino Minero-Muniz, Vicente Alarcon-Aquino: "Performance Evaluation of MPLS Path Restoration Schemes using OMNET++"

Paresh Shah, Utpal Mukhopadhyaya, Arun Sathiamurthi: "Overview of QoS in Packet-based IP and MPLS Networks"

Mina Amin, Kin-Hon Ho, George Pavlou: "MPLS QoS aware Traffic Engineering for Network Resilience"

www.cisco.com, www.omnetpp.org, www.timbercon.com, www.startneworks.info/2010/08, www.hindawi.com/journals/ijdmb/2010/836501, www.ng-ethernet.com, www.wikipedia.org

APPENDIX

SIMULATION BACKGROUND:

OMNET++ is a discrete event simulation tool designed to simulate computer networks, multi-processors and other distributed systems. Its applications can be extended for modelling other systems as well. OMNET++ has become a popular network simulation tool in the scientific community as well as in industry over the years.

OMNET++ itself is not a simulator of anything concrete, but rather provides infrastructure and tools for writing simulations. One of the fundamental ingredients of this infrastructure is component architecture for simulation models. OMNET++ model consists of hierarchical nested structure and modules at the lowest hierarchy are called simple modules which act as a part of compound modules. The top level module is the system module or it is called network. Models are assembled from reusable components termed modules. Each module has its open source code and message format to customize module behaviour or to parameterize the topology. Modules communicate throw message passing, where message may carry arbitrary data structures. They are interconnected with each-other using gates. The communication among modules is accomplished by different messages.

Simple modules are programmed in C++ language, and make use of simulation library as *omnetpp.h* has been included in OMNET++. Each simple module is associated **.cc* and **.h* extended files which defines the behaviour of module. Because of its generic architecture, OMNET++ has been used in following various problem domains:

- Modeling of wired and wireless communication networks
- Protocol modeling
- Modeling if queuing networks
- Validating of hardware architectures
- Modelling of multiprocessors and other distributed hardware systems
- Evaluating performance aspects of software's
- Simulation of any system where discrete event approach is suitable.

OMNET++ supports Graphical User Interface (GUI) to develop simulations of different communication networks, save time spent on code writing and visualization. We can test and debug the simulation with a powerful graphical user interface, and finally run it with a simple and fast user interface that supports batch execution. It also allows the user to initiate and terminate simulations, as well as change variable inside simulation models. These features are handy during the development and debugging phase of modules in a project.

The structure of the simulation model is described in network description file in OMNET++. Network description (NED) has its language network editor and lets the user declare simple modules, connect and assemble them easily in compound modules. The language defines several types of modules, the parameters associated and the interconnection between them while simultaneously graphical editor language (GNED) shows the result of the built network. The files containing the network descriptions should end with a *.ned* suffix. The NEDC compiler translates the network descriptions into C++ code. Then, it is compiled by the C++ compiler and linked into executable simulation

The environment for simulations Tkenv shows the results of the NED and C++ programming and allows a complete control of the simulation programmed. Tkenv is a portable graphical windowing user interface. Tracing, debugging, and simulation execution is supported by Tkenv. It has the ability to provide a detailed picture of the state of the simulation at any point during the execution. This feature makes Tkenv a good candidate in the development stage of a simulation or for presentations. This ambient show a timeline for the messages created and as a consequence shows the occurrence of events on the simulation.

OMNET++ also supports parallel distributed simulation by using several mechanisms for communication between partitions of a parallel distributed simulation. Models do not need any special instrumentation to be run in parallel, it can be easily done by configuration. Each network should have *.ini* extended configuration file where different parameters can be assigned to modules, parallel simulation can be configured. Parameters for modules can be configured from simulation **.ned* file itself or using *.ini* configuration file which configuration file takes priority. Different scenarios can be applied to simulation while simulation is in progress by using ScenarioManager or LifecycleController modules.

OMNET++ provides built-in support for result recording and analyse via output vectors, logs, scalars and histogram files. All these variable values modified by module event on a simulation are stored and plotted when the finish function is called. The changes in vector variables are plotted using graphical tool plove and the change on scalar variables are plotted using the tool scalars. Analyse configuration should be done in configurator file (*ini*). Output vectors are time series data, recorded from simple modules or channels. It is used to record end-to-end delay, round trip times, queue length and time, link utilization etc. Output scalars are summary results, computed during the simulation and written out when the simulation completes. Result can be integer number or statistical summary comprised of several fields such as count, mean, standard deviation, sum etc. There are two ways of collecting results:

- Based on the signal mechanism, using declared statistics.
- Directly from C++ code, using the simulation library.

The first method which is new feature for OMNET++ will be used for data analyse in this simulation.

CONFIGURATION OF *CAPSTONE.NED* FILE:

```
package inet.examples.CAPSTONE;
import inet.base.UnimplementedModule;
import inet.linklayer.ethernet.EtherHub;
import inet.mobility.models.ANSimMobility;
import inet.networklayer.autorouting.ipv4.IPv4NetworkConfigurator;
import inet.base.LifecycleController;
import inet.base.UnimplementedModule;
import inet.examples.bgpv4.BGPUpdate.BGPRouter;
import inet.networklayer.ipv4.NetworkInfo;
import inet.nodes.ethernet.Eth100G;
import inet.nodes.ethernet.EtherSwitch;
import inet.nodes.inet.NodeBase;
import inet.nodes.inet.Router;
import inet.nodes.inet.StandardHost;
import inet.nodes.inet.Sw;
import inet.nodes.inet.VideoHost;
import inet.nodes.inet.VoiceHost;
import inet.world.scenario.ScenarioManager;
import ned.DatarateChannel;
import inet.nodes.inet.LER;
import inet.nodes.inet.LSR;
import inet.linklayer.IWiredNic;
import inet.networklayer.ipv4.RoutingTable;
import inet.networklayer.ospfv2.OSPFRouting;
import inet.nodes.inet.NetworkLayer;

network MplsQoS
{
  parameters:
    @display("bgb=1198,595");
  types:
    channel H1 extends ned.DatarateChannel
    {
      datarate = 2Mbps;
      delay = 1ms;
    }
    channel E1 extends ned.DatarateChannel
    {
      datarate = 2Mbps;
      delay = 0.5ms;
    }
    channel C1 extends ned.DatarateChannel
    {
      datarate = 5Mbps;
      delay = 2ms;
    }
    channel D1 extends ned.DatarateChannel
    {
      datarate = 5Mbps;
      delay = 1.5ms;
    }

  submodules:
    LER1: LER {
      parameters:
        peers = "ethg0 ethg1 eth0";
```

```
      @display("p=406,231;is=l;i=abstract/opticalrouter");
}
LSR1: LSR {
   peers = "eth0 eth1 eth2";
   @display("p=497,119;is=l");
}
LSR2: LSR {
   peers = "eth0 eth1 eth2";
   @display("p=496,359;is=l");
}
LSR3: LSR {
   peers = "eth0 eth1 eth2";
   @display("p=686,122;is=l");
}
LSR4: LSR {
   peers = "eth0 eth1 eth2";
   @display("p=680,360;is=l");
}
LER2: LER {
   peers = "ethg0 ethg1 eth0";
   @display("p=759,234;is=l;i=abstract/opticalrouter");

}
lifecycleController: LifecycleController {
   parameters:
      @display("p=656,42");
}

configurator: IPv4NetworkConfigurator {
   parameters:
      @display("p=512,43");
}
Data3: StandardHost {
   @display("p=238,399");
}
Data31: StandardHost {
   @display("p=1129,460");
}
Voice1: StandardHost {
   @display("p=174,111");
}
Video2: StandardHost {
   @display("p=155,232");
}
Voice11: StandardHost {
   @display("p=1002,136");
}
Video21: StandardHost {
   @display("p=998,231");
}
SwE: Sw {
   peers = "eth0";
   @display("p=269,229");
}
//}
SwC: Sw {
   peers = "eth0";
   @display("p=880,236");
```

```
}
TData4: StandardHost {
   @display("p=80,492");
}
TData41: StandardHost {
   @display("p=1017,543");
}
TData5: StandardHost {
   @display("p=246,472");
}
TData51: StandardHost {
   @display("p=1018,441");
}
TData6: StandardHost {
   @display("p=95,516");
}
TData61: StandardHost {
   @display("p=1080,548");
}
TData1: StandardHost {
   @display("p=146,545");
}
TData11: StandardHost {
   @display("p=1158,531");
}
TData2: StandardHost {
   @display("p=199,480");
}
TData21: StandardHost {
   @display("p=1084,455");
}
TVideo11: StandardHost {
   @display("p=1063,281");
}
TVideo1: StandardHost {
   @display("p=36,360");
}
TVideo3: StandardHost {
   @display("p=52,306");
}
TVideo31: StandardHost {
   @display("p=1131,259");
}
TVideo4: StandardHost {
   @display("p=100,288");
}
TVideo41: StandardHost {
   @display("p=1159,355");
}
TVideo5: StandardHost {
   @display("p=67,229");
}
TVideo51: StandardHost {
   @display("p=1084,222");
}
TVideo6: StandardHost {
   @display("p=72,349");
}
```

```
TVideo61: StandardHost {
   @display("p=1151,222");
}
TVideo7: StandardHost {
   @display("p=126,319");
}
TVideo71: StandardHost {
   @display("p=1170,261");
}
TVoice2: StandardHost {
   @display("p=102,91");
}
TVoice3: StandardHost {
   @display("p=263,57");
}
TVoice4: StandardHost {
   @display("p=180,27");
}
TVoice5: StandardHost {
   @display("p=56,52;is=s");
}
TVoice6: StandardHost {
   @display("p=218,29");
}
TVoice7: StandardHost {
   @display("p=94,124");
}
TVoice8: StandardHost {
   @display("p=250,132");
}
TVoice9: StandardHost {
   @display("p=99,31");
}
TVoice21: StandardHost {
   @display("p=1167,72");
}
TVoice31: StandardHost {
   @display("p=931,109");
}
TVoice41: StandardHost {
   @display("p=995,60");
}
TVoice51: StandardHost {
   @display("p=1069,134");
}
TVoice61: StandardHost {
   @display("p=1139,115");
}
TVoice71: StandardHost {
   @display("p=1079,69");
}
TVoice81: StandardHost {
   @display("p=1137,37");
}
TVoice91: StandardHost {
   @display("p=946,28");
}
TVideo8: StandardHost {
```

```
        @display("p=34,271");
    }
    TVideo81: StandardHost {
        @display("p=1157,320");
    }
    TData7: StandardHost {
        @display("p=134,470");
    }
    TData71: StandardHost {
        @display("p=1110,507");
    }
    TData8: StandardHost {
        @display("p=1036,489");
    }
    TData81: StandardHost {
        @display("p=219,546");
    }
    TVoice01: StandardHost {
        @display("p=1055,34");
    }
    TVoice0: StandardHost {
        @display("p=36,95");
    }
    scenarioManager: ScenarioManager {
        parameters:
            @display("p=593,451");
    }
    Multi1: StandardHost {
        @display("p=334,526");
    }
    Multi2: StandardHost {
        @display("p=914,521");
    }
    MVoice1: StandardHost {
        @display("p=40,159");
    }
    MVoice2: StandardHost {
        @display("p=1110,157");
    }
    MVideo1: StandardHost {
        @display("p=36,419");
    }
    MVideo2: StandardHost {
        @display("p=1089,346");
    }
    MData1: StandardHost {
        @display("p=41,557");
    }
    MData2: StandardHost {
        @display("p=1169,450");
    }
  connections:
LER1.eth++ <--> E1 { @display("ls=#FF8000,2;t=E1"); } <--> LSR1.eth++;
LER1.eth++ <--> E1 { @display("ls=#FF8000,2;t=E1"); } <--> LSR2.eth++;
LSR1.eth++ <--> E1 { @display("ls=#FF8000,2;t=E1"); } <--> LSR2.eth++;
LSR1.eth++ <--> D1 { @display("ls=black,5;t=D1"); } <--> LSR3.eth++;
LSR2.eth++ <--> C1 { @display("t=C1;ls=#FF0080,3"); } <--> LSR4.eth++;
LSR3.eth++ <--> E1 { @display("t=E1;ls=#FF8000,2"); } <--> LSR4.eth++;
```

```
LSR3.eth++ <--> E1 { @display("t=E1;ls=#FF8000,2"); } <--> LER2.eth++;
LSR4.eth++ <--> E1 { @display("t=E1;ls=#FF8000,2"); } <--> LER2.eth++;
    SwC.ethg++ <--> H1 { @display("t=H1;ls=blue"); } <--> LER2.ethg++;
    SwE.ethg++ <--> H1 { @display("t=H1,t;ls=blue"); } <--> LER1.ethg++;
    //HOSTS
    Voice1.ethg++ <--> Eth100G <--> SwE.ethg++;
    Voice11.ethg++ <--> Eth100G <--> SwC.ethg++;
    Data3.ethg++ <--> Eth100G <--> SwE.ethg++;
    SwC.ethg++ <--> Eth100G <--> Data31.ethg++;
    Video2.ethg++ <--> Eth100G <--> SwE.ethg++;
    SwC.ethg++ <--> Eth100G <--> Video21.ethg++;
    SwC.ethg++ <--> Eth100G <--> TData61.ethg++;
    SwC.ethg++ <--> Eth100G <--> TData51.ethg++;
    SwC.ethg++ <--> Eth100G <--> TData21.ethg++;
    SwC.ethg++ <--> Eth100G <--> TData11.ethg++;
    SwE.ethg++ <--> Eth100G <--> TData4.ethg++;
    SwE.ethg++ <--> Eth100G <--> TData5.ethg++;
    SwE.ethg++ <--> Eth100G <--> TData1.ethg++;
    SwE.ethg++ <--> Eth100G <--> TData6.ethg++;
    SwE.ethg++ <--> Eth100G <--> TData2.ethg++;
    SwE.ethg++ <--> Eth100G <--> TVideo5.ethg++;
    SwE.ethg++ <--> Eth100G <--> TVideo1.ethg++;
    SwE.ethg++ <--> Eth100G <--> TVideo4.ethg++;
    SwE.ethg++ <--> Eth100G <--> TVideo3.ethg++;
    SwC.ethg++ <--> Eth100G <--> TVideo51.ethg++;
    SwC.ethg++ <--> Eth100G <--> TVideo41.ethg++;
    SwC.ethg++ <--> Eth100G <--> TVideo11.ethg++;
    SwC.ethg++ <--> Eth100G <--> TVideo31.ethg++;
    SwE.ethg++ <--> Eth100G <--> TVideo6.ethg++;
    SwE.ethg++ <--> Eth100G <--> TVideo7.ethg++;
    SwC.ethg++ <--> Eth100G <--> TVideo71.ethg++;
    SwC.ethg++ <--> Eth100G <--> TVideo61.ethg++;
    SwE.ethg++ <--> Eth100G <--> TVoice8.ethg++;
    SwE.ethg++ <--> Eth100G <--> TVoice2.ethg++;
    SwE.ethg++ <--> Eth100G <--> TVoice7.ethg++;
    SwE.ethg++ <--> Eth100G <--> TVoice5.ethg++;
    SwE.ethg++ <--> Eth100G <--> TVoice9.ethg++;
    SwE.ethg++ <--> Eth100G <--> TVoice4.ethg++;
    SwE.ethg++ <--> Eth100G <--> TVoice3.ethg++;
    SwE.ethg++ <--> Eth100G <--> TVoice6.ethg++;
    SwC.ethg++ <--> Eth100G <--> TVoice21.ethg++;
    SwC.ethg++ <--> Eth100G <--> TVoice31.ethg++;
    SwC.ethg++ <--> Eth100G <--> TVoice51.ethg++;
    SwC.ethg++ <--> Eth100G <--> TVoice61.ethg++;
    SwC.ethg++ <--> Eth100G <--> TVoice71.ethg++;
    SwC.ethg++ <--> Eth100G <--> TVoice41.ethg++;
    SwC.ethg++ <--> Eth100G <--> TVoice91.ethg++;
    SwC.ethg++ <--> Eth100G <--> TVoice81.ethg++;
    SwC.ethg++ <--> Eth100G <--> TVideo81.ethg++;
    SwE.ethg++ <--> Eth100G <--> TVideo8.ethg++;
    SwE.ethg++ <--> Eth100G <--> TData7.ethg++;
    SwE.ethg++ <--> Eth100G <--> TData81.ethg++;
    SwC.ethg++ <--> Eth100G <--> TData71.ethg++;
    SwC.ethg++ <--> Eth100G <--> TData8.ethg++;
    SwE.ethg++ <--> Eth100G <--> TVoice0.ethg++;
    SwC.ethg++ <--> Eth100G <--> TVoice01.ethg++;
    SwC.ethg++ <--> Eth100G <--> TData41.ethg++;
    SwE.ethg++ <--> Eth100G <--> MVoice1.ethg++;
```

```
    SwE.ethg++ <--> Eth100G <--> MVideo1.ethg++;
    SwE.ethg++ <--> Eth100G <--> MData1.ethg++;
    SwE.ethg++ <--> Eth100G <--> Multi1.ethg++;
    SwC.ethg++ <--> Eth100G <--> MVoice2.ethg++;
    SwC.ethg++ <--> Eth100G <--> MVideo2.ethg++;
    SwC.ethg++ <--> Eth100G <--> MData2.ethg++;
    SwC.ethg++ <--> Eth100G <--> Multi2.ethg++;
}
```

CONFIGURATION OF *OMNET.INI* FILE:

```
[General]
network = inet.examples.CAPSTONE.MplsQoS
parallel-simulation = false
sim-time-limit = 300s
**.partition-id =
**.result-recording-modes = all
**.scalar-recording = true
**.vector-recording = true
debug-statistics-recording = true
tkenv-plugin-path = ../../../etc/plugins
record-eventlog = true
# bgp settings
**.bgpConfig = xmldoc("BGPConfig.xml")
**.bgp*.dataTransferMode = "object"
#statistics
**.T*.udpApp[*].sentPk.result-recording-modes = count
**.T*.udpApp[*].rcvdPk.result-recording-modes = count
**.D*.udpApp[*].sentPk.result-recording-modes = count
**.D*.udpApp[*].rcvdPk.result-recording-modes = count
**.V*.udpApp[*].sentPk.result-recording-modes = count
**.V*.udpApp[*].rcvdPk.result-recording-modes = count

**.TVoice*.udpApp[*].endToEndDelay.result-recording-modes = vector, scalar, count
**.TVideo*.udpApp[*].endToEndDelay.result-recording-modes = vector
**.TVideo*.*.endToEndDelay.result-recording-modes = vector
**.TData*.udpApp[*].endToEndDelay.result-recording-modes = vector,scalar,count
**.Video*.udpApp[*].endToEndDelay.result-recording-modes = vector
**.Data*.udpApp[*].endToEndDelay.result-recording-modes = vector,scalar,count
**.Voice*.udpApp[*].endToEndDelay.result-recording-modes = vector
**.LER1*.eth[0].endToEndDelay.result-recording-modes = vector
**.LER1*.eth[0].endToEndDelay.result-recording-modes = vector
**.LER1.eth[0].**Queue.rcvdPk.result-recording-modes = count
**.LER1.eth[0].**Queue.rcvdPk.result-recording-modes = count
**.Sw*.eth[3].**Queue.rcvdPk.result-recording-modes = count
**.LSR*.eth[*].**Queue.rcvdPk.result-recording-modes = count
**.LSR1.eth[0].**Queue.rcvdPk.result-recording-modes = count

**.LER1.eth[*].**Queue.dropPk.result-recording-modes = count
**.Sw*.eth[3].**Queue.dropPk.result-recording-modes = count
**.LSR*.eth[*].**Queue.dropPk.result-recording-modes = count
**.LSR1.eth[0].**Queue.dropPk.result-recording-modes = count
**.LER1.eth[*].**Queue.queueLength.result-recording-modes = timeavg
**.Sw*.eth[3].**Queue.queueLength.result-recording-modes = timeavg
**.LSR*.eth[*].**Queue.queueLength.result-recording-modes = timeavg
**.LSR1.eth[0].**Queue.queueLength.result-recording-modes = timeavg
**.LER1.eth[*].**Queue.queueingTime.result-recording-modes = vector
**.Sw*.eth[3].Queue.queueingTime.result-recording-modes = vector
**.LSR*.eth[*].Queue.queueingTime.result-recording-modes = vector
```

```
**.udpApp[*].sentPk*.scalar-recording = true
**.udpApp[*].rcvdPk*.scalar-recording = true
**.udpApp[*].endToEndDelay*.scalar-recording = true
**.LER1.eth[0].**Queue.*.scalar-recording = true
**.LER2.eth[*].**Queue.*.scalar-recording = true
**.LSR*.eth[*].**Queue.*.scalar-recording = true

**.afQueue.*.scalar-recording = true
**.efQueue.*.scalar-recording = true
**.beQueue.*.scalar-recording = true
**.af11Queue.*.scalar-recording = true
**.endToEndDelay.result-recording-modes = vector
**.eth[*].queue.afQueue.queueLength.result-recording-modes = timeavg,vector
**.eth[*].queue.efQueue.queueLength.result-recording-modes = timeavg,vector
**.eth[*].queue.beQueue.queueLength.result-recording-modes = timeavg,vector


#MULTI
**.Multi*.numUdpApps = 3
**.Multi*.udpApp[*].typename = "UDPBasicBurst"
**.Multi*.udpApp[0].localPort = 2000
**.Multi*.udpApp[1].localPort = 1000
**.Multi*.udpApp[2].localPort = 2020
**.Multi*.udpApp[*].destPort = 2020
**.Multi*.udpApp[*].startTime = 20ms
**.Multi*.udpApp[*].destAddresses = "Multi2"
**.M*.udpApp[*].chooseDestAddrMode = "once"
**.Multi*.udpApp[0].messageLength = 172B
**.Multi*.udpApp[1].messageLength = 900B
**.Multi*.udpApp[2].messageLength = 1400B

**.MVoice*.numUdpApps = 1
**.MVideo*.numUdpApps = 1
**.MData*.numUdpApps = 1
**.MVoice*.udpApp[*].typename = "UDPBasicBurst"
**.MVideo*.udpApp[*].typename = "UDPBasicBurst"
**.MData*.udpApp[*].typename = "UDPBasicBurst"
**.MVoice*.udpApp[*].localPort = 2000
**.MVoice*.udpApp[*].destPort = 2000
**.MVoice*.udpApp[*].messageLength = 172B
**.MVoice1.udpApp[*].destAddresses = "Multi2"
**.MVoice2.udpApp[*].destAddresses = "Multi1"

**.M*.udpApp[*].burstDuration = exponential(0.380s)
**.M*.udpApp[*].sleepDuration = exponential(0.240s)
**.M*.udpApp[*].sendInterval = 20ms
**.MVoice1.udpApp[*].startTime = 25ms
**.MVoice2.udpApp[*].startTime = 25ms

**.MVideo*.udpApp[*].localPort = 1000
**.MVideo*.udpApp[*].destPort = 1000
**.MVideo*.udpApp[*].messageLength = 900B
**.MVideo1.udpApp[*].destAddresses = "Multi2"
**.MVideo2.udpApp[*].destAddresses = "Multi1"
**.MVideo1.udpApp[*].startTime = 25ms
**.MVideo2.udpApp[*].startTime = 25ms

**.MData*.udpApp[*].localPort = 2020
**.MData*.udpApp[*].destPort = 2020
**.MData*.udpApp[*].messageLength = 1400B
**.MData1.udpApp[*].destAddresses = "Multi2"
**.MData2.udpApp[*].destAddresses = "Multi1"
**.MData1.udpApp[*].startTime = 25ms
```

```
**.MData2.udpApp[*].startTime = 25ms

# voice streaming
**.Voice1*.numUdpApps = 1
**.Voice1*.udpApp[*].typename = "UDPBasicBurst"
**.Voice1*.udpApp[*].localPort = 2000
**.Voice1*.udpApp[*].destPort = 2000
**.Voice1*.udpApp[*].messageLength = 172B
**.Voice11.udpApp[*].destAddresses = "Voice1"
**.Voice1.udpApp[*].destAddresses = "Voice11"
**.Voice1*.udpApp[*].chooseDestAddrMode = "once"
**.Voice11.udpApp[*].startTime = uniform(1s,2s)
**.Voice11.udpApp[*].stopTime = 300s
**.Voice11.udpApp[0].burstDuration = exponential(0.352s)
**.Voice11.udpApp[0].sleepDuration = exponential(0.450s)
**.Voice11.udpApp[0].sendInterval = 150ms

**.TD*.udpApp[*].chooseDestAddrMode = "once"
**.Video*.udpApp[*].chooseDestAddrMode = "once"
**.TVideo*.udpApp[*].chooseDestAddrMode = "once"
**.Data*.udpApp[*].chooseDestAddrMode = "once"

**.Data*.udpApp[*].burstDuration = exponential(0.352s)
**.Video*.udpApp[*].burstDuration = exponential(0.352s)
**.TVideo*.udpApp[*].burstDuration = exponential(0.252s)
**.TData*.udpApp[*].burstDuration = exponential(0.352s)
**.TVoice*.udpApp[*].burstDuration = exponential(0.380s)

**.Data*.udpApp[*].sleepDuration = exponential(0.450s)
**.Video*.udpApp[*].sleepDuration = exponential(0.252s)
**.TVideo*.udpApp[*].sleepDuration = exponential(0.496s)
**.TData*.udpApp[*].sleepDuration = exponential(0.400s)
**.TVoice*.udpApp[*].sleepDuration = exponential(0.240s)

**.Voice1.udpApp[*].startTime = uniform(800ms,900ms)
**.Voice1.udpApp[*].stopTime = 300s
**.Voice1.udpApp[*].burstDuration = exponential(0.322s)
**.Voice1.udpApp[*].sleepDuration = exponential(0.390s)
**.Voice1.udpApp[*].sendInterval = 450ms

**.TVoice*.numUdpApps = 1
**.TVoice*.udpApp[*].chooseDestAddrMode = "once"
**.TVoice*.udpApp[*].typename = "UDPBasicBurst"
**.TVoice*.udpApp[*].localPort = 2000
**.TVoice*.udpApp[*].destPort = 2000
**.TVoice*.udpApp[*].messageLength = 172B

**.TVoice01.udpApp[*].destAddresses = "TVoice0"
**.TVoice0.udpApp[*].destAddresses = "TVoice01"
**.TVoice2.udpApp[*].destAddresses = "TVoice21"
**.TVoice21.udpApp[*].destAddresses = "TVoice2"
**.TVoice31.udpApp[*].destAddresses = "TVoice3"
**.TVoice3.udpApp[*].destAddresses = "TVoice31"
**.TVoice41.udpApp[*].destAddresses = "TVoice4"
**.TVoice4.udpApp[*].destAddresses = "TVoice41"
**.TVoice51.udpApp[*].destAddresses = "TVoice5"
**.TVoice5.udpApp[*].destAddresses = "TVoice51"
**.TVoice61.udpApp[*].destAddresses = "TVoice6"
**.TVoice6.udpApp[*].destAddresses = "TVoice61"
**.TVoice71.udpApp[*].destAddresses = "TVoice7"
**.TVoice7.udpApp[*].destAddresses = "TVoice71"
**.TVoice81.udpApp[*].destAddresses = "TVoice8"
**.TVoice8.udpApp[*].destAddresses = "TVoice81"
```

```
**.TVoice91.udpApp[*].destAddresses = "TVoice9"
**.TVoice9.udpApp[*].destAddresses = "TVoice91"

**.TVoice01.udpApp[*].startTime = uniform(900ms,999ms)
**.TVoice01.udpApp[*].stopTime = 300s
**.TVoice01.udpApp[*].sendInterval = 200ms
**.TVoice0.udpApp[*].startTime = 25ms   #new
**.TVoice0.udpApp[*].sendInterval = 20ms

**.TVoice21.udpApp[*].sendInterval = 20ms
**.TVoice21.udpApp[*].startTime = 25ms
**.TVoice2.udpApp[*].sendInterval = 20ms
**.TVoice2.udpApp[*].startTime = 25ms

**.TVoice31.udpApp[*].sendInterval = 20ms
**.TVoice31.udpApp[*].startTime = 25ms
**.TVoice3.udpApp[*].sendInterval = 20ms
**.TVoice3.udpApp[*].startTime = 25ms

**.TVoice41.udpApp[*].sendInterval = 20ms
**.TVoice41.udpApp[*].startTime = 25ms
**.TVoice4.udpApp[*].sendInterval = 20ms
**.TVoice4.udpApp[*].startTime = 25ms

**.TVoice51.udpApp[*].sendInterval = 20ms
**.TVoice51.udpApp[*].startTime = 35ms
**.TVoice5.udpApp[*].sendInterval = 20ms
**.TVoice5.udpApp[*].startTime = 35ms

**.TVoice61.udpApp[*].sendInterval = 20ms
**.TVoice6.udpApp[*].sendInterval = 20ms
**.TVoice6.udpApp[*].startTime = 25ms
**.TVoice61.udpApp[*].startTime = 25ms


**.TVoice71.udpApp[*].sendInterval = 20ms
**.TVoice71.udpApp[*].startTime = 24ms
**.TVoice7.udpApp[*].sendInterval = 20ms
**.TVoice7.udpApp[*].startTime = 24ms


**.TVoice81.udpApp[*].sendInterval = 20ms
**.TVoice81.udpApp[*].startTime = 24ms
**.TVoice8.udpApp[*].sendInterval = 20ms
**.TVoice8.udpApp[*].startTime = 24ms


**.TVoice91.udpApp[*].sendInterval = 20ms
**.TVoice9.udpApp[*].sendInterval = 20ms
**.TVoice9.udpApp[*].startTime = 25ms
**.TVoice91.udpApp[*].startTime = 25ms


###########################################################################

# Video streaming
**.Video2*.numUdpApps = 1
**.Video2*.udpApp[*].typename = "UDPBasicBurst"
**.Video2*.udpApp[*].destPort = 1000
**.Video2*.udpApp[*].localPort = 1000
**.Video2.udpApp[*].startTime = uniform(1s,2s)
**.Video21.udpApp[*].startTime = uniform(650ms,800ms)
**.Video2.udpApp[*].stopTime = 320s
```

```
**.Video21.udpApp[*].stopTime = 550s
**.Video2.udpApp[*].sendInterval = 350ms
**.Video21.udpApp[*].sendInterval = 260ms
**.Video2*.udpApp[*].messageLength = 900B
**.Video2.udpApp[*].destAddresses = "Video21"
**.Video21.udpApp[*].destAddresses = "Video2"

**.TVideo*.numUdpApps = 1
**.TVideo*.udpApp[*].typename = "UDPBasicBurst"
**.TVideo*.udpApp[*].destPort = 1000
**.TVideo*.udpApp[*].localPort = 1000
**.TVideo*.udpApp[*].messageLength = 600B

**.TVideo1.udpApp[*].destAddresses = "TVideo11"
**.TVideo11.udpApp[*].destAddresses = "TVideo1"
**.TVideo3.udpApp[*].destAddresses = "TVideo31"
**.TVideo31.udpApp[*].destAddresses = "TVideo3"
**.TVideo4.udpApp[*].destAddresses = "TVideo41"
**.TVideo41.udpApp[*].destAddresses = "TVideo4"
**.TVideo5.udpApp[*].destAddresses = "TVideo51"
**.TVideo51.udpApp[*].destAddresses = "TVideo5"
**.TVideo6.udpApp[*].destAddresses = "TVideo61"
**.TVideo61.udpApp[*].destAddresses = "TVideo6"
**.TVideo7.udpApp[*].destAddresses = "TVideo71"
**.TVideo71.udpApp[*].destAddresses = "TVideo7"
**.TVideo8.udpApp[*].destAddresses = "TVideo81"
**.TVideo81.udpApp[*].destAddresses = "TVideo8"


**.TVideo1.udpApp[*].sendInterval = 20ms
**.TVideo1.udpApp[*].startTime = 25ms
**.TVideo11.udpApp[*].sendInterval = 20ms
**.TVideo11.udpApp[*].startTime = 30ms
**.TVideo3.udpApp[*].sendInterval = 40ms
**.TVideo3.udpApp[*].startTime = 25ms
**.TVideo31.udpApp[*].sendInterval = 20ms
**.TVideo31.udpApp[*].startTime = 25ms
**.TVideo4.udpApp[*].sendInterval = 20ms
**.TVideo4.udpApp[*].startTime = 25ms
**.TVideo41.udpApp[*].sendInterval = 20ms
**.TVideo41.udpApp[*].startTime = 30ms
**.TVideo5.udpApp[*].sendInterval = 20ms
**.TVideo5.udpApp[*].startTime = 25ms
**.TVideo51.udpApp[*].sendInterval = 20ms
**.TVideo51.udpApp[*].startTime = 25ms
**.TVideo6.udpApp[*].sendInterval = 20ms
**.TVideo6.udpApp[*].startTime = 25ms
**.TVideo61.udpApp[*].sendInterval = 20ms
**.TVideo61.udpApp[*].startTime = 25ms
**.TVideo7.udpApp[*].sendInterval = 25ms
**.TVideo7.udpApp[*].startTime = 30ms
**.TVideo71.udpApp[*].sendInterval = 20ms
**.TVideo71.udpApp[*].startTime = 25ms
**.TVideo8.udpApp[*].sendInterval = 20ms
**.TVideo81.udpApp[*].startTime = 30ms
**.TVideo8.udpApp[*].startTime = 30ms
**.TVideo81.udpApp[*].sendInterval = 20ms

#data transfering ###################################################################
**.Data3*.numUdpApps = 2
**.Data3*.udpApp[*].typename = "UDPBasicBurst"
**.Data3*.udpApp[0].destPort = 2020
**.Data3*.udpApp[1].destPort = 2021
```

```
**.Data3*.udpApp[0].localPort = 2020
**.Data3*.udpApp[1].localPort = 2021
**.Data3.udpApp[*].startTime = uniform(400ms,500ms)
**.Data3.udpApp[*].stopTime = 500s
**.Data3.udpApp[*].sendInterval = 400ms
**.Data3*.udpApp[0].messageLength = 1200B
**.Data3*.udpApp[1].messageLength = 1200B
**.Data3.udpApp[*].destAddresses = "Data31"
**.Data31.udpApp[*].destAddresses = "Data3"
**.Data31.udpApp[*].startTime = uniform(600ms,700ms)
**.Data31.udpApp[*].stopTime = 500s
**.Data31.udpApp[*].sendInterval = 350ms

**.TData*.numUdpApps = 2
**.TData*.udpApp[*].typename = "UDPBasicBurst"
**.TData*.udpApp[0].destPort = 2020
**.TData*.udpApp[1].destPort = 2021
**.TData*.udpApp[0].localPort = 2020
**.TData*.udpApp[1].localPort = 2021
**.TData*.udpApp[0].messageLength = 1200B
**.TData*.udpApp[1].messageLength = 1200B

**.TData1.udpApp[*].destAddresses = "TData11"
**.TData11.udpApp[*].destAddresses = "TData1"
**.TData2.udpApp[*].destAddresses = "TData21"
**.TData21.udpApp[*].destAddresses = "TData2"
**.TData4.udpApp[*].destAddresses = "TData41"
**.TData41.udpApp[*].destAddresses = "TData4"
**.TData51.udpApp[*].destAddresses = "TData5"
**.TData5.udpApp[*].destAddresses = "TData51"
**.TData6.udpApp[*].destAddresses = "TData61"
**.TData61.udpApp[*].destAddresses = "TData6"
**.TData7.udpApp[*].destAddresses = "TData71"
**.TData71.udpApp[*].destAddresses = "TData7"
**.TData8.udpApp[*].destAddresses = "TData81"
**.TData81.udpApp[*].destAddresses = "TData8"

**.TData11.udpApp[*].sendInterval = 20ms
**.TData1.udpApp[*].sendInterval = 20ms
**.TData11.udpApp[*].startTime = 25ms
**.TData1.udpApp[*].startTime = 25ms
**.TData21.udpApp[*].sendInterval = 10ms
**.TData21.udpApp[*].startTime = 25ms
**.TData2.udpApp[*].sendInterval = 20ms
**.TData2.udpApp[*].startTime = 35ms
**.TData41.udpApp[*].sendInterval = 20ms
**.TData41.udpApp[*].startTime = 30ms
**.TData4.udpApp[*].sendInterval = 10ms
**.TData4.udpApp[*].startTime = 25ms
**.TData51.udpApp[*].sendInterval = 30ms
**.TData51.udpApp[*].startTime = 45ms
**.TData5.udpApp[*].sendInterval = 20ms
**.TData5.udpApp[*].startTime = 25ms
**.TData61.udpApp[*].sendInterval = 10ms
**.TData61.udpApp[*].startTime = 40ms
**.TData6.udpApp[*].startTime = 25ms
**.TData6.udpApp[*].sendInterval = 20ms
**.TData71.udpApp[*].sendInterval = 20ms
**.TData7.udpApp[*].sendInterval = 20ms
**.TData71.udpApp[*].startTime = 25ms
**.TData7.udpApp[*].startTime = 25ms
**.TData81.udpApp[*].sendInterval = 25ms
**.TData8.udpApp[*].sendInterval = 10ms
```

```
**.TData81.udpApp[*].startTime = 20ms
**.TData8.udpApp[*].startTime = 25ms


# QoS
**.Sw?.eth[*].ingressTCType = "EntryMark"
**.LER?.eth[*].ingressTCType = "EntryMark"
**.LSR?.eth[*].ingressTCType = "EntryMark"
**.LER?.eth[*].egressTCType = "EntryMark"
**.Sw?.eth[*].egressTCType = "EntryMark"
**.LSR?.eth[*].egressTCType = "EntryMark"

**.wrr.weights = "5 0 0 0 1"

**.LER*.eth[*].queueType = "PHB"
**.LSR*.eth[*].queueType = "PHB"
**.Sw?.eth[*].queueType = "PHB"


**.LER?.eth[*].ingressTC.classifier.filters = xmldoc("filters.xml",
"//experiment[@id='1']")
**.Sw?.eth[*].ingressTC.classifier.filters = xmldoc("filters.xml",
"//experiment[@id='1']")
**.LSR?.eth[*].ingressTC.classifier.filters = xmldoc("filters.xml",
"//experiment[@id='1']")
**.LSR?.eth[*].egressTC.classifier.filters = xmldoc("filters.xml",
"//experiment[@id='1']")
**.LER?.eth[*].egressTC.classifier.filters = xmldoc("filters.xml",
"//experiment[@id='1']")
**.Sw?.eth[*].egressTC.classifier.filters = xmldoc("filters.xml",
"//experiment[@id='1']")#
**.ingressTC.numClasses = 4
**.egressTC.numClasses = 4
**.ingressTC.marker.dscps = "EF AF11 AF21 BE"
**.egressTC.marker.dscps = "EF AF11 AF21 BE"


# LDP, MPLS settings
**.L*R*.holdTime = 3s
**.L*R*.helloInterval = 2s

# tcp config
**.ldp.dataTransferMode = "object"

# scenario
*.*.hasStatus = true
**.scenarioManager.script = xmldoc("scenario.xml")
```