2003-03

An evaluation of best effort traffic management of server and agent based active network management (SAAM) architecture

Ayvat, Birol

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

AN EVALUATION OF BEST EFFORT TRAFFIC MANAGEMENT OF SERVER AND AGENT-BASED ACTIVE NETWORK MANAGEMENT (SAAM) ARCHITECTURE

by

Birol Ayvat

March 2003

Thesis Advisor:	Geoffrey Xie
Second Reader:	John Gibson

Approved for public release; distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 2003	Master's Thesis

4. TITLE AND SUBTITLE: Evaluation of Best Effort Traffic Management of Server and Agent Based Active Network Management (SAAM) Architecture

5. FUNDING NUMBERS

6. AUTHOR (S) AYVAT, Birol

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
DARPA and NASA

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
G417

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the U.S. Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE
Statement A

13. ABSTRACT ***(maximum 200 words)***

The Server and Agent-based Active Network Management (SAAM) architecture was initially designed to work with the next generation Internet where increasingly sophisticated applications will require QoS guarantees. Although such QoS traffic is growing in volume, Best Effort traffic, which does not require QoS guarantees, needs to be supported for foreseeable future. Thus, SAAM must handle Best Effort traffic as well as QoS traffic.

A Best Effort traffic management algorithm was developed for SAAM recently to take advantage of the abilities of the SAAM server. However, this algorithm has not been evaluated quantitatively.

This thesis conducts experiments to compare the performance of the Best Effort traffic management scheme of the SAAM architecture against the well known MPLS Adaptive Traffic Engineering (MATE) Algorithm. A couple of realistic network topologies were used. The results show while SAAM may not perform as well as MATE with a fixed set of paths, using SAAM's dynamic path deployment functionality allows the load to be distributed across more parts of the network, thus achieving better performance than MATE. Much of the effort was spent on implementing the MATE algorithm in SAAM. Some modifications were also made to the SAAM code based on the experimental results to increase the performance of SAAM's Best Effort solution.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Next Generation Internet, Quality of Service, Best Effort Traffic, Networks, Routing, Resource Management	105
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500 **Standard Form 298 (Rev. 2-89)**
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited.

AN EVALUATION OF BEST EFFORT TRAFFIC MANAGEMENT OF SERVER AND AGENT BASED ACTIVE NETWORK MANAGEMENT (SAAM) ARCHITECTURE

Birol Ayvat
Lieutenant Junior Grade, Turkish Navy
B.S., Turkish Naval Academy, 1997

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2003

Author: Birol Ayvat

Approved by: Geoffrey XIE
Thesis Advisor

John Gibson
Second Reader

Peter Denning, Chairman
Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The Server and Agent-based Active Network Management (SAAM) architecture was initially designed to work with the next generation Internet where increasingly sophisticated applications will require QoS guarantees. Although such QoS traffic is growing in volume, Best Effort traffic, which does not require QoS guarantees, needs to be supported for foreseeable future. Thus, SAAM must handle Best Effort traffic as well as QoS traffic.

A Best Effort traffic management algorithm was developed for SAAM recently to take advantage of the abilities of the SAAM server. However, this algorithm has not been evaluated quantitatively.

This thesis conducts experiments to compare the performance of the Best Effort traffic management scheme of the SAAM architecture against the well known MPLS Adaptive Traffic Engineering (MATE) Algorithm. A couple of realistic network topologies were used. The results show while SAAM may not perform as well as MATE with a fixed set of paths, using SAAM's dynamic path deployment functionality allows the load to be distributed across more parts of the network, thus achieving better performance than MATE. Much of the effort was spent on implementing the MATE algorithm in SAAM. Some modifications were also made to the SAAM code based on the experimental results to increase the performance of SAAM's Best Effort solution.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGEMENTS

I would like to acknowledge Prof. Geoffrey Xie for his continuous support throughout all phases of this thesis. I would like to thank my second reader Prof. John Gibson for his valuable input to this thesis.

I am also grateful to my wife, who supported me by her love and patience throughout my study.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. SERVER AND AGENT BASED ACTIVE NETWORK MANAGEMENT

1. What is SAAM?

SAAM is a network management system that enables a network to provide Quality of Service (QoS) in addition to the traditional Best Effort Service (BES) . Instead of a distributed router-based architecture, as the current Internet uses, SAAM uses a server-based hierarchical routing architecture.

2. SAAM Architecture

The SAAM architecture consists of a SAAM server that controls a SAAM region, which includes a number of routers. The server collects the global picture of the region and makes routing decisions on behalf of the routers. This method provides a lightweight router that performs only its primary task of forwarding packets to their destination addresses. To make its service scalable for large networks, SAAM organizes its SAAM servers in a hierarchy. At the first level of the hierarchy, SAAM partitions the network into autonomous regions. These regions are called SAAM regions.

SAAM assigns one SAAM server for each SAAM region. Those regional servers report to higher level servers that have control over regions of servers. This central management gives the SAAM system the advantage of having control over the entire network and enables the applications to have one point of reference from which to obtain QoS support. Instead of negotiating with local routers or regional servers for end-to-end service that they cannot guarantee, the service request is always sent to a suitable server that can make the correct decision for end-to-end service. Also, this design reduces the processing power requirements on the router side. Figure 1 illustrates the hierarchical structure of SAAM.

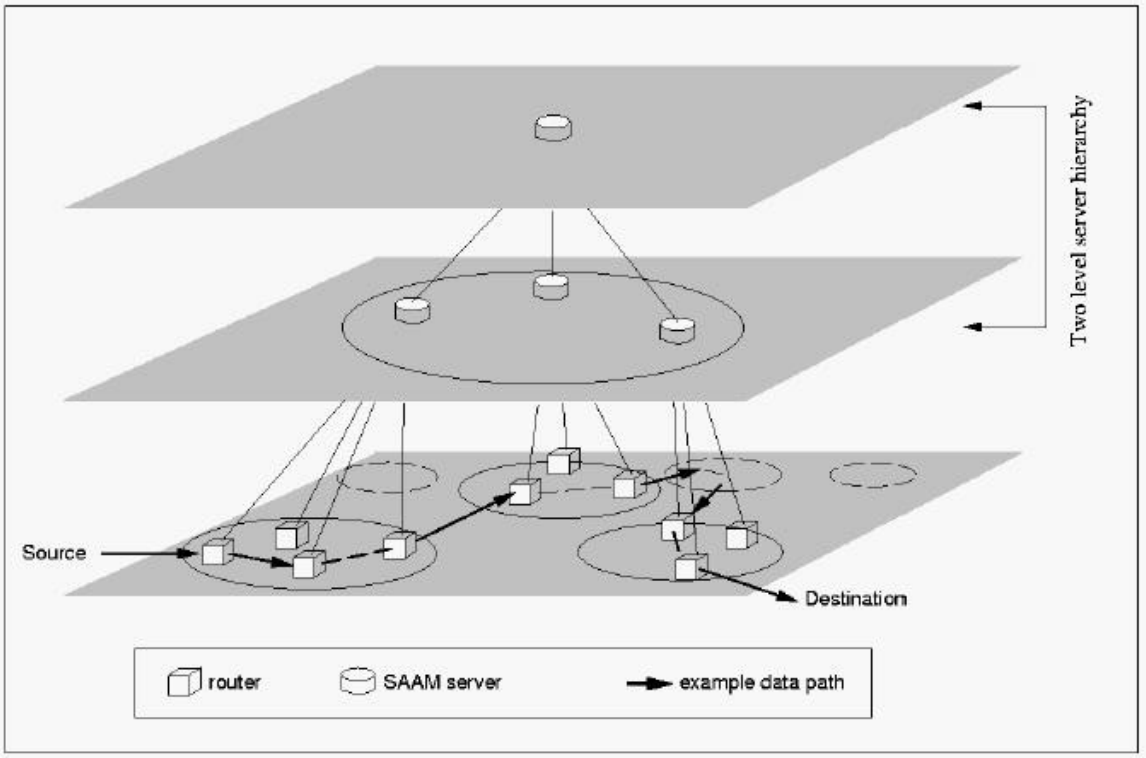


Figure 1. Hierarchical Organization of SAAM Servers [From [1]]

There are two major components in SAAM:

a. The SAAM Server

The SAAM server maintains an accurate picture of the QoS capabilities of the network by periodically retrieving link performance information from the routers and aggregating this information into a ready-to-use database of useful paths. This database is called the Path Information Base (PIB). By using the PIB, the SAAM server can efficiently implement network functions such as QoS, Best Effort Service, and re-routing of real-time flows, which are required to provide guaranteed and differentiated qualities of service.

b. The SAAM Router

The SAAM Routers handle actual data traffic from client applications. All client resource reservation requests enter the network through the edge routers, which forward the requests to the server for approval. The server responds to each request either by assigning a path to meet the request or by denying the request. When a path is assigned to a client for the first time, the server also sends route-update messages to all

routers in the path to install appropriate routing table entries to create the packet-forwarding path.

Each router is responsible for reporting its status to the server so that the server always has an accurate view of the network to make good routing decisions. Due to the hierarchical structure of the SAAM system, that status information first goes up to the regional server, and then is aggregated and forwarded by the regional server to servers higher in the hierarchy. This reduces the traffic on the network to update the router status, and keeps all decision makers informed of the status of each router.

The SAAM is designed to support resident agents allowing the components of a router to be upgraded and installed dynamically during the runtime. The precompiled byte code of a resident agent is registered as a new module of the receiving node. The module may run itself by creating a thread, or an existing thread may call it. For instance, if a node receives a server resident-agent it becomes a SAAM server; otherwise, it stands up as a router. This adds flexibility to the system so that the components of the server or the routers can be installed or uninstalled after the system is set and running according to the system's needs at that moment.

B. MOTIVATION FOR THIS THESIS

The SAAM architecture was initially designed with the next generation Internet (NGI) in mind in which increasingly sophisticated applications will provide QoS guarantees for their traffic flows in a SAAM network. Thus, most of the SAAM research has focused on matters of QoS pertaining to various parameter requirements and path constraints. Every QoS flow is centrally tracked by the SAAM server and its state maintained where necessary on SAAM routers.

One question that requires more attention is the manner in which SAAM will handle Best Effort (BE) traffic. Presumably, clients of a SAAM network provider will still have demand for BE services. This is because most of their applications do not require a guaranteed QoS from the network. Moreover, evidence suggests that deploying QoS to the Internet will be a very slow and expensive process. While these BE services may not generate as much revenue per byte as QoS streams, they will be used by a majority of applications in the foreseeable future. Therefore, SAAM must still handle the

BE traffic in a satisfactory manner to meet the performance expectation of the clients. At a minimum, the clients will expect the same range of packet latency and loss for BE traffic as the Internet already provides.

A Best Effort traffic management scheme has been added to SAAM in an earlier NPS M.S. thesis project [2]. The scheme intends to allow SAAM to route Best Effort traffic more intelligently than the existing Internet routing protocols, such as OSPF or RIP. It also aims to provide better performance in avoiding congestion and in achieving fairness than other similar proposals. However, the scheme was tested only on small network topologies with fewer than ten routers. The results were qualitative, and they were used to verify that the algorithm was implemented correctly. A more thorough evaluation of the efficiency and fairness of this solution remained as future work.

C. PROBLEM DEFINITION AND APPROACH

The focus of this thesis is to determine if SAAM's newly developed Best Effort traffic management scheme provides the intended improvements. If the result of this research shows that the performance of the SAAM scheme is not satisfactory, then the thesis will examine what caused the problem and what remedies are possible.

One way to evaluate SAAM's best effort management is to compare it with the current Internet's shortest path algorithms like RIP and OSPF. But this benchmark is very low. With the capabilities of a network management scheme like SAAM, achieving much better performance than those algorithms is expected. To evaluate the Best Effort Traffic management of SAAM objectively, comparing it with an equivalent solution is necessary. In this thesis, this approach is embraced and SAAM's BE solution is evaluated by means of comparing it with a similar state-of-the-art study among the literature.

Research must be conducted to find a solution comparable with SAAM's Best Effort management scheme. Once this solution is found, the next step will be to design and to implement it. Since SAAM's current simulation test-bed will be used, this new software component should be integrated into SAAM software, which is the reason this new software should be designed as a component of SAAM instead of a stand-alone software piece. Another concern in designing this new solution is that it should be easily

switchable with SAAM's current BE management scheme so that while running simulations, it is not necessary to hard code to change the BE management scheme.

Once this solution is built into SAAM, simulation testing begins by building test scenarios that validate key portions of this new BE traffic management solution. Those simulation results should provide similar results to those of the solution's authors.

The next phase is to run simulation tests to compare SAAM's current BE management scheme with the solution selected among the literature. Data that is necessary to compare the performance of those two solutions should be chosen before the simulations. After the simulation runs, the collected data will be analyzed. The result of the analysis should answer the questions: "Are there any performance inefficiency in SAAM's current BE management scheme? and Does it provide the intended performance improvement?"

After getting answers to these questions, possible remedies for the inefficiencies will be sought, if necessary. Corrective remedies will be implemented and tested in an iterative way. Once this iterative process is complete, recommendations will be made for areas of future research and improvement.

D. THESIS SCOPE

This thesis will evaluate and enhance the best effort traffic management scheme of SAAM. The evaluation will be based on experiments conducted using the SAAM emulated test-bed application. Realistic network topologies will be used in the experiments. The performance of the SAAM scheme will be compared to those achieved by the best of the related work. From the experimental results, the potential performance weaknesses of SAAM and the causes of those weaknesses will be identified. Possible solutions will then be developed. The project will involve code development for the SAAM test-bed application in the following areas: (1) creating XML-based SAAM test-bed configuration files for new test topologies, (2) implementing BE traffic management solutions used by the selected related work, and (3) implementing identified enhancements to the SAAM scheme.

E. THESIS ORGANIZATION

The remainder of this thesis is organized as follows:

Chapter II provides the background for this thesis work. Underlying concepts fundamental to traditional routing are discussed, as well as Best Effort traffic routing in Multi-Protocol Label Switched (MPLS) networks. Chapter III introduces the design of SAAM's current BE management scheme and the design of a selected MPLS Adaptive Traffic Engineering (MATE) algorithm [3]. Chapter IV gives the implementation details of the MATE algorithm. It describes the necessary changes that are made to the original MATE algorithm to adapt it to SAAM. Chapter V presents the testing and the results obtained in comparing two solutions. Chapter VI contains the corrections that are made to SAAM's current BE management scheme to improve its performance based on the simulation results. Chapter VII consists of recommendations and conclusions based on the overall thesis work.

II. BACKGROUND

A. BEST EFFORT VS. QUALITY OF SERVICE TRAFFIC

Today's Internet provides only one service model, the datagram service, which is also known as best-effort service [4]. With Best-Effort Service (BES), timing between packets is not guaranteed to be preserved, packets are not guaranteed to be received in the order in which they were sent, and the eventual delivery of packets is not guaranteed. A network that provides Best Effort Service does its best to deliver the packets to their destinations but does not give any guarantees. So a network that delivers all the packets to its destinations, and another one which does not deliver any of the packets are both called Best Effort service network. The type of traffic that does not require any guarantees—that satisfies with best effort service—is called best-effort traffic.

The IP protocol of the TCP/IP suite provides Best Effort service. On top of this, TCP provides reliability without any service guarantees by resending lost packets. Together, those two protocols are the de facto protocol suit of today's Internet. For best-effort traffic, (or simple data traffic) in which the only concern is having the packets eventually traverse from the source to the destination, this model has proven adequate. However, many other forms of traffic, for which a Best Effort service is not sufficient, are rapidly growing in volume. These forms of traffic require guarantees of a requisite quality of service, whether minimum bandwidth, maximum delay or loss rate, or a maximum delay variation. A partial list of these forms of traffic includes voice, video, and real-time data. Together, these forms are called quality-of-service (QoS) traffic.

Within a network that provides only Best Effort Service, QoS traffic encounters some performance problems. In BES networks, packets are delivered without concerning trip time. For an application for which trip time is important, this is not acceptable. Also Best Effort networks treat packets equally during congestion, so packets are dropped arbitrarily. This might bottleneck an application that is sensitive to packet losses. On the other hand, some applications require a constant delay variance (jitter). Of course, this can not be guaranteed with the capabilities of a Best Effort network.

Meeting these requirements of QoS traffic by providing more network resources maybe possible. However, with the increasing volume of QoS traffic, this is becoming more difficult and expensive. For this reason, other means of mechanisms are needed. The Internet Engineering Task Force (IETF) has proposed many service models and mechanisms to meet the demand for QoS. Notable among these models are the integrated services/Resource Reservation Protocol (RSVP) model, the differentiated services model Multi-Protocol Label Switching (MPLS), traffic engineering, and constraint-based routing.

B. BEST EFFORT ROUTING IN TODAY'S INTERNET

In today's Internet, packets are routed to their destinations by routers. Routers make their routing decisions based on some type of lookup table. These lookup tables are established by means of routing protocols.

In a global network, such as the Internet, it is highly unlikely that a single routing protocol will be used for the entire network. Rather, the network is organized as a collection of Autonomous Systems (AS), each of which comprises a set of routers that will, in general, be administered by a single entity. Each AS will have its own Intradomain Routing Protocol, which may differ significantly among the AS's. In order to share reachability information with each other, the AS's will use an interdomain routing protocol.

1. Intradomain Routing

There are generally two types of intradomain routing protocols; Distance Vector Routing Protocols and Link State Routing Protocols.

a. Distance Vector Protocols

With these simple protocols, a router updates a vector data structure for each of its neighbors, which contains entries for a destination address versus a single metric, usually a hop count. Routers communicate with their immediate neighbors and share their distance vectors through a relationship of implicit trust. Each router updates its distance vectors and then through some comparison of metrics (e.g., which neighbor has the shortest advertised hop count to a destination) updates its routing table entries for

each destination. Since these information broadcasts are periodic, distance vector protocols allow for dynamic conditions with no need of manual reconfiguration.

Of the distance vector protocols, Routing Information Protocol (RIP) was the most deployed routing protocol during the early growth of the Internet. RIP continues to be widely implemented today. RIP uses a simple hop count metric for route computation. RIP's major drawbacks are its hop count limit of 16 and simplistic path weighting (1 hop = 1 distance unit). Those drawbacks are addressed with RIP version 2. If its inclusion into the Internet Protocol Version 6 (IPv6) standards development process is any indication, RIP will be around for a long time yet.

b. Link State Protocols

These are more complicated than the distance vectors in a number of ways. First, rather than just obtaining and trusting information from its immediate neighbors, each router will obtain information from every other router in the network. Second, rather than selecting routes based on a simple metric comparison, each router will build a graphical representation of the network from the information it receives and then use the shortest path algorithms to determine the next hop for each destination. For this reason, link-state algorithms are sometimes referred to as shortest-path first algorithms. Overall, link-state algorithms are more computationally intensive and require more information sharing than distance vector protocols. Indeed, link-state information has to be flooded onto a network to ensure that every single router receives every other router's link-state information. Usually, this involves many redundant message transmissions. The number of hosts and acceptable bandwidth overhead are usually considered when tuning the link state flooding frequency.

The premier link-state protocol is the Open Shortest Path First (OSPF) Interior Gateway Protocol, developed by the IETF to replace RIP. OSPF was designed with scalability and rapid convergence in mind. Rather than constantly flooding redundant information, OSPF only advertises on topology changes and at that, only sends changed information. Apart from this, short "I'm alive" messages are exchanged infrequently to verify established connections. OSPF is more sophisticated than RIP in that its path metric can be set to something besides hop count. The most popular choice is

a number inversely proportional to bandwidth so that small links will be preferentially avoided due to their large weighting.

2. Interdomain Routing

The two most prevalent protocols for interdomain routing are the Exterior Gateway Protocol (EGP) and the Border Gateway Protocol (BGP). These protocols operate through neighboring AS border routers sharing reachability information. When a border router receives an outbound packet from its AS, it must determine by which neighboring border router that destination is reachable. EGP was the predecessor of BGP and is rapidly being phased out due to scalability issues. Specifically, EGP assumes the Internet is still arranged as it once was, in a tree-like topology. BGP, on the other hand, allows all the Internet's AS's to be connected arbitrarily. BGP only presumes router memory space as the amount of reachability information can be overwhelming sixty-four MB, for example, is no longer sufficient without some form of route aggregation.

Reachability information consists of long lists of domains that are next to each other in the Internet. A BGP router will receive such an AS sequence, append its own AS to that sequence, and then advertise this new information to neighbors. Loops are avoided by checking new AS sequence lists to ensure a BGP's own AS number is not already included. This, however, does not prevent selecting a long route when shorter routes exist, only loops.

C. NEW APPROACHES FOR ROUTING

While RIP, OSPF, and other protocols have proved sufficient for routing BE packets, research continues in search of greater efficiencies, optimization, and other criteria such as congestion avoidance and resolution. Ultimately, such research focuses on building a better routing algorithm, finding better inputs for existing algorithms, or making better use of algorithm outputs.

Research has been done on how best to tune a network by adjusting path weights. But a drawback for this approach is that while changing path weights resolves congestion at some part of the network, it may cause problems on the other side of the network. For those solutions that allow path weights to change dynamically with traffic conditions, another variable is considered: time constant. Networks that respond too quickly or with

too much of an adjustment for a change in conditions can experience instability. Instability describes the situation in which a network undergoes large swings in traffic conditions while attempting to correct itself: It never converges on a new solution. Part of this parallels the classical control theory problem of setting the time constant too small to allow the last error correction to effect feedback. It is also compounded in networks where many routers may be acting autonomously to correct the same problem. Without coordination, they may be continually shifting traffic from previously congested areas and creating new areas of congestion so that their algorithms will be constantly hunting and sabotaging each other's solutions.

Besides modifying the algorithms or their inputs, it is also possible to change how outputs are used. The shortest-path routing has usually involved selecting the single shortest path to a destination and then routing all of that destination's traffic onto that path. One of the first alternatives to be proposed was a method called Equal Cost Multi-Path (ECMP). In this approach, if the routing algorithm found multiple "shortest" routes of equal cost, then traffic would be evenly split along those paths. Besides ECMP, other solutions involve splitting traffic to a destination along the next shortest route(s).

Another area of research is the management aspect of BE traffic. Currently, there is no management in the Internet without specialized devices. TCP performs the end-to-end management of a session that does not exist in the underlying IP protocol. TCP functions above IP and delivers to the user or application the full transmission of packets in the proper order. TCP simply tracks a session on a packet-by-packet basis and performs retransmission or reordering as needed. This is management on the user end. TCP also performs an important form of management for the Internet at large, known as the TCP rate control. Even if a group of IP routers unintelligently sends all of their traffic over the same path and fails to respond to resulting congestion, TCP will cause the end-to-end sessions comprising that traffic to throttle back on the send rates. This is the only control mechanism in today's Internet. Rather than reducing congestion by using alternate paths it reduces it by delaying traffic submission and, thus, delivery.

This form of control continues to work today because the majority of Internet applications run on TCP. Contending TCP flows cooperate and reduce their transmission

rates while non-TCP flows continue unchecked and take advantage of the TCP applications' selflessness. However, the number of audio/video applications is growing and it is feared that this will cause the percentage of non-TCP traffic to grow, and with it the suitability of TCP based flow control to shrink.

By refining the TCP protocol and hoping that everyone will use it, congestion could be managed by the hosts at each end of the session. But there is not a satisfactory enforcement mechanism for this approach. Further, it is widely open to exploitation by malicious users. An alternative to this end-to-end management scheme is the router-based management. While end-to-end management is simpler router control could potentially enforce some degree of fairness. At the very least, it would require a switch apply aggregate scheduling to per flow queuing. But, currently, only small groups of routers take on management functions, and then only for QoS traffic with solutions varying among proprietors. For BE traffic in the greater Internet there is no router-based management solution. TCP continues to be relied on for all management aspects of the BE flows while the underlying IP routers perform packet forwarding.

D. TRAFFIC ENGINEERING

Network congestion can be caused by a lack of network resources or an uneven distribution of traffic. In the first case, all routers and links are overloaded, and the only solution is to provide more resources by upgrading the infrastructure. In the second case, some parts of the network are overloaded while other parts are lightly loaded. A new initiative, called traffic engineering, has emerged to solve the second problem. The IETF has embraced this approach and established the Traffic Engineering Working Group. This group clarifies and develops this new approach with Requests for Comment (RFC's) and Internet Drafts.

In [5], Traffic Engineering is defined as "that aspect of Internet network engineering dealing with the issue of performance optimization of operational IP networks. Traffic Engineering encompasses the application of technology and scientific principles to the measurement, characterization, modeling, and control of Internet traffic." Traffic engineering can also be defined as the task of mapping traffic flows onto an existing physical topology. Traffic engineering is a powerful tool that can be used by

ISPs to balance traffic loads on the various links, routers, and switches in the network so that none of these components is over-used or under-used.

Numerous research efforts have focused on how to incorporate traffic engineering into existing protocols. Although incorporating traffic engineering into existing shortest path algorithms improve the efficiency of the network, those algorithms suffer from several limitations [3]. Some of the limitations are

- load sharing cannot be accomplished among paths of different costs,
- traffic/policy constraints are not taken into account,
- modification of link metrics to re-adjust traffic mapping tends to have network wide effects, causing undesirable and unanticipated traffic shifts,
- traffic demands must be known beforehand.

Multi-Protocol Label Switching (MPLS) [6] has emerged as one of the foremost technologies through which traffic engineering can be realized. This is because traffic engineering makes abstractions of both data and hardware. Instead of discussing packets, links, and nodes, traffic engineering discusses data as "flows" and "trunks" and physical hardware as "paths" and "routes." MPLS fits this schema with its simple mechanism of packet labeling and label-switched paths (LSP's).

MPLS is an advanced forwarding scheme. It extends routing with respect to packet forwarding and path controlling. Each MPLS packet has a label. An MPLS-capable router, called a Label Switching Router (LSR) examines the label. At ingress LSRs of the MPLS-capable domain, packets are classified and routed based on a combination of the information carried in the IP header of the packet and the local routing information maintained by the LSRs. An MPLS label is then attached to each packet. Within a MPLS-capable domain, a LSR will use the label as the index to look up the forwarding information in a table of the LSR. The packet is processed as specified by the forwarding table entry. The incoming label is switched with the outgoing label, and the packet is switched to the next LSR. The paths between an ingress LSR and an egress LSR are called Label-Switched Paths (LSPs). MPLS uses some signaling protocol, such

as Resource Reservation Protocol (RSVP) or Label Distribution Protocol (LDP) to setup LSPs.

Current chapter discussed the routing schemes in use and the new routing approaches. Among those new approaches, specifically the traffic engineering was explained in detail. The next chapter, explains two routing schemes, the SAAM's BE solution and the MATE algorithm, which are the examples of traffic engineering.

III. SAAM'S BEST EFFORT TRAFFIC MANAGEMENT AND THE MATE ALGORITHM

A. DESIGN OF SAAM'S CURRENT BEST EFFORT TRAFFIC MANAGEMENT

Currently, SAAM handles Best Effort (BE) traffic management with two components, BestEffortTable and BestEffortManager, introduced by [1].

BestEffortTable is a software agent installed on edge routers. It is basically a lookup table that maps BE traffic onto paths. Best Effort packets reach the edge routers (ingress routers) without any flow label (path ID) since no flow reservations occur for BE traffic. BestEffortTable provides this missing path ID and, thereafter, the BE packets can be routed easily by the core routers based on the path ID.

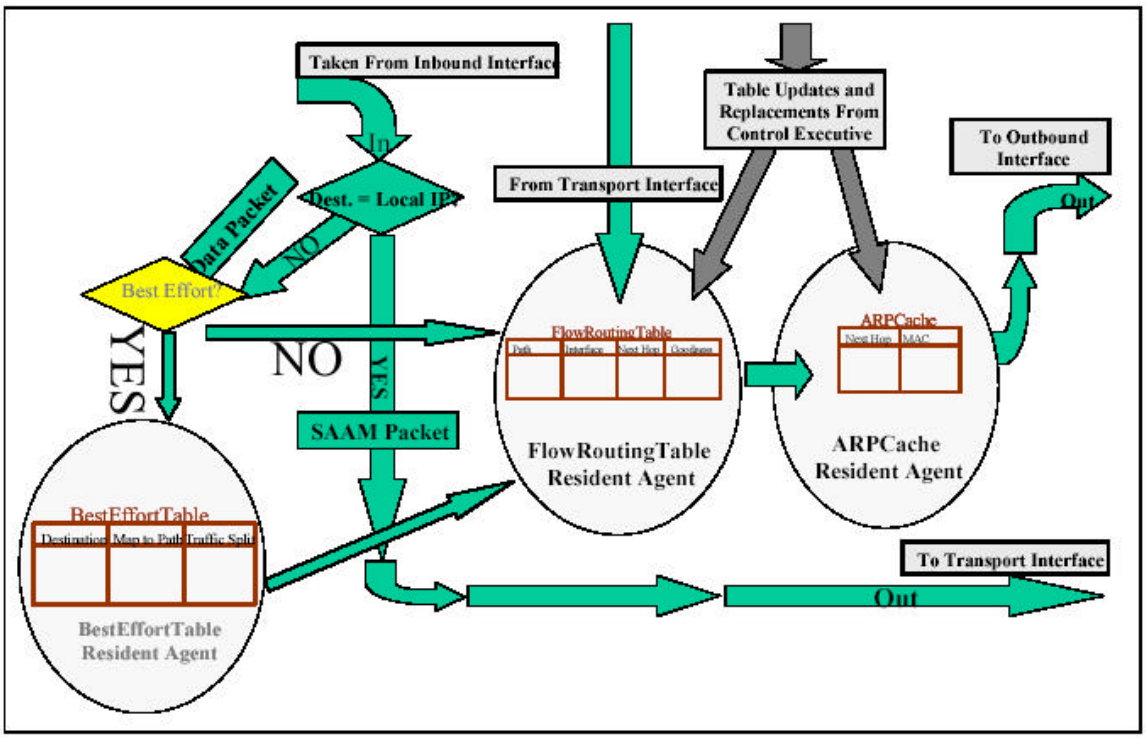


Figure 2. SAAM's Current Routing Method (From [1])

BestEffortTable does its traffic mapping according to advice given by BestEffortManager. BestEffortManager is a software component on the SAAM Server that selects and deploys paths for BE traffic. Communication between BestEffortManager and BestEffortTable is done through message exchanges as shown in Figure 3.

Periodically, Link State Advertisement (LSA) messages, received from all SAAM routers, provide a complete view of the network to the SAAM Server. Edge routers send EdgeNotification messages when receiving BE traffic with a new destination to allow the Server to select and deploy new paths for the BE traffic. Additionally, BestEffortManager uses CongestionAdvisory messages to provide performance updates for the deployed BE paths to edge routers. The edge routers will shift traffic between the BE paths based on these performance updates.

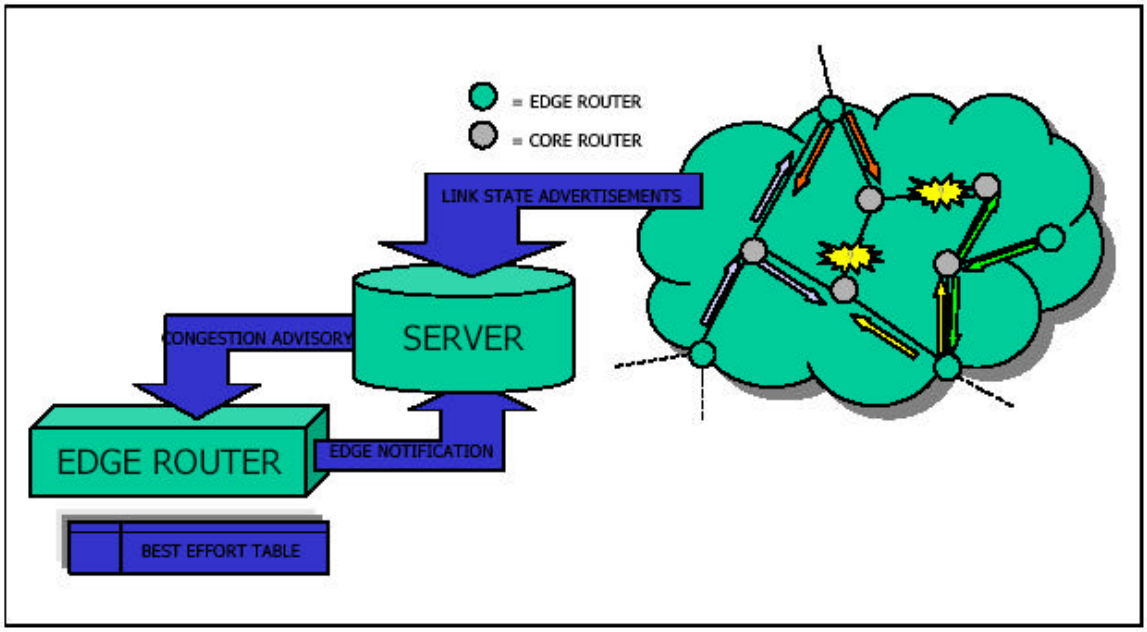


Figure 3. Server/Edge Router Communication (From [1])

The following explains the functions of BestEffortTable and BestEffortManager in detail.

1. BestEffortTable

In a SAAM network, packets are routed according to their path IDs embedded in the flow label field of the packet header. But unlike QoS traffic, Best Effort traffic, entering a SAAM network, does not have assigned path IDs. An appropriate pathID is inserted into each BE packet with the help of BestEffortTable. This process is done once when the packet enters the network at the ingress router and the pathID is carried by the packet as long as it stays in the SAAM region.

BestEffortTable is simply a lookup table that consists of three columns: "DestIPv6 Address", "Map to Path" and "Traffic Split". The first column keeps the IP version 6 address of a destination. The second column holds a path ID that may be assigned to BE packets that go to the destination as specified in the first column. The SAAM server typically deploys multiple paths for one destination to the BestEffortTable agent. The third column shows the percentage of traffic going to one destination that is currently routed over the corresponding path.

a. Destination Management

The BestEffortTable agent manages BE traffic on a per-destination basis. The SAAM server typically deploys two active routes for each destination to balance the load. If possible, the server also deploys two spare routes. Those route entries are sent by BestEffortManager to the BestEffortTable agent by way of BestEffortTableEntry messages. Each BestEffortTableEntry message introduces either a new destination address or provides a new path for a previously known destination address.

For each new destination address, BestEffortTable receives two active routes. The first received path is the primary path for this destination and the traffic split of this path is automatically set to 100%. The second path received is the alternate path and the traffic split for this path is set to 0% initially.

If the BestEffortTable agent receives a third path, this is added to the table with the traffic split set to 0%. If a fourth path is received, it is the alternate path for congestion bypass. Consequently, the BestEffortTable agent resets the splits of the first two paths to 0% and sets the split of the third path to 100% as the new primary path. The first two routes stay in the table until more routes are received, in which case they will be overwritten by the new paths. In other words, the BestEffortTable agent holds a maximum of four paths for one destination.

b. Traffic Splitting

To balance the load among multiple paths, a traffic split mechanism is necessary. For example, one path carries 20 percent of the traffic and the other carries 80 percent. The issue is how to achieve the split precisely.

The current implementation splits the incoming traffic into 10 buckets. This partitioning is based on a hash function. The incoming packets are hashed according to their source IP addresses. Then those buckets are assigned to the paths according to the path's split value. For example; if a path has a split of 30, then three buckets are assigned to this path. The number of buckets is a parameter that can be adjusted.

c. Load Balancing

BestEffortTable does load balancing among two active routes. Load balancing involves shifting traffic from one path to another based on CongestionAdvisory messages received from the SAAM server. Two processes provide this functionality: redirection and reversion. Redirection simply shifts traffic iteratively to an alternate path when congestion on the primary path is detected. When the server notifies the BestEffortTable agent that the congestion is cleared, the BestEffortTable holds the current split first. Thereafter, it starts reversion of traffic. Reversion is shifting traffic back to the primary path. This is done for performance reasons since the primary path was selected as the shortest or the widest path or both. The third and last mechanism related to load balancing is called switch-back. This is simply a shortcut to the long reversion process. When the BestEffortTable receives a new complement of routes, it resets the traffic split to 100/0. The server employs this feature by sending the same routes that the BestEffortTable already has, which causes the shifting of traffic to the primary path instantly. Otherwise, this process is done by reversion iteratively.

Redirection and reversion frequencies are different. Since redirection involves shifting traffic from a congested region of the network, it is a more rapid process. In the current implementation, the redirection interval is set to be the same as the AutoConfigurationCycle, which is the period for Link State Advertisement (LSA) messages to be sent by the routers to the server. That means the server's view of the network is updated after each AutoConfigurationCycle. The AutoConfigurationCycle is typically set to several hundred milliseconds. The reversion period is much larger, e.g., 30 minutes. Unlike redirection, it is not an urgent process. Also if the reversion period were short, traffic may often be shifted to a region where congestion has not subsided.

d. Fault Tolerance

The SAAM server detects a node or link failure in two auto configuration cycles. After detection, BestEffortManager immediately notifies the BestEffortTable agent via a CongestionAdvisory message. That triggers the BestEffortTable agent to shift traffic from the failed paths to the surviving spare paths.

2. BestEffortManager

The BestEffortManager is the orchestra chief on the server end that controls the Best Effort traffic. Its tasks consist of maintaining the Best Effort topology, sending paths for BE traffic, sending congestion advisories, and maintaining statistics. The BestEffortManager always functions in one of two modes: reactive monitoring or proactive monitoring.

a. Best Effort Topology Maintenance

The BestEffortManager maintains a topology, different from the SAAM network topology, which is called the BE topology. It is a smaller version of the overall SAAM topology. It would be easier to maintain the same topology as the overall SAAM topology, but this would not scale well. Also, this is not necessary since not all the SAAM routers are origins or destinations of BE traffic. Instead, the BestEffortManager maintains a topology that only contains possible origins and destinations of BE traffic. This topology includes the edge routers that received the BE traffic and the regional interface addresses that appear in the destination address field of the BE packets.

This BE topology is established by an edge router discovery process. Every SAAM router is a core router initially. A core router cannot directly accept BE traffic from customers until its BestEffortTable functionality is activated. This activation process occurs as follows: a core router, that receives BE traffic without pathID, sends an EdgeNotification message to the server, to which the server responds with path routing information. The edge router also sends additional EdgeNotification messages when it receives BE packets with new destination addresses.

After receiving all the EdgeNotification messages, the BestEffortManager builds the BE topology by connecting each pair of BE origin and destination through two paths.

b. Reactive Monitoring

This is the default mode of the BestEffortManager. During reactive monitoring, the BestEffortManager monitors the link state advertisements for the BE traffic loss rate information and then reacts when problems are detected. Specifically reactive monitoring is designed to combat local congestion that can be resolved by a shifting traffic among existing BE paths.

When the BestEffortManager detects congestion between two BE nodes, it notifies the source router to begin bad balancing in an attempt to resolve the congestion locally. Then the BestEffortManager waits for a fixed amount of time, called the Congestion Bypass Time. In the current configuration, Congestion Bypass Time equals the number of buckets multiplied by the duration of an Auto Configuration Cycle. In this way, the server allows the source router sufficient time to potentially shift all the traffic from the primary to the alternate path in an effort to resolve the congestion.

After the Congestion Bypass Time, if the congestion is not resolved, the BestEffortManager starts the congestion bypass procedure. This procedure involves deploying new paths that bypass the congested region. The old paths are deactivated for 30 minutes. If no paths are available for the congestion bypass, then the BestEffortManager declares the network to be globally congested and initiates global congestion resolution procedures.[1]

c. Proactive Monitoring

This is the mode which the BestEffortManager enters during global congestion periods. During global congestion, the main aim of the BestEffortManager is to provide fairness between the BE flows. The first step to provide fairness is keeping statistics to determine which BE traffic is disproportionately penalized and which traffic is receiving more than its fair share. After this, the BestEffortManager attempts to enforce fairness through a “rob from the rich, give to the poor” Robin Hood approach. The “rich” are those aggregate BE traffic flows that experience a packet loss rate one standard deviation less than the mean packet loss rate. Similarly, the “poor” are those

experiencing a packet loss rate at least one standard deviation above the mean packet loss rate [1].

In "robbing from the rich", the BestEffortManager reclaims the "rich" flow's least widest path. This is done in an attempt to alleviate congestion in more heavily congested paths. In "giving to the poor," the BestEffortManager first reclaims deactivated BE paths that are currently congestion free. Then, the BestEffortManager decides which of two methods might increase a flow's available bandwidth: switching back to the primary path or deploying a reclaimed path. For the former, the BestEffortManager orders the BestEffortTable to begin the switchback process. For the latter, the BestEffortManager deploys the reclaimed paths to increase bandwidth and alleviate congestion. In either case, if an action is taken, the BestEffortManager waits one local resolution timeout to allow the problem to resolve before taking any further action. The local resolution timeout is set to two seconds. If no action is possible, or there are no rich or poor flows, then the BestEffortManager waits until the congestion subsides or a poor or rich flow appears.

3. Routing Algorithms

Three routing algorithms are used to select BE paths: The Shortest Widest Path (SWP), the Shortest Widest Most Disjoint Path (SWMDP) and the Shortest Widest Least Congested Path (SWLCP).

The SWP algorithm chooses the shortest of the widest paths between two nodes. It does this by first determining the widest paths in terms of bandwidth. If there are multiple widest paths, then the shortest–in terms of hop count–among those is selected. This is used when selecting the primary path. In BE traffic routing, the main purpose is to avoid congestion. In order to route BE traffic to its destination without congestion, wide paths are the best choice.

The SWMDP algorithm chooses the shortest of the widest paths which are also the most disjoint compared to some reference path between two nodes [1]. It first finds the paths that have the least nodes and links in common with some reference path. Then among those paths the shortest of the widest is selected. This is used when selecting an

alternate path. Since the alternate path is used for load balancing and fault tolerance, an alternate path, which is as much disjoint as possible from the primary path, is preferred.

The SWLCP algorithm selects the shortest of the widest of the least congested paths. The least congested refers to the path with the least packet loss rate. This algorithm is employed during the congestion bypass. During congestion it is desirable to shift traffic to less congested paths to bypass the points of congestion. Thus, SWLCP is the algorithm of choice.

4. Messages

Three messages are used for communication between the BestEffortManager and the BestEffortTable agent of a BE source node. These messages are as follows.

An EdgeNotification message is sent by the edge router to the BestEffortManager help build the BE topology as explained previously. A BestEffortTableEntry message is used by BestEffortManager to deploy paths to the BestEffortTable agents. Specifically, the BestEffortTableEntry contains the path ID to be deployed and the destination address for which the path is deployed. CongestionAdvisory messages are used by the BestEffortManager. They carry congestion advisory information to the BestEffortTable agents. The BestEffortTable agent conducts load balancing according to those advisories. The two main fields of this message are path ID and a color coded advisory type. Three color codes are used. RED represents path failure, YELLOW means congestion on the path, and GREEN means no congestion.

B. DESIGN OF MATE ALGORITHM

This thesis compares SAAM's current BE solution with a similar solution from the literature. So a literature search was done to find a BE management solution that is comparable with the SAAM's approach. Most of the traffic engineering architectures that have been published were explored. The research was then narrowed down to two papers: [3] and [7]. Between them, the MATE paper [3] is referenced more by other papers. The other [7] is a more recent paper. It uses the Additive Increase Multiplicative Decrease (AIMD) control approach to achieve the traffic engineering objectives. Although it is a promising approach, [7] contains some unrealistic assumptions. For example, it assumes no common link or node between any pair of deployed BE paths. Obviously, this is not a

realistic assumption. For this reason, MATE was selected as a reference model for SAAM's approach.

1. Overview of MATE

There are two types of traffic engineering schemes: time-dependent and state-dependent. Time-dependent mechanisms use historical information, based on variations in traffic, over time to pre-program path layout and traffic assignment. They consider only long-term traffic variations, not unpredictable, short-term variations. Conversely, state-dependent traffic engineering is designed to address short-term network changes. MATE is a state-dependent traffic engineering scheme.

MATE is designed to work on MPLS networks. It achieves traffic engineering objectives by balancing the load among explicit LSPs (paths) established between each node pair in an MPLS domain. It assumes that between 2 and 5 explicit LSPs between each node pair have been established using a standard protocol. The MATE algorithm does not cover the mechanism to select and establish the LSPs.

Once the LSPs are setup, the ingress Switched Routers (LSR) distributes the traffic among the LSPs so that the traffic loads are balanced and congestion is thus minimized. All traffic between an ingress and egress node pair, referred to as an aggregated flow, is engineered at the ingress LSR. Although MATE can be used both for differentiated traffic and Best Effort traffic, it designed mainly for Best Effort traffic.

Some of the main features of MATE are

- Traffic engineering on a per-class basis,
- Distributed load-balancing algorithm,
- End-to-end protocol between ingress and egress LSR,
- No new hardware or protocol requirement on intermediate LSRs, nor a priori traffic distributions,
- No assumption on the scheduling or buffer management schemes at the LSR,
- Optimization decisions based on LSP congestion measures,
- Minimal packet reordering due to traffic engineering,
- No clock synchronization between two LSRs.

Figure 4 shows the functional diagram of the MATE component at an ingress node. Incoming traffic enters a filtering and distribution process whose main purpose is to group traffic before split among the LSPs in order to minimize the possibility of having packets arrive at the destination out of order. This is necessary since TCP applications are sensitive to out of order delivery of packets.

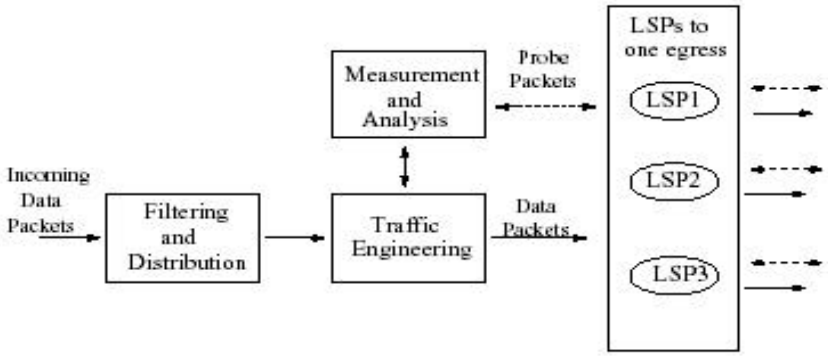


Figure 4. MATE Functions in an Ingress Node (From [3])

The Traffic Engineering function decides when and how to shift groups of traffic among the LSPs. This is done based on LSP performance statistics, which are obtained from measurements taken using probe packets. The Traffic Engineering function consists of two phases: a monitoring phase and a load balancing phase. In the monitoring phase, if an appreciable and persistent change in the performance of some of the LSPs is detected, a transition is made to the load balancing phase. In the load balancing phase, the function tries to equalize the performance among the LSPs. Once determining the performance measures of different LSPs to be within tolerance, the function returns to the monitoring phase and the whole process repeats.

The role of the Measurement and Analysis function is to obtain one-way LSP statistics, such as packet delay and packet loss, used by the Traffic Engineering function. It transmits probe packets periodically to the egress node, which immediately returns them to the ingress node. The round trip delays of the probe packets are used to derive an

estimate of the one-way packet delay, while the percentage of probe packets not returned is used to compute an estimate for the packet loss rate.

2. Filtering and Distribution

MATE performs a two-stage traffic distribution process. First, MATE distributes the traffic for a given ingress-egress pair equally among a small number of bins at the ingress LSR. MATE does not enforce a single kind of traffic filtering scheme. Instead, it offers some possibilities. One of the offered schemes is to distribute the traffic on a per-packet basis without filtering. For example, one may distribute incoming packets at the ingress LSR to the bins in a round-robin fashion. Although it does not have to maintain any per-flow state, the method suffers from potentially having to reorder an excessive amount of packets for a given flow, which is undesirable for TCP applications.

An alternative would be filtering the traffic on a per-flow basis (e.g., based on <source IP address, source port, destination IP address, destination port, IP protocol> tuple). Although per-flow traffic filtering preserves packet sequencing, it must maintain a large number of states to keep track of each active flow.

Yet another method is to filter the incoming packets by using a hash function on the IP field(s). The fields can be based on the source and destination address pair, or other combinations. The purpose of the hash function is to randomize the address space to prevent uneven clustering. Traffic can be distributed into the bins by applying a modulo-operation on the hash space. Note that the packet sequence of each flow is maintained with this method.

After the engineered traffic is distributed into the bins, a second function maps each bin to the corresponding LSP. The rule for the second function is very simple. If LSP(i) is to receive twice as much traffic as LSP(j), then LSP(i) should receive traffic from twice as many bins as LSP(j).

3. Measurement and Analysis

MATE does not require each LSR to perform traffic measurement. Only the ingress and egress LSRs are required to participate in the measurement process.

MATE measures the packet delays along the LSPs as the congestion metric. The delay of a packet along an LSP is obtained by transmitting a probe packet from the ingress LSR to the egress LSR. The probe packet is time-stamped at the ingress LSR at time T1 and recorded at the egress LSR at time T2. If the ingress' clock is faster than the egress' clock by Td, then the total packet delay (i.e., queuing time, propagation time, and processing time) is T2-T1+Td. A collection of probe packets can easily yield an estimate on the mean packet delay Tm+Td, where Tm is the long-term average of T2-T1. One important point to note is that the value of Td is not required when only the marginal delay is needed. MATE exploits this property by relying only on marginal delays rather than absolute delays. Therefore, clock synchronization is not necessary.

Packet loss probability is another metric that MATE offers as an alternative congestion metric. Packet loss probability can be estimated by encoding a sequence number in the probe packet to tell the egress LSR how many probe packets transmitted by the ingress LSR and another field in the probe packet to indicate how many probe packets received by the egress LSR. When a probe packet returns, the ingress LSR can estimate the one-way packet loss probability based on the two fields. The advantage of this approach is that it is resilient to losses in the reverse direction.

4. Traffic Engineering

a. Traffic Engineering Problem

Given the following parameters for each ingress-egress pair s:

Offered Load : a_s

Set of LSPs : P_s

Vector of Traffic Splits : $?_s=\{?_{sp} | p \in P_s\}$

Each ingress node assigns fraction λ_{sp} to path p in P_s according to some traffic engineering objective.

Let C_p be the cost function (e.g., packet delay) of LSP p. It is a function of the link utilization λ_{sp}. The cost function should be measurable and convex. In this

context, the cost function should increase with the load. Packet delay and packet loss rate are examples of convex cost functions. Consequently the traffic engineering objective can be formalized as follows. Each ingress node splits traffic in a way that minimizes the total cost.

$$\text{Minimize} \quad C(?)=\sum_{p} C_p(?) \qquad (1)$$

$$\text{s.t.} \quad \sum_{p \in P_s} ?_{sp} = a_s \qquad (2)$$

$$?_{sp} \geq 0. \qquad (3)$$

b. Optimality

The MATE authors presented a theorem that defines the optimal traffic split. They proved the correctness of their theorem. According to the theorem a split, λ, is optimal if and only if, for each ingress-egress pair, all paths with positive flow have minimum (hence equal) cost function derivatives. [3]

c. Gradient Projection Algorithm

To achieve optimality, MATE uses a gradient projection algorithm to find the optimal split.[3] In this algorithm, routing is iteratively adjusted in the opposite direction of the gradient and projected onto the feasible space defined by equations (2) and (3). Each iteration of the algorithm takes the form:

$$\lambda_s(t+1) = [\lambda_s(t) - \gamma \nabla C^s(t)]^+.$$

$\lambda_s(t)$: vector of traffic splits for pair s

$\nabla C^s(t)$: vector of (measured) path cost derivatives

γ : a preset step size

In essence, the gradient projection algorithm shifts traffic from paths with the highest derivatives to paths with the lowest derivatives by a small amount with each iteration. The cost derivative is the rate of change in cost as a function of traffic load.

d. *Asynchronous Environment*

The asynchronous environment is necessary in traffic engineering solutions. This is because if all ingress nodes start load balancing synchronously, then the result would be transferring congestion from one area to another.

In MATE, each ingress egress node pair probes the network independently hence asynchronously. Also the delays of the probe packets are different for each ingress-egress pair. These features ensure traffic split adjustments of different node pairs are not synchronized.

5. Discussion

For best-effort traffic, the traffic engineering objective is to avoid congested LSPs. The congestion measure can be characterized by the sensitivity of the LSP to changes in the offered load. An LSP is said to be sensitive if a change in the load results in a significant change in the mean packet delay. MATE computes the derivative of the mean packet delay with respect to the offered load to quantify the sensitivity of an LSP. For a given LSP, the derivative increases as the load increases. This is evident since the mean delay is an increasing and convex function with respect to the load. The derivative can be derived by observing the mean delays over two different loads.

MATE performs load balancing by simply equalizing the derivatives across the LSPs. The implementation is simplified since only the relative derivatives must be known. MATE essentially applies a Min-Max operation to minimize the load on an LSP that has maximum congestion. The objective is also equivalent to minimizing the sum of all LSP delays. MATE can be easily made to respond to changes in network state adaptively as long as the traffic is quasi-stationary. Network changes can be tracked by periodically measuring packet delays. If the changes in delays exceed a certain threshold, the algorithm minimizes the derivative of the most congested LSP by re-equalizing the derivatives.

An LSP is said to be lossless if each LSR along the LSP never discards packets. Otherwise, the LSP is said to be lossy. MATE operates in a lossless mode unless it detects a packet loss. For the lossy case, the packet loss information must be incorporated

into the load balancing process. This is done by incorporating each packet loss as a fixed (large) delay.

MATE employs a four-phase algorithm for load balancing.[8] The first phase initializes the congestion measure for each LSP. In the second phase, the algorithm tries to equalize the congestion measure for each LSP. Once the measures are equalized, the algorithm moves to the third phase. The algorithm monitors each LSP in the third phase. If an appreciable change in the network state is detected, the algorithm moves to the fourth phase in which the congestion measures are appropriately adjusted. Then, the algorithm proceeds to the second phase and the process repeats.

C. SUMMARY

In this chapter, SAAM's BE solution and the MATE algorithm were introduced. The main software components of SAAM's BE solution were described. while the MATE algorithm was presented analytically. The next chapter will explain how the MATE algorithm was integrated into the SAAM code.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. IMPLEMENTATION OF MATE ALGORITHM IN SAAM

In order to compare SAAM's best effort management scheme with MATE algorithm, it is necessary to perform some simulation runs. SAAM's own test bed is the natural choice of simulation environment for the simulation runs. Thus MATE algorithm should be integrated into SAAM.

In addition to the original MATE paper [3], [8] was used as a guide while implementing MATE algorithm. [8] is an Internet draft where MATE authors explain the implementation details of the algorithm. While using those references as guides, this work does not follow them completely. Some design changes to the original MATE algorithm had to be made in order to make MATE work within SAAM. While making those changes, necessary care was taken to ensure that the new algorithm would achieve the same performance level as the original MATE algorithm. Moreover, some necessary corrections were made to the original MATE algorithm to allow it to operate under realistic network conditions. This chapter explains these modifications as well as the added software components of the MATE implementation in SAAM.

A. MODIFICATIONS TO ORIGINAL MATE ALGORITHM

1. Cost Function

As mentioned in Chapter II, MATE defines a cost function for an LSP. The MATE algorithm uses the cost function to represent the current performance level of the LSP. The derivative of this cost function with regard to load is considered the sensitivity metric of the LSP. At a traffic source (ingress router), the algorithm tries to equalize the sensitivity metrics of all LSPs to a destination in an attempt to avoid congestion. MATE authors proposed in [3] and [8] a cost function based on packet delay or packet loss rate. In their scheme, delay is used as the main congestion measure and packet loss rate is incorporated into delay by treating each packet loss as an event where a packet has experienced a fixed (extreme large) delay.

The MATE implementation of this thesis uses the packet loss rate of an LSP as the cost function rather than a combination of packet delay and loss rate. For best effort traffic, unlike QoS traffic, delay is not a major concern. The main concern in best effort

traffic management is to achieve low packet loss rate. That is why this work uses packet loss rate as the sole congestion measure.

2. Traffic Measurement

The original MATE algorithm uses probe packets to measure packet delays. Each ingress router periodically sends probe packets to the egress router along every LSP between the two routers. The egress router immediately turns these probes back allowing the ingress router to derive sample end-to-end packet delays over the LSPs. In SAAM, this is not necessary. At every Auto Configuration Cycle, each SAAM router sends its Link State Advertisement message to server. The Link State Advertisement message contains link state information such as packet delay and loss rate. Therefore the SAAM server has the information that MATE needs. The MATE implementation of SAAM uses server-assisted traffic measurement. In this scheme, the SAAM server periodically sends a packet loss rate update to each ingress router via a BestEffortPerformanceUpdate message. Instead of updating the packet loss rate for one LSP at a time, the server sends packet loss rate information for all LSPs of a specific ingress router in one message. The period of BestEffortPerformanceUpdate is set to be two times the Auto Configuration Cycle (ACC). ACC is typically in the range of several hundred milliseconds. Since MATE does not provide any traffic measurement period, this value is set based on observations from simulation runs. A parametric analysis may be done to achieve optimal performance. This analysis is beyond the scope of this thesis.

3. Path Selection and Path Deployment

MATE algorithm does not cover the path selection and path deployment mechanisms. It basically assumes that multiple LSPs have been deployed between ingress-egress node pairs. But it is worth to mention those mechanisms in this work since they affect the performance.

The MATE implementation of SAAM uses the same path deployment scheme as used by SAAM's own best effort management solution.[1] In this scheme, an edge (ingress) router sends an EdgeNotification message to the server when it receives a packet with an unknown destination address. Upon receiving the message, the server selects the appropriate paths according to a path selection algorithm and deploys the

paths via BestEffortTableEntry messages to the edge router and FlowRoutingTableEntry messages to other routers on the paths.

The SAAM path selection algorithm selects two paths for each ingress-egress node pair.[1] A Shortest Widest Path is selected as the primary path for performance reasons. A Shortest Widest Most Disjoint Path is selected as the secondary path to achieve load balancing and fault tolerance. Although currently two paths are selected, the implementation is flexible enough to allow the selection of any number of paths, including all possible paths between the ingress and egress.

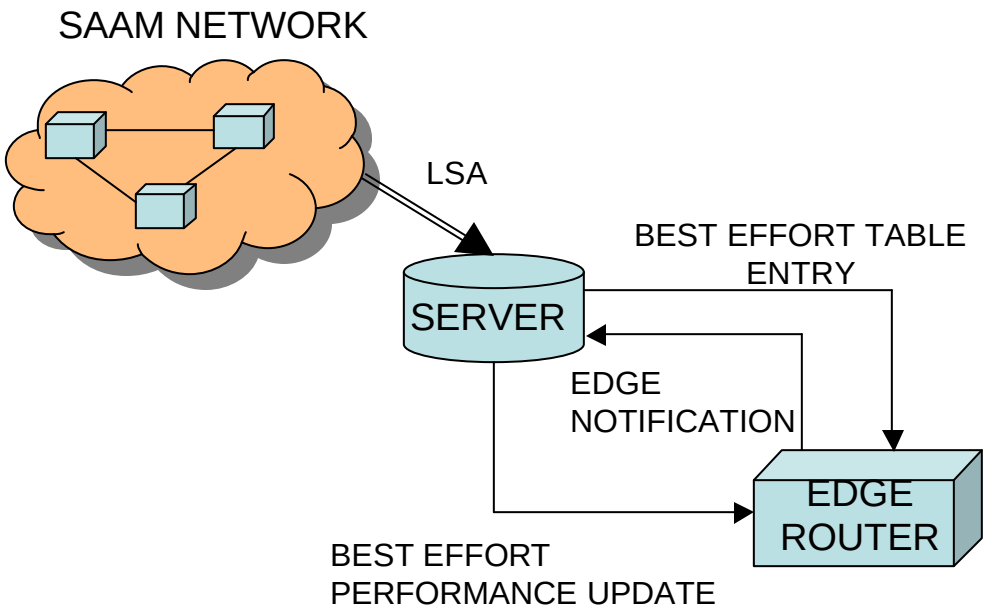


Figure 5. Communication Between Server and Edge Router in MATE.

4. Phase Three (Network Monitoring Phase)

In phase three, the MATE algorithm monitors the congestion measures of each LSP. If an appreciable change in a congestion measure is detected, the algorithm shifts to phase four. The condition that needs to be met to go to phase four is as follows [8]:

*If (|LossRate (i) – LossRate_New (i)| > alpha * |LossRate_New (i) – LossRate_Old (i)|)*

Go to Phase Four (1)

The MATE algorithm calculates the current derivative of a congestion measure of a specific LSP by measuring the difference of the congestion measure at two most recent sampling intervals and dividing this value by the amount of traffic that is shifted to the LSP during the same time. In this work, the congestion measure is packet loss rate. Let the loss rate on LSP (i) at time t1 be *LossRate_Old (i).* Assume at time t1, *n* buckets of traffic are shifted to LSP (i). Let the measured packet loss rate on the same LSP at next sampling time t2 be *LossRate_New (i).* Then the loss rate derivative at time t2 can simply be calculated by this formula *LossRate_New(i) – LossRate_Old(i) / n.*

Recall the condition statement (1) that determines if MATE should go to phase 4, *LossRate_New (i)* and *LossRate_Old (i)* denote the measured loss rate values for calculating the last loss rate derivative during phase one or phase two. *LossRate (i)* refers to the current measured loss rate on the same LSP. Alpha is a threshold value for loss rate change. MATE does not define a value for Alpha. It is left to implementers, but a typical value is on the order of one percent. The statement simply says that if the marginal loss rate change exceeds some threshold value, go to phase four.

The problem here is that the algorithm exits the network monitoring phase (phase 3) upon a very small marginal loss rate change when *LossRate_New (i)* and *LossRate_Old(i)* are equal. This can happen frequently. For example, *LossRate_New (i)* and *LossRate_Old(i)* are often both zeros. The quick exit is undesirable because reacting to small performance variations will cause load fluctuations and degrades the algorithm's stability. To fix this problem, inequality (1) was changed to the following:

If (|LossRate (i) – LossRate_New (i)| > max (alpha, |LossRate_New (i) – LossRate_Old (i)|)
Go to Phase Four (2)

With the new condition, the algorithm exits phase 3 only if the marginal loss rate change is larger than the maximum between *alpha*, a threshold value that is typically much greater than zero, and |*LossRate_New (i) – LossRate_Old (i)*|. Obviously the new condition addresses the quick exit problem. However, another problem may arise. In the case that the value of |*LossRate_New (i) – LossRate_Old (i)*| is too high, the algorithm never exits the network monitoring phase even upon large marginal loss rate changes.

Fortunately, the algorithm's phase one and phase two actions prevent this problem by ensuring *LossRate_New(i)* and *LossRate_Old (i)* to be close to each other.

5. Phase Four (Refreshing State Table)

Below is the pseudo code of the MATE algorithm's phase four operation as described in [8]:

LSP(i) detects a significant change in its marginal loss rate
set Flag(i)
if (LossRate(i) > LossRate_New(i))
    *LossRate_Old(i) <- LossRate(i) - 2 * (LossRate_New (i) - LossRate_Old (i))*
    LossRate_New (i) <- LossRate(i)
    Inc(i) <- 1
else
    LossRate_Old (i) <- LossRate(i) - (LossRate_New (i) - LossRate_Old (i)) / 2
    LossRate_New (i) <- LossRate(i)
    Inc(i) <- 1
For each LSP k such that (k != i)
    LossRate_Old (k) <- LossRate(k) - (LossRate_New (k) - LossRate_Old (k))
    LossRate_New (k) <- LossRate(k)
Go to Phase 2

In this phase, the loss rate derivative of an LSP with appreciable loss rate change is adjusted manually. The derivative is doubled or halved depending on whether the loss rate has increased or decreased. Manually adjusting derivatives in phase 4 is necessary because phase three does not modify loss rate derivatives. It is meaningless to do so since there is no traffic shift in phase 3.

This phase has a similar problem with phase three. When *LossRate_New* and *LossRate_Old* are equal, the loss rate derivative becomes zero. Obviously this is not a desired behavior. To avoid this problem, two changes were made to the above pseudo code.

*LossRate_Old(i) <- LossRate(i) - 2 * (LossRate_New (i) - LossRate_Old (i))*
was replaced with

LossRate_Old(i) <- LossRate(i) – min(LossRate (i) – LossRate_New (i), 2 max (|LossRate_New (i) – LossRate_Old (i)| , alpha))

And,

LossRate_Old (i) <- LossRate(i) - (LossRate_New (i) - LossRate_Old (i)) / 2

was replaced with

LossRate_Old (i) <- LossRate(i) – max(LossRate(i) – LossRate_New(i), -1/2 max (|LossRate_New(i) – LossRate_Old(i)|, alpha))

With those changes, different derivative values are assigned to LSPs, so that phase two can do derivative equalization.

B. SOFTWARE COMPONENTS OF SAAM'S MATE IMPLEMENTATION

In order to integrate the MATE algorithm, some modifications and additions were made to the existing Java based SAAM code.

1. Modifications to Existing Code

The MATE implementation performs many similar functions like the *BestEffortTable* module in SAAM. The two classes were put under a common interface. *BestEffortTable* was changed to be an interface and all the code of BestEffortTable was moved to a new class named *SAAMTable*. On the other hand, the MATE algorithm was implemented as the *MATETable* class. In this new scheme, *SAAMTable* provides the functionality of SAAM's best effort traffic management while *MATETable* provides the MATE algorithm functionality. Those two agent classes both implement the *BestEffortTable* interface as shown in the Figure 6 below.

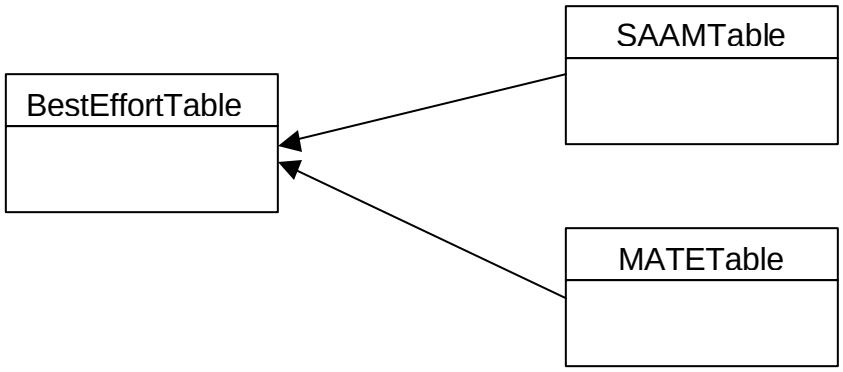


Figure 6. Class structure of SAAMTable and MATETable

A new element was added to SAAM XML configuration file syntax in order to allow a user to choose a best effort traffic manager between *SAAMTable* and *MATETable*. The new element is named *BestEffortAgentType* and placed under the *node* element. This new element can be assigned two values: "SAAMTable" or "MATETable". According to the value of the *BestEffortAgentType*; the *SAAMTable* agent or the *MATETable* agent is installed to routers when the network is set up. Corresponding modifications were also made to the SAAM_Net DTD file and the *SAXNetworkInfoParser* class.

a. Change to EdgeNotification Message Format

A new field was added to an *EdgeNotification* message. This field holds a short integer which carries the Best Effort agent type: *SAAMTable* or *MATETable*. This is used by edge routers to notify the server about the type of Best Effort agent they are using.

b. Changes to BestEffortManager Class

New methods were added to the *BestEffortManager* class to provide the server side functionality of the MATE algorithm.

(1) private processMATEEdgeNotification ()

This method is called by processEdgeNotification method in the BasePIB class when an EdgeNotification message is received from a router running the MATETable agent. This method does the necessary initiations on the server side.

(2) private updateMATEBETopology ()

This method is called by the processMATEEdgeNotification method. It updates the topology and deploys necessary paths (LSPs) to routers.

(3) private sendTableEntries ()

This is an overloaded method of the original sendTableEntries method. It is called by the updateMATEBETopology to send the MATETable entries to the routers.

(4) private startBestEffortPerformanceUpdate ()

This method is called by processMATEEdgeNotification () when the first Edge Notification message is received. It starts a timer which invokes the updateBestEffortPerformance () periodically.

(5) private updateBestEffortPerformance ()

This method computes the loss rate values on the deployed paths and sends those values to routers via BestEffortPerformanceUpdate messages. The method is invoked periodically using a timer.

c. Changes to BasePIB Class

A new method, called *findMATEPaths*, was added to the *BasePIB* class. This method is called by the *updateBETopology* method of the *BestEffortManager* class. It finds paths for node pairs, using existing routing algorithms. The current implementation finds two paths per node pair and it uses the Shortest Widest Path and the Shortest Widest Most Disjoint Path algorithms to select those paths.

2. Addition of New Components

a. MATETable Class

A new *MATETable* class is created under the *saam.agent.router* package. It implements the *BestEffortTable* interface. It provides the functionality of the MATE algorithm on the router side.

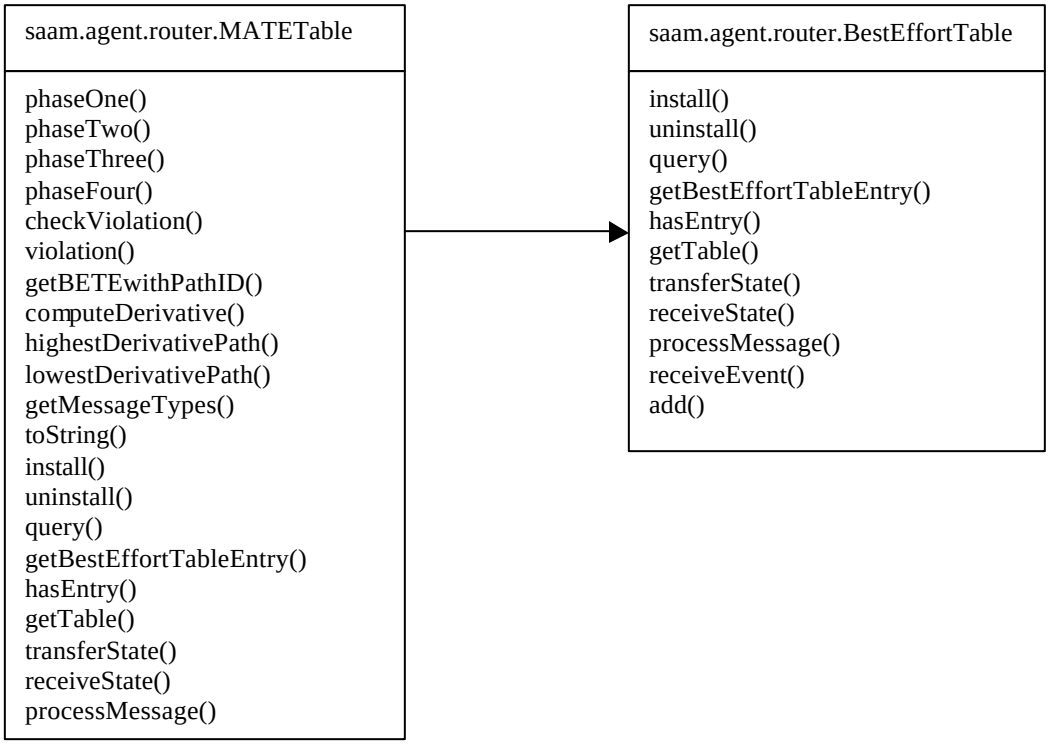


Figure 7. MATETable Class Structure

b. BestEffortPerformanceUpdate Class

A new *BestEffortPerformanceUpdate* class is created under the *saam.message* package. This class defines the format of BestEffortPerformanceUpdate messages and provides methods for packing and unpacking such messages. The SAAM server uses BestEffortPerformance messages to report the loss rate values of deployed paths to edge routers.

The BestEffortPerformanceUpdate class composes and manipulates a message that carries loss rate values of all paths that are installed on a specific router. The Path ID and the loss rate of each path is kept in an inner class called PathPacketLossRate (PPLR). Some methods are also included in the BestEffortPerformanceUpdate class, in order to append a PathPacketLossRate object to or extract one from a BestEffortPerformanceUpdate messages.

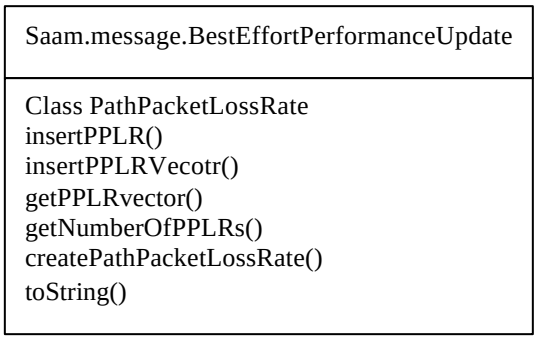


Figure 8. BestEffortPerformanceUpdate Class Structure

V. TESTS AND RESULTS

Tests were conducted in order to validate the MATE implementation in SAAM and to compare the performance of SAAM's Best Effort management scheme against the MATE algorithm. The tests were performed with custom test topologies, written in XML files [XML SAX Parser], and the flow generator and sink agents developed in [9].

A. TESTS WITH MATE TOPOLOGY

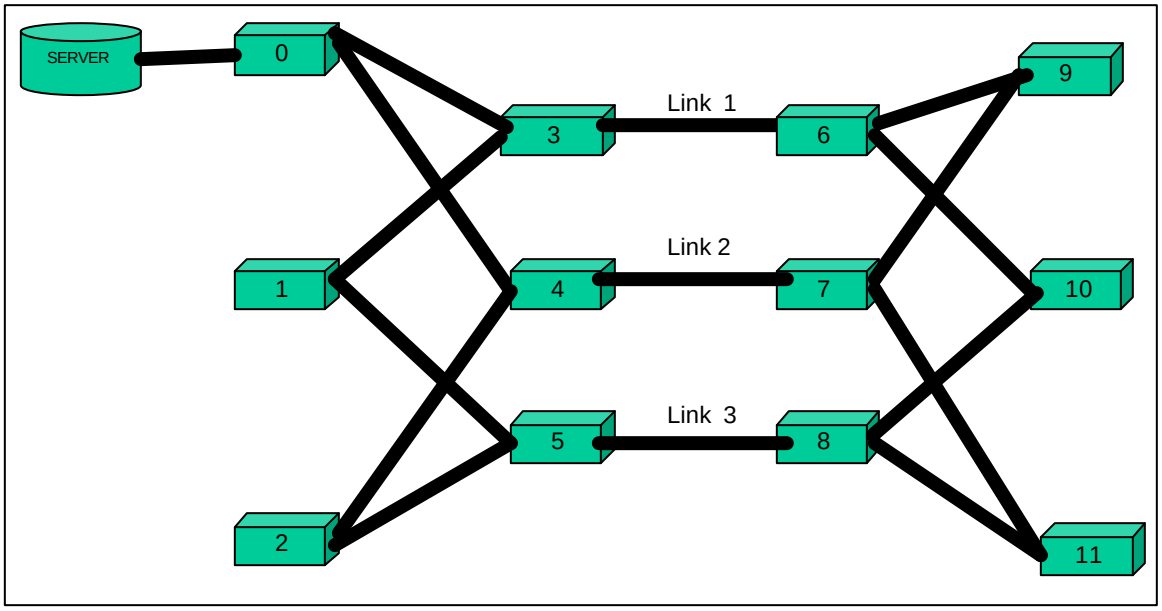


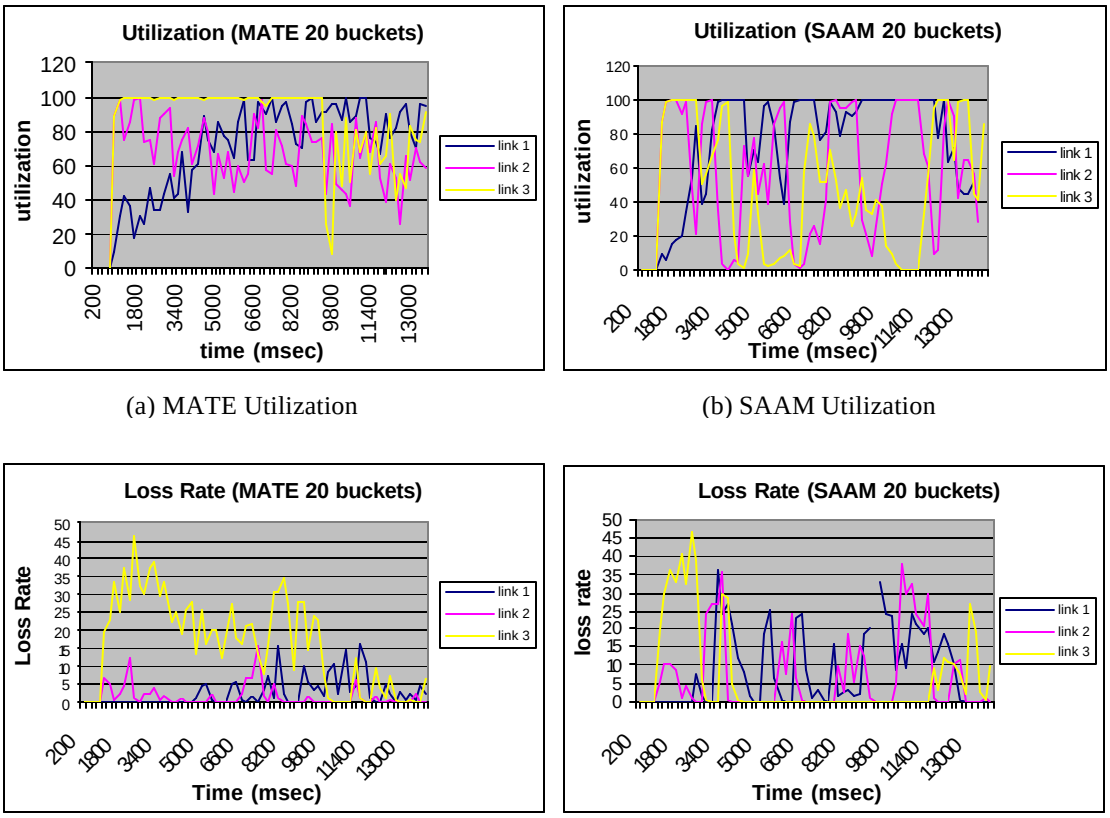
Figure 9. The MATE Topology

The first test topology, called MATE topology, is illustrated in Figure 9. This topology was also used by [3] and [8] to test different traffic engineering solutions. The experiment parameters used in [3] and [8] were described in [8]. Initially, these original experiment parameters were used for this work. However, because of the heavy packet traffic, the memory usage of the SAAM simulation program quickly exceeded the host machine's capacity and the program would hang. Therefore, to reduce the number of packets, the packet length is increased four fold while the packet generation rate is decreased four fold. The bit rate remains the same. Unfortunately, this change did not sufficiently improve the memory performance. Consequently, the capacity of each link

and the bit rate of all input traffic were scaled down by a factor of 10. To further reduce the memory usage, a change was made to the path deployment scheme, which was developed as part of SAAM's BE solution and later adapted to work with the MATE algorithm. The scheme was changed to only deploy the necessary paths for active BE source destination pairs. Previously, BE paths were deployed for all BE source destination pairs, regardless of whether or not they had traffic. With these changes, the memory usage was reduced to a reasonable level for the simulation to run properly.

After the changes were made, the resulting parameters are as follows. The nodes labeled 0, 1 and 2 are ingress nodes. The corresponding egress nodes are labeled 9, 10 and 11, respectively. Each ingress node has two paths (LSPs) to reach the egress. For example, the two paths for ingress node 0 are: 0-3-6-9 and 0-4-7-9. All links in the topology have a capacity of 4.5 Mbps. In [3] and [8] the simulation duration was 7200 seconds. However, the memory limitation discussed above did not allow this simulation to run that long. Therefore, the simulation duration was scaled down to 36 seconds. The test results indicate that this simulation duration was enough for the tested algorithms to converge.

There are 32 Poisson sources sending traffic into each ingress node. These sources were implemented by one Poisson flow generator as designed in [9], with an average packet size of 1008 bytes and average packet rate of 246.1 packets/second. Poisson cross-traffic was introduced at each of the three intermediate nodes 3, 4 and 5 and exited at nodes 6, 7 and 8, respectively. The cross traffic pattern varied over time. Node 3 generated cross traffic with 15 Poisson sources for the entire experiment. Each Poisson source generated 1008-byte packets at a rate of 12.15 packets per second. The number of sources in Node 4 increased from 15 to 30 at the eighteenth second of the simulation. Finally, for Node 5, the number of sources decreased from 35 to 20 at the ninth second of the simulation.



(a) MATE Utilization

(b) SAAM Utilization

(c) MATE Loss Rate

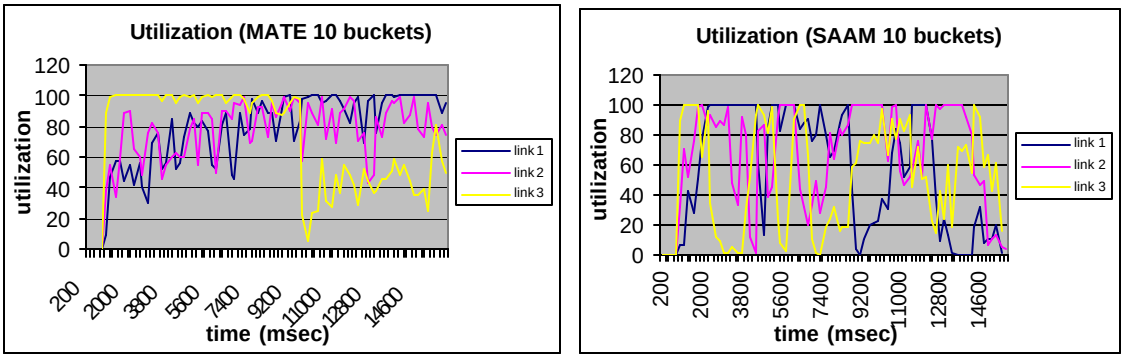
(d) SAAM Loss Rate

Figure 10. Comparison of MATE vs. SAAM's BE Solution

Each test set consisted of two runs using, respectively, the MATE and SAAM algorithms. The number of buckets used for traffic splitting was set to 20 for both algorithms. Although the simulation duration was set to 36 seconds, the memory limitation still prevented the simulation from finishing. The program was locked at different times in different runs using different algorithms. The data collected from some runs was truncated so that a uniform time reference could be established for comparing the performance of the algorithms.

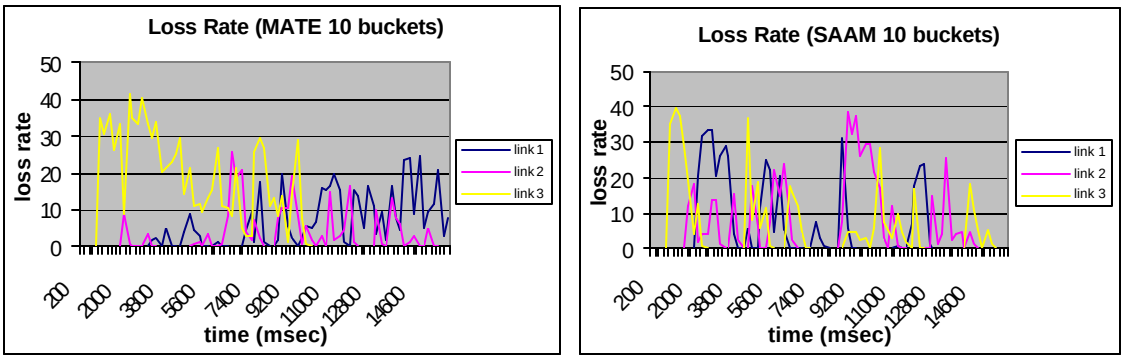
The results from one set of tests using the MATE topology are plotted in Figure 10. It shows the utilizations and loss rates of links 1, 2 and 3 under the MATE and SAAM algorithms. The results indicate the MATE algorithm was converging by distributing the load among the two possible paths. In contrast, the SAAM algorithm is shown to have caused a lot of fluctuation in performance to the links. To see the effect of

the number of buckets on the results, the same experiment was repeated with the number of buckets set to 10 instead of 20. The results are plotted in Figure 11.



(a) MATE Utilization

(b) SAAM Utilization



(c) MATE Loss Rate

(d) SAAM Loss Rate

Figure 11. Comparison of MATE and SAAM's BE Solution Using 10 Buckets

The performance of the MATE algorithm was lower with 10 buckets. However, the performance of the SAAM algorithm did not have any significant change with 10 buckets, still causing performance fluctuation on the links.

A. TESTS WITH EXPANDED MATE TOPOLOGY

From the early test results, it was clear that SAAM had worse performance than MATE on the MATE topology. On some routers, the SAAMTable contained three duplicate primary path entries. This is a bug because those routers followed the SAAM algorithm and dynamically balanced the load between two of the three seemingly

different primary paths, actually concentrating 100 percent of the traffic on one physical path. This bug would certainly lower the performance of SAAM.

Furthermore, the MATE topology limited each ingress router to two paths to the destination. As a result, SAAM could not utilize one of its important features, dynamic path deployment. To see the effect of SAAM's dynamic path deployment feature, a new topology was created. The new topology (Figure 12) was an expanded version of the MATE Topology. It had two new nodes, 12 and 13, to allow additional paths to be deployed for an ingress router when there is congestion in both of the original paths.

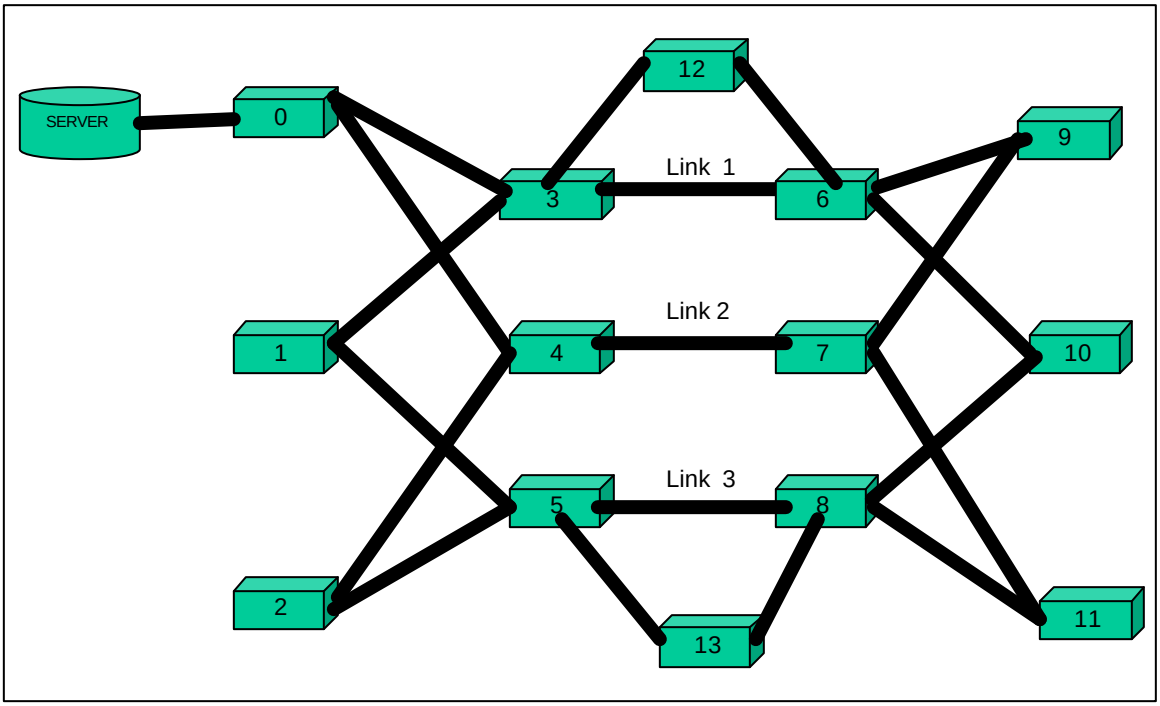
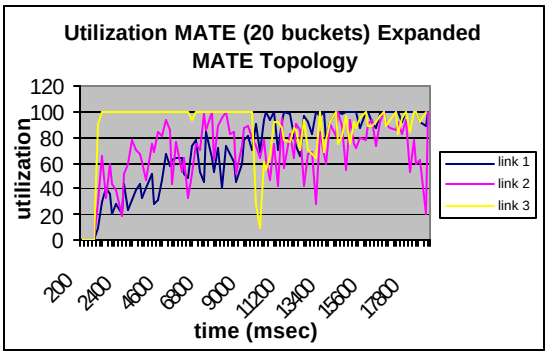
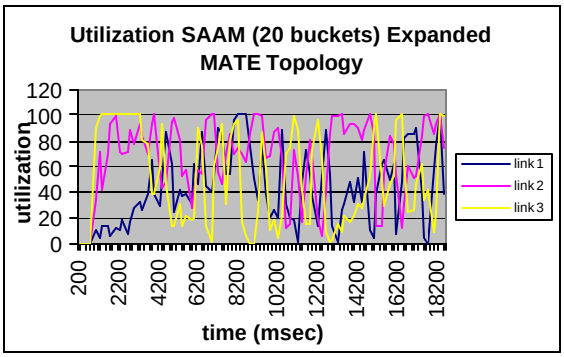


Figure 12. Expanded MATE Topology

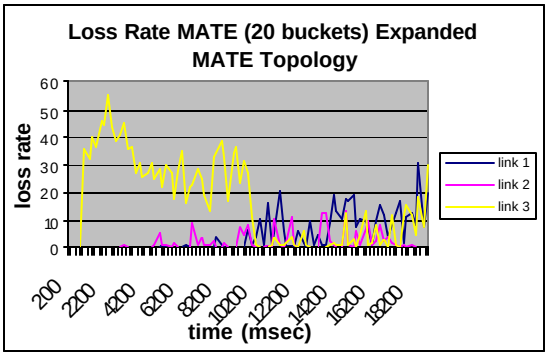
The results from a set of tests using the expanded MATE topology are shown in Figure 13. Since MATE does not support deploying paths on the fly, its performance remained the same. Contrary to the expectation that SAAM would perform better with the expanded topology, there was not any significant performance improvement for SAAM.



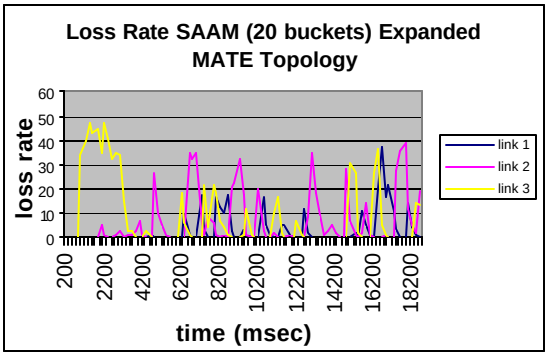
(a) MATE Utilization



(b) SAAM Utilization



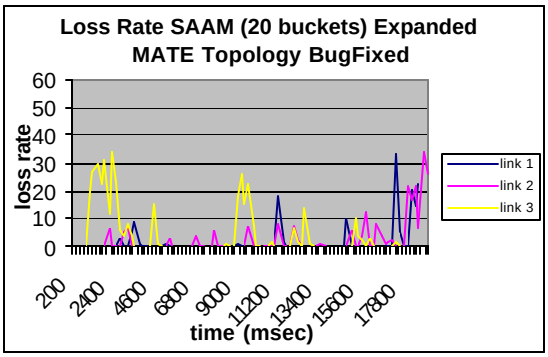
(c) MATE Loss Rate



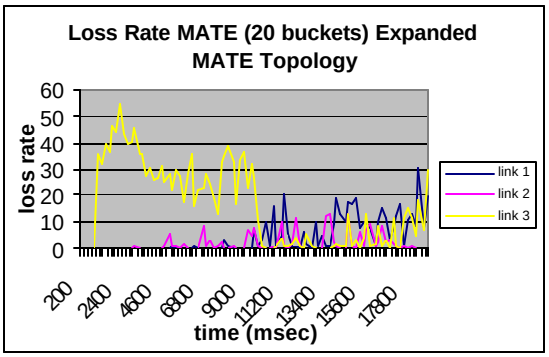
(d) SAAM Loss Rate

Figure 13. Simulation Results with Expanded MATE Topology

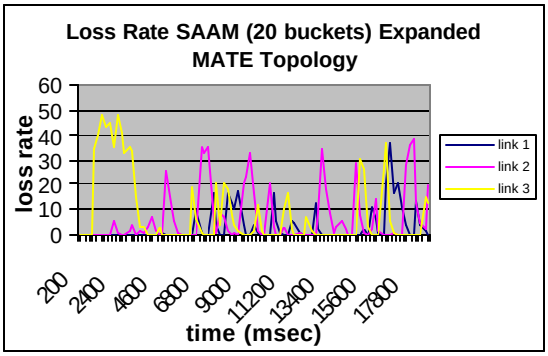
The results showed that SAAM might not perform as expected, again pointing to a bug in the software. To find the bug in the SAAM code, a detailed investigation of the log files was performed. The BestEffortManager log files revealed that SAAM's Switchback mechanism did not have the intended behavior. Switchback is used when the server wants to shift all traffic of an ingress-egress pair back to the primary path. In order to do that, the server deploys the primary and the alternate path again to the ingress router. However, the current implementation was deploying the primary path two times. Consequently, the SAAMTable has two duplicate primary paths that are active, which is a bug because the duplicate paths nullify the effect of load balancing.. This bug was fixed and the simulation was rerun. This time the results (Figure ?) showed that SAAM utilized the redundant paths and managed to keep the loss rates of the links at reasonable levels, most of the time.



(a) SAAM Loss Rate (after bug fix)



(b) MATE Loss Rate



(c) SAAM Loss Rate (with bug)

Figure 14. Comparison of the SAAM's BE Solution and the MATEAlgorithm After Bug Fix

THIS PAGE INTENTIONALLY LEFT BLANK

VI. MODIFICATIONS TO SAAM CODE TO IMPROVE PERFORMANCE OF SAAM'S BE SOLUTION

A. BEST EFFORT TOPOLOGY MAINTENANCE

The behavior of the BestEffortManager in the current SAAM implementation, when an EdgeNotification message arrives, is shown in Figure 15. In this scheme, when an EdgeNotification message arrives to the server, BestEffortManager initially resets all the BE paths. Then it deploys all paths between all BE routers. In other words, each time a new edge (BE) router is discovered, two paths between every BE router pair are deployed. Obviously, this is not necessary. A new discovery does not affect the paths already deployed. Therefore, this approach unnecessarily increases the BE path deployment time, resulting in unnecessary packet losses at each edge router due to the BE table not having the right paths.

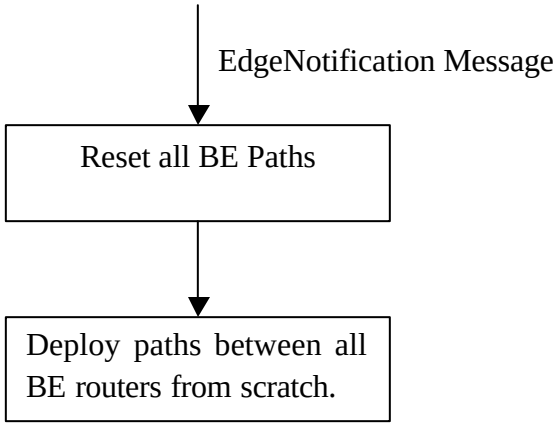


Figure 15. Behavior of the BestEffortManager in Current SAAM when an EdgeNotification Message Arrives

This approach also causes another undesirable behavior. Deploying paths that were already deployed will trigger a Switchback procedure, in which the BE routers receiving the duplicated paths will switch all traffic back to the primary path. For those routers in the middle of a Redirection procedure, this Switchback process shifts traffic back to a congested primary path.

To address these problems, the path deployment mechanism of the BestEffortManager was changed. In this new mechanism, only the paths for the newly discovered edge router were deployed. Not redeploying the existing paths accelerated the forming of the BE topology and avoided unintended Switchbacks.

B. SWITCHBACK

Switchback is a procedure initiated by the server to switch all traffic of an ingress-egress pair back to the primary path. The server initiates this procedure in two cases: congestion is discovered on the alternate path while the primary is congestion-free, or congestion on the primary path clears while the alternate path remains congested. Switchback involves redeploying the primary and the alternate paths to the ingress node. Upon receiving the duplicated paths, the ingress node sets the traffic split so that the primary path for the specified egress router receives 100% of the traffic.

Recall from Chapter V, it was discovered that the Switchback procedure was not implemented correctly. Instead of redeploying the primary and the alternate paths, Switchback was redeploying the primary path twice. This simple bug caused a significant decrease in performance. The bug was subsequently fixed.

VII. CONCLUSIONS AND FUTURE WORK

A. CONCLUSIONS

In this thesis, the performance of SAAM's BE management solution was evaluated by comparing it with the MATE algorithm presented in [3]. Much of the effort was spent on implementing the MATE algorithm in SAAM. One major difficulty while implementing the MATE algorithm was translating the MATE features into the SAAM compatible code. The implementation needs to provide the same effect as the original MATE algorithm and also it needs to work well with the SAAM code. Meeting these two conditions turned out to be more difficult than anticipated.

The topology used in the original MATE paper [3] was selected as the test topology. This selection should have made it possible to validate the MATE implementation in this thesis by comparing the simulation results with the results reported in [3]. However, the extensive memory usage of the simulation program (due to Java's memory handling) did not allow simulation durations to be sufficiently long for a meaningful quantitative comparison. Therefore, the validation of the MATE algorithm was done qualitatively rather than quantitatively.

The durations of the simulations were too short to support a thorough comparison of the two BE management schemes. However, enough data was collected for a general comparison. On one hand, the initial results indicate that given a fixed set of paths, SAAM may not perform as well as MATE. On the other hand, using SAAM's dynamic path deployment functionality allows the load to be distributed across more parts of the network, thus achieving better performance than MATE.

Besides evaluating SAAM's BE management solution, this thesis also provided a stress test for the SAAM test-bed. Until this study, SAAM's test-bed had not been tested with as significant a packet load. Under heavy packet loading SAAM's test-bed had some memory problems. It was concluded that the main source of the memory problem was the Java programming language.

B. FUTURE WORK

Depending on the topology that is used, the performance of the BE management scheme may vary. For example, the dynamic path deployment feature of SAAM's BE management would provide better results on a network with more alternate paths. The test topologies, which were used in this thesis, were based on the MATE Topology introduced in [3]. To evaluate the SAAM's BE solution completely, experiments with different network topologies should be performed.

SAAM's BE solution provides fairness and fault tolerance, as well as congestion control. While evaluating the loss rate performance, this study did not evaluate the fairness and the fault tolerance feature of SAAM's BE management solution. An evaluation of those features was left as a future work.

SAAM's BE solution has some adjustable parameters that can change the performance of the scheme. These parameters are: *Number of Buckets, Redirection Interval, Reversion Interval, Congestion Bypass Time, Path Expiration Interval* and *Local Resolution Timeout*. In this thesis, the original design values were used.[1] Experimenting with different parameter value sets or a parametric analysis is necessary to determine the optimal values. This is left for future work also.

APPENDIX A: MATETABLE SOURCE CODE

```
//26Mar03[Ayvat]  - created


package org.saamnet.saam.agent.router;

import java.net.InetAddress;
import java.net.UnknownHostException;
import java.util.*;
import java.lang.Math;

import org.saamnet.channel.*;
import org.saamnet.saam.agent.*;
import org.saamnet.saam.control.*;
import org.saamnet.saam.event.*;
import org.saamnet.saam.gui.*;
import org.saamnet.saam.message.*;
import org.saamnet.saam.net.*;
import org.saamnet.saam.router.*;

/**
 * Routes BE traffic according to MPLS Adaptive Traffic Engineering (MATE) algorithm
 * It is a lookup table the router uses to associate
 * unlabeled BE traffic destined for a particular address to a path
 * that is installed in the FlowRoutingTable.
 * <p>Copyright: Copyright (c) 2003</p>
 * <p>Company: Naval Postgraduate School</p>
 * @author B. Ayvat
 * @version 1.0
 */
public class MATETable extends Hashtable implements BestEffortTable
{
  /**
   * the maximum number of routes to split to a single destination
   */
  public final static int MAX_ROUTES = 2;

  private TableGui gui;

  private Vector columnLabels = new Vector();

  private ControlExecutive controlExec;

  private byte[] myMessages =
  {
     Message.BEST_EFFORT_TBL_ENTRY,
     Message.BEST_EFFORT_PERFORMANCE_UPDATE,
  };

  /**
   * hashtable of <code>TrafficDestination</code> objects keyed by destination IP address
   */
  private Hashtable destinationList = new Hashtable();

  private int timeScale;

  private ComponentPanel panel;

  //index values for the array in the paths hashtable of the <code>TrafficDestination<code>
  public final static int LOSSRATE_OLD = 0;
  public final static int LOSSRATE_NEW = 1;
  public final static int INCREMENT = 2;
  public final static int DERIVATIVE = 3;
  public final static int BACKOFF_BEPU_COUNT = 4;
```

```
//Four phases of MATE Algorithm.
public final static int PHASE_ONE = 1;
public final static int PHASE_TWO = 2;
public final static int PHASE_THREE = 3;
public final static int PHASE_FOUR = 4;

//If a ,violation of monotonous property of loss rate versus load detected,
//while shifting traffic, algorithm enters violation phase until violation removes.
public final static int VIOLATION_REMOVAL = 5;

//number of buckets to shift from lowest derivative path to highest derivative path
//at one time when equilizing derivatives during phase two.
public final static int NOOFBUCKETS_TOSHIFT_ATONCE = 1;

//threshold on the load change. it is used in phase three and phase four.
public final static int LOAD_CHANGE_THRESHOLD = 100;

//weighting constant to calculate LossRate(i) value in phase three
//measured loss rate is not used directly. instead it is updated by a low pass filter.
public final static float WEIGHTING_CONSTANT = 0.2f;

public final static float OVERSHOOT_THRESHOLD = 100.0f;

//Those are used as arguments when violation() method is called. It tells the method;
//what kind of violation check to conduct.
public final static byte INCREASE_VIOLATION = 0;
public final static byte DECREASE_VIOLATION = 1;


/**
 * A data structure used by the <code>MATETable</code> to
 * track information on a per-destination basis.  Notably, it holds one key
 * key hash table.  Hashtable pathLossRates holds loss rate and derivative
 * information about the paths installed for this destination. It is keyed
 * by the path ID.
 */
public static class TrafficDestination
{
    public IPv6Address destination;
    public int noOfPaths; // Number of paths deployed for this destination.
    public int phase;     // Determines the phase in which this destination is.
    public int indexOfLSPtoShiftTraffic; //In Phase One,
    public Integer highestDerivativePath = null;
    public Integer lowestDerivativePath = null;

    //Contains loss rate and derivative information about paths. Keyed by path ID.
    public Hashtable pathLossRates = new Hashtable();

    //Phase that called the checkViolation() method
    public int checkViolationCallerPhase;

    //Counter to count how many times checkViolation() method is visited.
    public int checkViolationCounter;

                //Stores the pathIDs of the paths that traffic is shifted to and shifted from.
                //Those are used in checkViolation() method.
                public Integer shiftToPath;
                public Integer shiftFromPath;

                //This is set to true when checkViolation() method computes the derivatives, so that,
                //it is not computed again, when other phases called.
                public boolean alreadyComputedDerivatives;

                //Determines if compensation mechanism in phase two is run or not.
                public boolean compensated;

                public int violationShiftIncrement;
```

```
  public TrafficDestination(IPv6Address address)
  {
    destination = address;
    noOfPaths = 0;
    phase = PHASE_ONE;
    indexOfLSPtoShiftTraffic = 1;
    checkViolationCounter = 0;
    alreadyComputedDerivatives = false;
    compensated = false;
  }
} //end of inner static class Traffic Destination


/**
 * Required method of the <code>ResidentAgent</code> interface.
 *
 * @param controlExec The <code>ControlExecutive</code> on the router this agent
 * is being installed on.
 * @param instanceName The name of this agent instance
 * @param parameters The array of parameter strings for this agent
 */
public void install(ControlExecutive controlExec,
                    String instanceName,
                    String [] parameters)
{
  columnLabels.add("Dest IPv6 Address");
  columnLabels.add("Map to Path");
  columnLabels.add("Traffic Split");

  RouterGui routerGui = controlExec.getRouterGui();
  gui = new TableGui("", columnLabels);
  /*ComponentPanel*/ panel = new ComponentPanel(instanceName, routerGui, gui);
  routerGui.addComponentPanel(panel, RouterGui.ROUTING_TABLES_MENU);
  this.controlExec = controlExec;
  controlExec.registerMessageProcessor(myMessages, this);
  timeScale = controlExec.getTimeScale();

                    controlExec.getRoutingAlgorithm().setBestEffortRoutingTable(this);
                    controlExec.getRoutingAlgorithm().bestEffortAgentType = controlExec.MATE_IS_ACTIVE;
} //end of install()


/**
 * Required uninstall method of the ResidentAgent interface.
 */
public void uninstall()
{
  clear();
}


/**
 * The communication method through which <code>ResidentAgent</code> talks.
 * @param   message  a BETE with only the destination field
 * @return  a BETE with a path ID filled in to map to
 */
public Message query (Message message)
{
  IPv6Address destAddr = ((BestEffortTableEntry) message).getDestAddr();
  int bucketMap = ((BestEffortTableEntry) message).getSplit();
  BestEffortTableEntry result = getBestEffortTableEntry(destAddr, bucketMap);
  return result;
} //end of query()
```

```
/**
 * Retrieves the BET entry for a destination address and bucket map.
 * @param   destAddr
 * @param   bucketMap
 * @return  the associated BETE
 */
public BestEffortTableEntry getBestEffortTableEntry(IPv6Address destAddr, int bucketMap)
{
 TrafficDestination trafDest = (TrafficDestination) destinationList.get(destAddr.toString());

  if (trafDest != null) //may be no such entry yet; see RoutingAlogrithm
  {
    int serialNo = 0;
    int split = 0;//0%
    int percentile = (bucketMap + 1) * PERCENT_PER_BUCKET;

    for (int counter = 0; counter < trafDest.noOfPaths; counter++)
    {
      split += ((BestEffortTableEntry)get (destAddr.toString() + counter)).getSplit();
      if (percentile <= split)
      {
        serialNo = (serialNo + counter) ;
        break;
      }
    }

    String key = destAddr.toString() + serialNo;
    BestEffortTableEntry result = (BestEffortTableEntry) get(key);
    return result;//expect null if no entry; see RoutingAlgorithm
  }
  else
  {
   return null;
  }
} //end of getBestEffortTableEntry()


/**
 * Returns true if t he BET contains an entry indexed by destination
 * address andfalse otherwise.
 * @param destAddr A particular destination IP address
 * @return true if the BET contains an entry indexed by the destination address, false otherwise
 */
public boolean hasEntry(IPv6Address destAddr)
{
 if (destinationList.get(destAddr.toString()) != null)
  {
   return true;
  }
 else
  {
   return false;
  }
} //end of hasEntry()

/**
 * Returns the entire contents of this BestEffortTable or null
 * if this BestEffortTable is empty.
 * @return A Vector of all entries currently in the flow routing table.
 */
public Vector getTable()
{
 if (isEmpty())
  {
   return null;
  }

  Vector table = new Vector(size());
```

```
  Enumeration e = elements();
  while (e.hasMoreElements())
  {
   Vector oneRow = new Vector();
   BestEffortTableEntry betentry = (BestEffortTableEntry) e.nextElement();

   oneRow.add("" + betentry.getDestAddr());
   oneRow.add("" + betentry.getPathMap());
   oneRow.add("" + betentry.getSplit());

   table.add(oneRow);
  }

  return table;
 } //end of getTable()


 /**
  * Required method for <code>ResidentAgent</code> interface.
  * @param   replacement the <code>ResidentAgent</code> replacement
  */
 public void transferState(ResidentAgent replacement)
 {
  for (Enumeration e = elements(); e.hasMoreElements();)
  {
   replacement.receiveState((BestEffortTableEntry) e.nextElement());
  }
 } //end of transferState()


 /**
  * Required method for <code>ResidentAgent</code> interface.
  * @param   message a BETE (one at a time from <code>transferState</code> method)
  */
 public void receiveState (Message message)
 {
  add((BestEffortTableEntry) message);
 }


 /**
  * <code>BestEffortTable</code> process two types of messages, BEST_EFFORT_TBL_ENTRY and
  * BEST_EFFORT_PERFORMANCE_UPDATE. For BEST_EFFORT_TBL_ENTRY, it adds the entry and makes a
* new <code>TrafficDestination</code> if it does not have this destination on file. For
* BEST_EFFORT_PERFORMANCE_UPDATE It updates loss rate values of the paths.
  * @param   message
  */
 public void processMessage (Message message)
 {
  switch (message.getBytes()[0])
  {
   case Message.BEST_EFFORT_TBL_ENTRY:

    BestEffortTableEntry betentry = null; //-crcp
    try  //-crcp
    {
     betentry = new BestEffortTableEntry(message.getBytes()); //-crcp generic way
    }
    catch(UnknownHostException uhe)
    {
     panel.appendMessage("BestEffortTable Error: can't create local BETE." + uhe);
    }

    panel.appendMessage("New BETE message: Path " + betentry.getPathMap() +" through " +
betentry.getDest Addr().toString());
    //check to see if this is a known destination
    if (destinationList.containsKey(betentry.getDestAddr().toString()))
    {
```

```
   TrafficDestination trafDest = (TrafficDestination) destinationList.get(betentry.getDestAddr().toString());

   add(betentry);
   ++trafDest.noOfPaths;

  float [] lossRates = new float[] {0, 0, 0, 0, -1};
  Integer pathID = new Integer (betentry.getPathMap());

  trafDest.pathLossRates.put(pathID, lossRates);

 }
 else //need to start tracking this new destination
 {
  TrafficDestination trafDest = new TrafficDestination(betentry.getDestAddr());

  ++trafDest.noOfPaths;

  Integer thisPathID = new Integer (betentry.getPathMap());
  float [] lossRates = new float[] {/*firstPathLossRate*/0, 0, 100.0f, 0, -1};
  trafDest.pathLossRates.put(thisPathID, lossRates);

  destinationList.put(betentry.getDestAddr().toString(), trafDest);
  betentry.split = 100;
  add(betentry);
 }
 //this is the server's way of granting edge router permission
 controlExec.acceptEdgeTraffic();
 break;

case Message.BEST_EFFORT_PERFORMANCE_UPDATE:

        BestEffortPerformanceUpdate bepu = new BestEffortPerformanceUpdate (message.getBytes());
        panel.appendMessage("Following BEP update is received: ");
        Enumeration enum = bepu.getPPLRvector().elements();
        while (enum.hasMoreElements())
        {
          BestEffortPerformanceUpdate.PathPacketLossRate pplr =
            (BestEffortPerformanceUpdate.PathPacketLossRate)enum.nextElement();
          panel.appendMessage(pplr.toString());
        }

        Vector pplrVector = bepu.getPPLRvector();

        Enumeration myEnum = destinationList.elements();
        while (myEnum.hasMoreElements())
        {
          TrafficDestination trafficDest = (TrafficDestination)myEnum.nextElement();
          int phaseNo = trafficDest.phase;

          switch (phaseNo)
          {
             case PHASE_ONE:

                        phaseOne(pplrVector,trafficDest);
                        break;

                      case PHASE_TWO:

                        phaseTwo(pplrVector,trafficDest);
                        break;

                      case PHASE_THREE:
                        phaseThree(pplrVector,trafficDest);
                        break;

                      case PHASE_FOUR:

                        phaseFour(pplrVector,trafficDest, new Integer (0), 0);
```

```
                        break;

                                case VIOLATION_REMOVAL:

                        checkViolation (pplrVector, trafficDest, null, null);
                        break;


                       default:
                        break;
                     }//end of switch
              }//end while

              break;

    default:
            break;
   }//end of outer switch

} //end of processMessage()


/**
 * Implements the phase one of the MATE algorithm. Initializes the loss rate
 * derivatives on the paths
 * @param   pplrVector the vector that contains path packet loss rate information
 */
private void phaseOne(Vector pplrVector, TrafficDestination trafficDest)
{
  trafficDest.phase = PHASE_ONE;
            int increment = (int) Math.floor(NUM_OF_BUCKETS / trafficDest.noOfPaths) * PERCENT_PER_BUCKET;

            boolean violation;

            Integer firstPathID = new Integer (((BestEffortTableEntry)get(trafficDest.destination.toString() +
                0)).getPathMap());

            if (trafficDest.noOfPaths >= (trafficDest.indexOfLSPtoShiftTraffic + 1))
            {
              int previousSplit = ((BestEffortTableEntry)get(trafficDest.destination.toString() + 0)).getSplit();
              ((BestEffortTableEntry)get(trafficDest.destination.toString() + 0)).setSplit(previousSplit - increment) ;
((BestEffortTableEntry)get(trafficDest.destination.toString() +
            trafficDest.indexOfLSPtoShiftTraffic)).setSplit(increment) ;

              gui.fillTable(getTable());

              Integer lastPathID = new Integer (((BestEffortTableEntry)get(trafficDest.destination.toString() +
                 (trafficDest.indexOfLSPtoShiftTraffic - 1))).getPathMap());
              Integer thisPathID = new Integer (((BestEffortTableEntry)get(trafficDest.destination.toString() +
                 trafficDest.indexOfLSPtoShiftTraffic)).getPathMap());
             ((float [])(trafficDest.pathLossRates.get(firstPathID)))[INCREMENT] = increment;
             ((float [])(trafficDest.pathLossRates.get(thisPathID)))[INCREMENT] = increment;

             if (!trafficDest.alreadyComputedDerivatives)
             {
               computeDerivative(pplrVect or, trafficDest, lastPathID);
               computeDerivative(pplrVector, trafficDest, firstPathID);

                        //It is possible to see increase in loss rate on a path while we split traffic from this path,
              //or decrease in loss rate while we split traffic to this path. This is "violation". checkViolation
              //method,detects violation, if it finds violation; returns true and shifts traffic back and forth
              //until violation removes. If it does not find; returns false.
               if (trafficDest.indexOfLSPtoShiftTraffic != 1)//No need to check for violation at first round
               {
                violation = checkViolation(pplrVector, trafficDest, lastPathID, firstPathID);
                if (violation)
                {
                   return;
```

```
            }
           }
          }
          trafficDest.alreadyComputedDerivatives = false;
          computeDerivative(pplrVector, trafficDest, thisPathID);

          ++trafficDest.indexOfLSPtoShiftTraffic;
         }
         else
         {
          Integer lastIndexPathID = new Integer (((BestEffortTableEntry)get(trafficDest.destination.toString() +
                    (trafficDest.noOfPaths - 1))).getPathMap());

          if (!trafficDest.alreadyComputedDerivatives)
          {
           computeDerivative(pplrVector, trafficDest, lastIndexPathID);
           computeDerivative(pplrVector, trafficDest, firstPathID);

           violation = checkViolation(pplrVector, trafficDest, lastIndexPathID, firstPathID);
           if (violation)
           {
            return;
           }
          }
          trafficDest.alreadyComputedDerivatives = false;

          Enumeration enum = trafficDest.pathLossRates.elements();
          Enumeration keyEnum = trafficDest.pathLossRates.keys();
          panel.appendMessage("Path Loss Derivatives for destination " + trafficDest.destination.toString() + " are");
          for(int pathIndex = 0; pathIndex < trafficDest.noOfPaths; ++pathIndex)
          {
           Integer thisPathID = new Integer (((BestEffortTableEntry)get(trafficDest.destination.toString() +
                    pathIndex)).getPathMap());
           float [] lossRateArray = (float [])trafficDest.pathLossRates.get(thisPathID);
           panel.appendMessage("Derivative on path " + thisPathID.intValue() + " is " + lossRateArray
                    [DERIVATIVE]);
          }

          trafficDest.phase = PHASE_TWO;
          phaseTwo (pplrVector, trafficDest);
         }
}//end phaseOne()


/**
 * Implements the phase two of the MATE algorithm. Tries to equealize the
 * derivatives on the paths for a destination.
 * @param   pplrVector the vector that contains path packet loss rate information
 */
private void phaseTwo(Vector pplrVector, TrafficDestination trafficDest)
{
         trafficDest.phase = PHASE_TWO;
         panel.appendMessage("\nIn phaseTwo()");

         boolean firstRound = (trafficDest.highestDerivativePath == null && trafficDest.lowestDerivativePath == null);
         if (!firstRound)
         {
          if (!trafficDest.alreadyComputedDerivatives)
          {
           Enumeration enum = trafficDest.pathLossRates.keys();
           while (enum.hasMoreElements())
           {
            Integer thisPathID = (Integer)enum.nextElement();
            computeDerivative (pplrVector, trafficDest, thisPathID);
            //computeDerivative (pplrVector, trafficDest, trafficDest.highestDerivativePath);
            //computeDerivative (pplrVector, trafficDest, trafficDest.lowestDerivativePath);
           }
```

```
                //It is possible to see increase in loss rate on a path while we split traffic from this path,
                //or decrease in loss rate while we split traffic to this path. This is "violation". checkViolation
                //method,detects violation, if it finds violation; returns true and shifts traffic back and forth
                //until violation removes. If it does not find; returns false.
                boolean violation = checkViolation (pplrVector, trafficDest, trafficDest.lowestDerivativePath,
                        trafficDest.highestDerivativePath);
                if (violation)
                {
                        return;
                }
              }
              trafficDest.alreadyComputedDerivatives = false;

              panel.appendMessage("L_OLD is " + ((float
            [])trafficDest.pathLossRates.get(trafficDest.highestDerivativePath))[LOSSRATE_OLD] +
                        " L_NEW is " + ((float
            [])trafficDest.pathLossRates.get(trafficDest.highestDerivativePath))[LOSSRATE_NEW] +
                                                                " DERIVATIVE is " + ((float
            [])trafficDest.pathLossRates.get(trafficDest.highestDerivativePath))[DERIVATIVE] +
                                                                " on path " +
            trafficDest.highestDerivativePath.intValue());

            panel.appendMessage("L_OLD is " + ((float
            [])trafficDest.pathLossRates.get(trafficDest.lowestDerivativePath))[LOSSRATE_OLD] +
                        " L_NEW is " + ((float
            [])trafficDest.pathLossRates.get(trafficDest.lowestDerivativePath))[LOSSRATE_NEW] +
                                                                " DERIVATIVE is " + ((float
            [])trafficDest.pathLossRates.get(trafficDest.lowestDerivativePath))[DERIVATIVE] +
                                                                " on path " +

            trafficDest.lowestDerivativePath.intValue());


              boolean overshoot = (((((float
              [])trafficDest.pathLossRates.get(trafficDest.highestDerivativePath))[LOSSRATE_NEW] + ((float
              [])trafficDest.pathLossRates.get(trafficDest.lowestDerivativePath))[LOSSRATE_NEW]) -
                 (((float [])trafficDest.pathLossRates.get(trafficDest.highestDerivativePath))[LOSSRATE_OLD] + ((float
              [])trafficDest.pathLossRates.get(trafficDest.lowestDerivativePath))[LOSSRATE_OLD])) >
              OVERSHOOT_THRESHOLD);

             boolean totalLSPLossRateNoLongerDecreased            = ((((float
             [])trafficDest.pathLossRates.get(trafficDest.highestDerivat ivePath))[LOSSRATE_OLD] + ((float
             [])trafficDest.pathLossRates.get(trafficDest.lowestDerivativePath))[LOSSRATE_OLD]) <=
                 (((float [])trafficDest.pathLossRates.get(trafficDest.highestDerivativePath))[LOSSRATE_NEW] + ((float
            [])trafficDest.pathLossRates.get(trafficDest.lowestDerivativePath))[LOSSRATE_NEW]));
              boolean HDPhasNoMoreAvailableBucket = (((getBETEwithPathID(trafficDest,
            trafficDest.highestDerivativePath).getSplit()) - (NOOFBUCKETS_TOSHIFT_ATONCE *
            PERCENT_PER_BUCKET)) < 0);

              if (overshoot)
              {
                panel.appendMessage("Overshoot detected, compensating");
                int previousSplitofHDP = getBETEwithPathID(trafficDest, trafficDest.highestDerivativePath).getSplit();
                getBETEwithPathID (trafficDest, trafficDest.highestDerivativePath).setSplit(previousSplitofHDP +
(NOOFBUCKETS_TOSHIFT_ATONCE * PERCENT_PER_BUCKET)) ;
                //((float [])trafficDest.pathLossRates.get(trafficDest.highestDerivativePath))[INCREMENT] =
NOOFBUCKETS_TOSHIFT_ATONCE * PERCENT_PER_BUCKET;
                panel.appendMessage("Traffic split on HDP (path " + trafficDest.highestDerivativePath.intValue() + ") is increased " +
(NOOFBUCKETS_TOSHIFT_ATONCE * PERCENT_PER_BUCKET) + " percent.");
                panel.appendMessage("New split is " + getBETEwithPathID (trafficDest, trafficDest.highestDerivativePath).getSplit());

                int previousSplitofLDP = getBETEwithPathID(trafficDest, trafficDest.lowestDerivativePath).getSplit();
                getBETEwithPathID(trafficDest, trafficDest.lowestDerivativePath).setSplit(previousSplitofLDP -
(NOOFBUCKETS_TOSHIFT_ATONCE * PERCENT_PER_BUCKET)) ;
                //((float [])trafficDest.pathLossRates.get(trafficDest.lowestDerivativePath))[INCREMENT] =
NOOFBUCKETS_TOSHIFT_ATONCE * PERCENT_PER_BUCKET;
                panel.appendMessage("Traffic split on LDP (path " + trafficDest.lowestDerivativePath.intValue() + ") is decreased " +
                (NOOFBUCKETS_TOSHIFT_ATONCE * PERCENT_PER_BUCKET) + " percent.");
```

```
                panel.appendMessage("New split is " + getBETEwithPathID (trafficDest, trafficDest.lowestDerivativePath).getSplit());

                gui.fillTable(getTable());
                trafficDest.compensated = true;
               }

               if (totalLSPLossRateNoLongerDecreased || HDPhasNoMoreAvailableBucket)
               {
                  panel.appendMessage("Finished equializing derivatives. Go to phase three");
                  trafficDest.phase = PHASE_THREE;
                        return;
               }
              }

             trafficDest.highestDerivativePath = highestDerivativePath(trafficDest);
             trafficDest.lowestDerivativePath = lowestDerivativePath(trafficDest);
             panel.appendMessage("HDP is " + trafficDest.highestDerivativePath  + " and LDP is " +
trafficDest.lowestDerivativePath);

             if (firstRound)
             {
               panel.appendMessage("L_OLD is " + ((float
[])trafficDest.pathLossRates.get(trafficDest.highestDerivativePath))[LOSSRATE_OLD] +
                          " L_NEW is " + ((float
[])trafficDest.pathLossRates.get(trafficDest.highestDerivativePath))[LOSSRATE_NEW] +
                                                                  " DERIVATIVE is " + ((float
[])trafficDest.pathLossRates.get(trafficDest.highestDerivativePath))[DERIVATIVE] +
                                                                  " on path " + trafficDest.highestDerivativePath.intValue());
    panel.appendMessage("L_OLD is " + ((float
[])trafficDest.pathLossRates.get(trafficDest.lowestDerivativePath))[LOSSRATE_OLD] +
                          " L_NEW is " + ((float
[])trafficDest.pathLossRates.get(trafficDest.lowestDerivativePath))[LOSSRATE_NEW] +
                                                                  " DERIVATIVE is " + ((float
[])trafficDest.pathLossRates.get(trafficDest.lowestDerivativePath))[DERIVATIVE] +
                                                                  " on path " + trafficDest.lowestDerivativePath.intValue());

             }

             if (trafficDest.highestDerivativePath == trafficDest.lowestDerivativePath)
             {
               panel.appendMessage("Derivatives are equal, no need to equalize derivatives; go to phase three");
               trafficDest.phase = PHASE_THREE;
               //panel.appendMessage("Phase is changed to PHASE_THREE");
             }
             else
             {
               panel.appendMessage("Equializing derivatives");
               int previousSplitofHDP = getBETEwithPathID(trafficDest, trafficDest.highestDerivativePath).getSplit();
               getBETEwithPathID (trafficDest, trafficDest.highestDerivativePath).setSplit(previousSplitofHDP -
(NOOFBUCKETS_TOSHIFT_ATONCE * (PERCENT_PER_BUCKET))) ;
               ((float [])trafficDest.pathLossRates.get(trafficDest.highestDerivativePath))[INCREMENT] =
NOOFBUCKETS_TOSHIFT_ATONCE * (PERCENT_PER_BUCKET);
               panel.appendMessage("Traffic split on HDP  (path " + trafficDest.highestDerivativePath.intValue() + ") is decreased " +
               ((float [])trafficDest.pathLossRates.get(trafficDest.highestDerivativePath))[INCREMENT] + " percent.");
               panel.appendMessage("New split is " + getBETEwithPathID (trafficDest, trafficDest.highestDerivativePath).getSplit());

               int previousSplitofLDP = getBETEwithPathID(trafficDest, trafficDest.lowestDerivativePath).getSplit();
               getBETEwithPathID(trafficDest, trafficDest.lowestDerivativePath).setSplit(previousSplitofLDP +
(NOOFBUCKETS_TOSHIFT_ATONCE * (PERCENT_PER_BUCKET))) ;
               ((float [])trafficDest.pathLossRates.get(trafficDest.lowestDerivativePath))[INCREMENT] =
NOOFBUCKETS_TOSHIFT_ATONCE * (PERCENT_PER_BUCKET);
               panel.appendMessage("Traffic split on LDP (path " + trafficDest.lowestDerivativePath.intValue() + ") is increased " +
               ((float [])trafficDest.pathLossRates.get(trafficDest.highestDerivativePath))[INCREMENT] + " percent.");
               panel.appendMessage("New split is " + getBETEwithPathID (trafficDest, trafficDest.lowestDerivativePath).getSplit());

               gui.fillTable(getTable());
             }//end if/else
 }//end phaseTwo()
```

```
/**
  * Implements the phase three of the MATE algorithm. Monitors the network conditions
  * @param   pplrVector the vector that contains path packet loss rate information
  */
 private void phaseThree(Vector pplrVector, TrafficDestination trafficDest)
 {
  panel.appendMessage("\nIn phaseThree()");
            trafficDest.phase = PHASE_THREE;

            //Phase Two exits after compensation without checking for violation. It is good to
            //check for violation here.
            if (trafficDest.compensated == true)
            {
             if (!trafficDest.alreadyComputedDerivatives)
             {
              computeDerivative (pplrVector, trafficDest, trafficDest.highestDerivativePath);
              computeDerivative (pplrVector, trafficDest, trafficDest.lowestDerivativePath);

               boolean violation = checkViolation (pplrVector, trafficDest, trafficDest.highestDerivativePath,
trafficDest.lowestDerivativePath);
              if (violation)
              {
               return;
              }
             }
             trafficDest.alreadyComputedDerivatives = false;
             trafficDest.compensated = false;
            }

            Enumeration enum = pplrVector.elements();
            while (enum.hasMoreElements())
            {
             BestEffortPerformanceUpdate.PathPacketLossRate pplr =
(BestEffortPerformanceUpdate.P athPacketLossRate)enum.nextElement();
             //panel.appendMessage("In while loop");
             //panel.appendMessage("pplr.getPathID() is " + pplr.getPathID());
             if (trafficDest.pathLossRates.containsKey(pplr.getPathID()))
             {
               float filteredLossRate = ((float[])trafficDest.pathLossRates.get(pplr.getPathID()))[LOSSRATE_NEW];
                        filteredLossRate = WEIGHTING_CONSTANT * filteredLossRate + (1 - WEIGHTING_CONSTANT) *
pplr.getPacketLossRate();
                        panel.appendMessage("\nFiltered current loss rate on path " + pplr.get PathID() + " is " + filteredLossRate);
                        panel.appendMessage("L_NEW on path " + pplr.getPathID() + " is " +
((float[])trafficDest.pathLossRates.get(pplr.getPathID()))[LOSSRATE_NEW]);
                        float currentLossRateChange = filteredLossRate -
((float[])trafficDest.pathLossRates.get(pplr.getPathID()))[LOSSRATE_NEW];
                        panel.appendMessage("Current loss rate change (Filtered current loss rate - L_NEW) on path " +
pplr.getPathID() + " is " + currentLossRateChange);

                        panel.appendMessage("L_OLD on path " + pplr.getPathID() + " is " +
((float[])trafficDest.pathLossRates.get(pplr.getPathID()))[LOSSRATE_OLD]);
                        float baseLossRateChange = ((float[])trafficDest.pathLossRates.get(pplr.getPathID()))[LOSSRATE_NEW] -
((float[])trafficDest.pathLossRates.get(pplr.getPathID()))[LOSSRATE_OLD];
                                panel.appendMessage("base loss rate (L_NEW - L_OLD) change on path " + pplr.getPathID() + " is
" + baseLossRateChange);
                                //panel.appendMessage("LOAD_CHANGE_THRESHOLD * baseLossRateChange on this path is "
+ LOAD_CHANGE_THRESHOLD * baseLossRateChange);

                        //To minimize syncronization due to multiple LSRs detecting the network change simultaneously;
                        //a random backOff time is used. This scheme outputs three number (0,1 and 2). In case "0", if other
                        //conditions are met, it directly enters phase four. In case "1" it waits for the next BEPU message
                        //and then check the network condition.In the last case (case "2"); it waits two BEPU messages
                        // and then checks network conditions. this scheme decreases the probability of entering phase
                        //simultaneously to 33%.

                        if (Math.abs(currentLossRateChange) > Math.max(LOAD_CHANGE_THRESHOLD,
Math.abs(baseLossRateChange)))
                        {
```

```
                        panel.appendMessage("Change in network conditions detected on path " + pplr.getPathID().intValue());
                        //panel.appendMessage("BACKOFF_BEPU_COUNT on this path is " + ((float
[])trafficDest.pathLossRates.get(pplr.getPathID()))[BACKOFF_BEPU_COUNT]);
                        if (((float [])trafficDest.pathLossRates.get(pplr.getPathID()))[BACKOFF_BEPU_COUNT] == -1f)
                        {
                          float randomBackOffNumber = (float)Math.floor(Math.random() * 3);
                                ((float[])trafficDest.pathLossRates.get(pplr.getPathID()))[BACKOFF_BEPU_COUNT] =
randomBackOffNumber;
                        }

                          int backOffBEPUcount =
(int)((float[])trafficDest.pathLossRates.get(pplr.getPathID()))[BACKOFF_BEPU_COUNT];

                          if (backOffBEPUcount == 0)
                          {
                           panel.appendMessage("backOffBEPUcount is 0");
                                trafficDest.phase = PHASE_FOUR;

                                phaseFour(pplrVector, trafficDest, pplr.getPathID(), filteredLossRate);
                                 break;
                          }
                          else if (backOffBEPUcount == 1)
                          {
                            panel.appendMessage("backOffBEPUcount is 1");

                            --((float[])trafficDest.pathLossRates.get(pplr.getPathID()))[BACKOFF_BEPU_COUNT];
                          }
                          else if (backOffBEPUcount == 2)
                          {
                            panel.appendMessage("backOffBEPUcount is 2");
                            --((float[])trafficDest.pathLossRates.get(pplr.getPathID()))[BACKOFF_BEPU_COUNT];
                          } else
                          {
                            panel.appendMessage("Unknown BACKOFF_BEPU_COUNT");
                          }
                         }//end if
                        else
                        {
                          panel.appendMessage("No appreciable change on path " + pplr.getPathID().intValue());
                          ((float [])trafficDest.pathLossRates.get(pplr.getPathID()))[BACKOFF_BEPU_COUNT] = -1f;
                        }
                       }//end if
                    }//end while

  }//end phaseThree()


  /**
   * Implements the phase four of the MATE algorithm. In the case of a congestion adjust
   * the derivatives values and calls phaseTwo
   * @param   pplrVector the vector that contains path packet loss rate information
   */
  private void phaseFour(Vector pplrVector, TrafficDestination trafficDest, Integer affectedPathID, float filteredLossRate)
  {
            trafficDest.phase = PHASE_FOUR;
            panel.appendMessage("\nIn phase four");

            if (filteredLossRate > ((float[])trafficDest.pathLossRates.get(affectedPathID))[LOSSRATE_NEW])
            {
              ((float[])trafficDest.pathLossRates.get(affectedPathID))[LOSSRATE_OLD] = filteredLossRate -
Math.min((filteredLossRate - ((float[])trafficDest.pathLossRates.get(affectedPathID))[LOSSRATE_NEW]), 2 * Math.max(
Math.abs(((float[])trafficDest.pathLossRates.get(affectedPathID))[LOSSRATE_NEW] -
                        ((float[])trafficDest.pathLossRates.get(affectedPathID))[LOSSRATE_OLD]),
LOAD_CHANGE_THRESHOLD));

                        ((float[])trafficDest.pathLossRates.get(affectedPathID))[LOSSRATE_NEW] = filteredLossRate;
                        ((float[])trafficDest.pathLossRates.get(affectedPathID))[INCREMENT] = 10;
```

```
                        ((float[])trafficDest.pathLossRates.get(affectedPathID))[DERIVATIVE] =
Math.abs(((float[])trafficDest.pathLossRates.get(affectedPathID))[LOSSRATE_NEW] -
                        ((float[])trafficDest.pathLossRates.get(affectedPathID))[LOSSRATE_OLD]) /
((float[])trafficDest.pathLossRates.get(affectedPathID))[INCREMENT];
                        panel.appendMessage("Derivative on path " + affectedPathID.intValue() + " is modified to " +
((float[])trafficDest.pathLossRates.get(affectedPathID))[DERIVATIVE]);

                        ((float[])trafficDest.pathLossRates.get(affectedPathID))[BACKOFF_BEPU_COUNT] = -1;
                      }
                      else //if (filteredLossRate <= ((float[])trafficDest.pathLossRates.get(affectedPathID))[LOSSRATE_NEW])
                      {
                        ((float[])trafficDest.pathLossRates.get(affectedPathID))[LOSSRATE_OLD] = filteredLossRate -
Math.max((filteredLossRate -
                          ((float[])trafficDest.pathLossRates.get(affectedPathID))[LOSSRATE_NEW]), -0.5f * Math.max(
Math.abs(((float[])trafficDest.pathLossRates.get(affectedPathID))[LOSSRATE_NEW] -
                              ((float[])trafficDest.pathLossRates.get(affectedPathID))[LOSSRATE_OLD]),
LOAD_CHANGE_THRESHOLD));

                        ((float[])trafficDest.pathLossRates.get(affectedPathID))[LOSSRATE_NEW] = filteredLossRate;
                        ((float[])trafficDest.pathLossRates.get(affectedPathID))[INCREMENT] = PERCENT_PER_BUCKET;

                        ((float[])trafficDest.pathLossRates.get(affectedPathID))[DERIVATIVE] =
Math.abs(((float[])trafficDest.pathLossRates.get(affectedPathID))[LOSSRATE_NEW] -
                              ((float[])trafficDest.pathLossRates.get(affectedPathID))[LOSSRATE_OLD]) /
((float[])trafficDest.pathLossRates.get(affectedPathID))[INCREMENT];
                        panel.appendMessage("Derivativ e on path " + affectedPathID.intValue() + " is modified to " +
((float[])trafficDest.pathLossRates.get(affectedPathID))[DERIVATIVE]);

                        ((float[])trafficDest.pathLossRates.get(affectedPathID))[BACKOFF_BEPU_COUNT] = -1;
                      }

                      Enumeration enum = pplrVe ctor.elements();
                      while (enum.hasMoreElements())
                      {
                        BestEffortPerformanceUpdate.PathPacketLossRate pplr =
(BestEffortPerformanceUpdate.PathPacketLossRate)enum.nextElement();
                        if (trafficDest.pathLossRates.containsKey(pplr.getPathID()))
                        {
                          if (pplr.getPathID().intValue() != affectedPathID.intValue())
                          {
                            float thisFilteredLossRate = ((float[])trafficDest.pathLossRates.get(pplr.getPathID()))[LOSSRATE_NEW];
                            thisFilteredLossRate = WEIGHTING_CONSTANT * thisFilteredLossRate + (1 -
WEIGHTING_CONSTANT) * pplr.getPacketLossRate();

                            ((float[])trafficDest.pathLossRates.get(pplr.getPathID()))[LOSSRATE_OLD] = thisFilteredLossRate -
                            (((float[])trafficDest.pathLossRates.get(pplr.getPathID()))[LOSSRATE_NEW] -
((float[])t rafficDest.pathLossRates.get(pplr.getPathID()))[LOSSRATE_OLD]);

                            ((float[])trafficDest.pathLossRates.get(pplr.getPathID()))[LOSSRATE_NEW] = thisFilteredLossRate;

                            ((float[])trafficDest.pathLossRates.get(pplr.getPathID()))[DERIVATIVE] =
Math.abs(((float[])trafficDest.pathLossRates.get(pplr.getPathID()))[LOSSRATE_NEW] -
                            ((float[])trafficDest.pathLossRates.get(pplr.getPathID()))[LOSSRATE_OLD]) /
((float[])trafficDest.pathLossRates.get(pplr.getPathID()))[INCREMENT];
                            panel.app endMessage("Derivative on path " + pplr.getPathID().intValue() + " is modified to " +
((float[])trafficDest.pathLossRates.get(pplr.getPathID()))[DERIVATIVE]);

                            ((float[])trafficDest.pathLossRates.get(pplr.getPathID()))[BACKOFF_BEPU_COUNT] = -1;
                          }//end if
                        }//end if
                      }//end while

                      trafficDest.highestDerivativePath = null;
                      trafficDest.lowestDerivativePath = null;
                      phaseTwo(pplrVector, trafficDest);

  }//end phaseFour()
```

```
 /**
  * Checks whether there is a violation of monotonous property of loss rate versus load.
  * If finds; shifts traffic back and forth until violation removes.
  * @param   trafDest TrafficDestination
  * @param   shiftToPath pathID of the path that traffic is shifted to
  * @param   shiftFromPath pathID of the path that traffic is shifted from
  * @param   callingPhase phase that calls this method
  * @return  violation true if violation exist, false otherwise
  */
 private boolean checkViolation (Vector pplrVector, TrafficDestination trafDest, Integer shiftToPath, Integer shiftFromPath)
 {
            panel.appendMessage("\nIn checkViolation() method");

            if (trafDest.checkViolationCounter == 0)
            {
              if (trafDest.phase == PHASE_ONE)
              {
                trafDest.violationShiftIncrement = (int) Math.floor((NUM_OF_BUCKETS / trafDest.noOfPaths) *
(PERCENT_PER_BUCKET));
              }
              if (trafDest.phase == PHASE_TWO)
              {
                trafDest.violationShiftIncrement = NOOFBUCKETS_TOSHIFT_ATONCE * PERCENT_PER_BUCKET;
              }
              if (trafDest.phase == PHASE_THREE)
              {
                trafDest.violationShiftIncrement = NOOFBUCKETS_TOSHIFT_ATONCE * PERCENT_PER_BUCKET;
              }

              trafDest.shiftToPath = shiftToPath;
              trafDest.shiftFromPath = shiftFromPath;

              if (!violation(trafDest, trafDest.shiftToPath, INCREASE_VIOLATION) && !violation(trafDest, trafDest.shiftFromPath,
DECREASE_VIOLATION))
              {
                        panel.appendMessage("No violation detected");
                        return false;
              }
              else
              {
                        trafDest.checkViolationCallerPhase = trafDest.phase;
                        trafDest.phase = VIOLATION_REMOVAL;

                        Integer tempPath = trafDest.shiftToPath;
                        trafDest.shiftToPath = trafDest.shiftFromPath;
                        trafDest.shiftFromPath = tempPath;
              }
            }
            else if ((trafDest.checkViolationCounter % 2) == 1)
            {
              panel.appendMessage("trafDest.checkViolationCounter % 2 is 1");
              computeDerivative(pplrVector, trafDest, trafDest.shiftToPath);
              computeDerivative(pplrVector, trafDest, trafDest.shiftFromPath);

              Integer tempPath = trafDest.shiftToPath;
              trafDest.shiftToPath = trafDest.shiftFromPath;
              trafDest.shiftFromPath = tempPath;
            }
            else if ((trafDest.checkViolationCounter % 2) == 0)
            {
              panel.appendMessage("trafDest.checkViolationCounter % 2 is 0");
              computeDerivative(pplrVector, trafDest, trafDest.shiftToPath);
              computeDerivative(pplrVector, trafDest, trafDest.sh iftFromPath);

              if (!violation(trafDest, trafDest.shiftToPath, INCREASE_VIOLATION) && !violation(trafDest, trafDest.shiftFromPath,
DECREASE_VIOLATION))
                        {
```

```
                panel.appendMessage("Violation removed");
                        trafDest.checkViolationCounter = 0;
                        trafDest.alreadyComputedDerivatives = true;
                        panel.appendMessage("trafDest.checkViolationCallerPhase is " + trafDest.checkViolationCallerPhase);
                        switch (trafDest.checkViolationCallerPhase)
                        {
                         case PHASE_ONE:
                           ++trafDest.indexOfLSPtoShiftTraffic;
                                panel.appendMessage("Returning back to PHASE_ONE");
                                phaseOne(pplrVector, trafDest);
                                break;

                         case PHASE_TWO:
                                panel.appendMessage("Returning back to PHASE_TWO");
                                phaseTwo(pplrVector, trafDest);
                                break;

                         case PHASE_THREE:
                                panel.appendMessage("Returning back to PHASE_THREE");
                                phaseThree(pplrVector, trafDest);
                                break;

                         default:
                                panel.appendMessage("Unknown checkViolationCallerPhase");
                                break;
                        }//end switch
                        return true;
                 }
                 else
                 {
                  Integer tempPath = trafDest.shiftToPath;
                        trafDest.shiftToPath = trafDest.shiftFromPath;
                        trafDest.shiftFromPath = tempPath;
                 }
                }

                panel.appendMessage("Shifting traffic in VIOLATION_REMOVAL phase");
                int previousSplitofShiftToPath = getBETEwithPathID(trafDest, trafDest.shiftToPath).getSplit();
                getBETEwithPathID (trafDest, trafDest.shiftToPath).setSplit(previousSplitofShiftToPath +
trafDest.violationShiftIncrement) ;
                ((float [])trafDest.pathLossRates.get(trafDest.shiftToPath))[INCREMENT] = trafDest.violationShiftIncrement;
                panel.appendMessage("Traffic split on ShiftToPath (path " + trafDest.shiftToPath.intValue() + ") is increased " +
                ((float [])trafDest.pathLossRates.get(trafDest.shiftToPath))[INCREMENT] + " percent.");
                panel.appendMessage("New split is " + getBETEwithPathID (trafDest, trafDest.shiftToPath).getSplit());

                int previousSplitofShiftFromPath = getBETEwithPathID(trafDest, trafDest.shiftFromPath).getSplit();
                getBETEwithPathID(trafDest, trafDest.shiftFromPath).setSplit(previousSplitofShiftFromPath -
trafDest.violationShiftIncrement) ;
                ((float [])trafDest.pathLossRates.get(trafDest.shiftFromPath))[INCREMENT] = trafDest.violationShiftIncrement;
                panel.appendMessage("Traffic split on ShiftFromPath (path " + trafDest.shiftFromPath.intValue() + ") is decreased " +
                ((float [])trafDest.pathLossRates.get(trafDest.shiftFromPath))[INCREMENT] + " percent.");
                panel.appendMessage("New split is " + getBETEwithPathID (trafDest, trafDest.shiftFromPath).getSplit());

                gui.fillTable(getTable());

                ++trafDest.checkViolationCounter;

                return true;
  }

  /**
   * Checks whether there is a violation of monotonous property of loss rate versus load.
   * @param   trafDest TrafficDestination
   * @param   pathID pathID of the path to be checked
                 * @return  violation true if violation exist, false otherwise
                 */
                private boolean violation (TrafficDestination trafDest, Integer pathID, byte action)
                {
```

```
            boolean violation = false;

            if (action == INCREASE_VIOLATION)
            {
              if (((float[])trafDest.pathLossRates.get(pathID))[LOSSRATE_NEW] <
                  ((float[])trafDest.pathLossRates.get(pathID))[LOSSRATE_OLD])
              {
                        violation = true;
                        panel.appendMessage("INCREASE_VIOLATION detected on path " + pathID.intValue());
              }
            }
            if (action == DECREASE_VIOLATION)
            {
              if (((float[])trafDest.pathLossRates.get(pathID))[LOSSRATE_NEW] >
                  ((float[])trafDest.pathLossRates.get(pathID))[LOSSRATE_OLD])
              {
                        violation = true;
                        panel.appendMessage("DECREASE_VIOLATION detected on path " + pathID.intValue());
              }
            }

            return violation;
 }//end violation()

 /**
  * Finds the Best Effort Table Entry in the table with given trafficDestination and pathID
  * @param   destination destination of the path
  * @param   path pathID
  * @return  bete Best Effort Table Entry
  */
 private BestEffortTableEntry getBETEwithPathID (TrafficDestination trafficDest, Integer path)
 {
   BestEffortTableEntry bete = null;
            for (int i = 0; i <= trafficDest.noOfPaths -1; ++i)
            {
              Integer thisPath = new Integer(((BestEffortTableEntry)get(trafficDest.destination.toString() + i)).getPathMap());
              if (path.intValue() == thisPath.intValue())
              {
                        bete = (BestEffortTableEntry)get(trafficDest.destination.toString() + i);
                        break;
              }
            }
            return bete;
 }// end of getBETEwithPathID()


 /**
  * Computes the derivative of a path
  * @param pplrVector PathPacketLossRate Vector
  * @param pathID pathID
  */
 private void computeDerivative(Vector pplrVector, TrafficDestination trafDest, Integer pathID)
 {
   Enumeration enum = pplrVector.elements();
            while (enum.hasMoreElements())
            {
              BestEffortPerformanceUpdate.PathPacketLossRate pplr =
(BestEffortPerformanceUpdate.PathPacketLossRate)enum.nextElement();
              if (pplr.getPathID().intValue() == pathID.intValue())
              {
                ((float [])trafDest.pathLossRates.get(pplr.getPathID()))[LOSSRATE_OLD] = ((float
[])trafDest.pathLossRates.get(pplr.getPathID()))[LOSSRATE_NEW];
                ((float [])trafDest.pathLossRates.get(pplr.getPathID()))[LOSSRATE_NEW] = pplr.getPacketLossRate();
                ((float [])trafDest.pathLossRates.get(pplr.getPathID()))[DERIVATIVE] = Math.abs(((float
[])trafDest.pathLossRates.get(pplr.getPathID()))[LOSSRATE_NEW] -
                          ((float [])trafDest.pathLossRates.get(pplr.getPathID()))[LOSSRATE_OLD]) / ((float
[])trafDest.pathLossRates.get(pplr.getPathID()))[INCREMENT];
                        }
```

```
            }
}//end of computeDerivative()


/**
 * Finds the path with highest derivative
 * @param   trafDest traffic destination
 * @return  highestDerivativePath the path with highest deriv ative
 */
private Integer highestDerivativePath(TrafficDestination trafDest)
{
  Enumeration keyEnum = trafDest.pathLossRates.keys();
            Integer highestDerivativePath = (Integer)keyEnum.nextElement();
            while (keyEnum.hasMoreElements())
            {
              Integer thisPath = (Integer)keyEnum.nextElement();
              float thisDerivative = ((float [])trafDest.pathLossRates.get(thisPath))[DERIVATIVE];
              float highestDerivative = ((float [])trafDest.pathLossRates.get(highestDerivativePath))[DERIVATIVE];

              if (thisDerivative > highestDerivative)
              {
                highestDerivativePath = thisPath;
              }
            }
            return highestDerivativePath;
}// end of highestDerivativePath()


/**
 * Finds the path with lowest derivative
 * @param   trafDest traffic destination
 * @return  lowestDerivativePath the path with lowest derivative
 */
private Integer lowestDerivativePath(TrafficDestination trafDest)
{
  Enumeration keyEnum = trafDest.pathLossRates.keys();
            Integer lowestDerivativePath = (Integer)keyEnum.nextElement();
            while (keyEnum.hasMoreElements())
            {
              Integer thisPath = (Integer)keyEnum.nextElement();
              float thisDerivative = ((float [])trafDest.pathLossRates.get(thisPath))[DERIVATIVE];
              float lowestDerivative = ((float [])trafDest.pathLossRates.get(lowestDerivativePath))[DERIVATIVE];

              if (thisDerivative < lowestDerivative)
              {
                lowestDerivativePath = thisPath;
              }
            }
            return lowestDerivativePath;
}// end of lowestDerivativePath()



/**
 * Required method for <code>MessageProcessor</code> interface.
 * @return message types processed
 */
public byte[] getMessageTypes()
{
  return myMessages;
}

/**
 * Required method for <code>ChannelListener</code> interface.
 * @param ce  Channel event received
 */
public void receiveEvent(ChannelEvent ce){}
```

```
 /**
  * If a <code>BestEffortTableEntry</code> has already been constructed,
  * this method allows it to be entered into the table.
  * @param betentry the <code>BestEffortTableEntry</code> to be entered.
  */
 public synchronized void add (BestEffortTableEntry betentry)
 {
   String key = betentry.getDestAddr().toString() + betentry.getSerialNo();
   put(key, betentry);
   gui.fillTable(getTable());
 } //end of add()


 /**
  * Returns the contents of the best effort table
  * in the form of a String (useful for displaying the table).
  * @return A String representation of the contents of the entire table.
  */
 public String toString()
 {
   String result = "Best Effort Table\n";

   Enumeration enum = elements();
   while (enum.hasMoreElements())
   {
     BestEffortTableEntry nextEntry = (BestEffortTableEntry) (enum.nextElement());
     result += nextEntry.toString() + "\n";
   }

   return result;
 } //toString()

} //end of class MATETable
```

APPENDIX B: BEST EFFORT PERFORMANCE UPDATE SOURCE CODE

```
//26Mar03[Ayvat]  - created

package org.saamnet.saam.message;

import org.saamnet.saam.net.*;
import org.saamnet.saam.util.*;
import org.saamnet.saam.router.*;

import java.net.UnknownHostException;
import java.util.Vector;

/**
 * Sent by the server to the best effort edge routers which is running MATE.
 * Server sends each edge router the packet loss rates of all paths that
 * they are sending best effort traffic.
 * <p>Copyright: Copyright (c) 2003</p>
 * <p>Company: Naval Postgraduate School</p>
 * @author SAAM Network Research Group
 * @version 1.0
 */
public class BestEffortPerformanceUpdate extends Message
{
  public static final int NUMBER_OF_PPLRS_INDEX = 3; //index to numberOfPPLRs field

  private static final short BEPU_FIXED_FIELD_LENGTH = 1; //there is only one byte fixed field

                                //which is numberOfPPLRs
  //private IPv6Address targetRouterID;
  private byte        numberOfPPLRs = 0;   //number of (code> PathPacketLossRate <code> entries
  private Vector      PPLRvector;

  //used by the server.
  /**
   * Constructs a <code>BestEffortPerformanceUpdate</code> with a given router ID.
   */
  public BestEffortPerformanceUpdate()
  {
    super (Message.BEST_EFFORT_PERFORMANCE_UPDATE);

    //Added message length field to the byte array
    bytes = Array.concat(type, PrimitiveConversions.getBytes(BEPU_FIXED_FIELD_LENGTH));
    bytes = Array.concat(bytes, numberOfPPLRs);

    PPLRvector = new Vector();
  }

  /**
   * Constructs a <code>BestEffortPerformanceUpdate</code> based on the given byte array
   * @param bytes byte array representation of the <code>BestEffortPerformanceUpdate</code>
   */
  public BestEffortPerformanceUpdate (byte [] bytes)
  {
    super (Message.BEST _EFFORT_PERFORMANCE_UPDATE);
    this.bytes = bytes;

    PPLRvector = new Vector();

    numberOfPPLRs = bytes[NUMBER_OF_PPLRS_INDEX];

    int index = NUMBER_OF_PPLRS_INDEX + 1;
    PathPacketLossRate pplr = null;
    for (int i = 0; i < numberOfPPLRs; i++)
```

```
  {
   pplr = new PathPacketLossRate (Array.getSubArray(bytes,index, index + PathPacketLossRate.length));
              index += PathPacketLossRate.length;

   PPLRvector.add(pplr);
  }

}//end constructor taking a byte [] input


/**
* inner class to hold path packet loss rate
*/
public class PathPacketLossRate
{
    private int pathID;
    private short packetLossRate;
    private byte [] bytes = null;
    public static final int length = 6;

    /*
    * constructor
    * @param pathID path ID
    * @param packetLossRate packet loss rate on the path which has the path ID "pathID".
    */
    public PathPacketLossRate (Integer pathID, short packetLossRate)
    {
            this.pathID = pathID.intValue();
            this.packetLossRate = packetLossRate;
            bytes = Array.concat(PrimitiveConversions.getBytes(pathID.intValue()),
              PrimitiveConversions.getBytes(packetLossRate));
    }

    /*
    * Constructs a <code>PathPacketLossRate<code> with the given byte array
    * @param iBytes byte array representation of the <code>PathPacketLossRate<code>
    */
    public PathPacketLossRate (byte [] iBytes)
    {
            int pathIDindex = 0;
            int packetLossRateIndex = 4;
            pathID = PrimitiveConversions.getInt(Array.getSubArray(iBytes, pathIDindex, pathIDindex + 4));
            packetLossRate = PrimitiveConversions.getShort(Array.getSubArray(iBytes, packetLossRateIndex,
                                                    packetLossRateIndex + 2));
    }

    /*
    * returns the byte array "bytes"
    * @return byte array representation of the <code>PathPacketLossRate<code>
    */
    public byte [] getBytes ()
    {
            return bytes;
    }

    /*
    * returns path ID
    * @return pathID path ID
    */
    public Integer getPathID ()
    {
            Integer thisPathID = new Integer (pathID);
            return thisPathID;
    }

    /*
    * returns packet loss rat e
    * @return packetLossRate
```

```
   */
   public short getPacketLossRate ()
   {
            return packetLossRate;
   }


   /*
   * returns string representation of the class
   * @return String representation of the inner class <code>PathPacketLossRate<code>
   */
   public String toString ()
   {
            return ("Packet loss rate on path " + pathID + " is " + packetLossRate);
   }
}//end of inner class <code>PathPacketLossRate<code>


  /**
 * Inserts an interface SA.
 * @param isa item to insert
 */
public void insertPPLR(PathPacketLossRate pplr)
{
  //increment the number of path packet loss rates
  numberOfPPLRs++;

  //update the byte [] at number of interfaces
  bytes[NUMBER_OF_PPLRS_INDEX] = numberOfPPLRs;

  bytes = Array.concat(bytes, pplr.getBytes());
  PPLRvector.add(pplr);

  //update the message length field
  byte [] lengthBytes = PrimitiveConversions.getBytes((short) (bytes.length - 3));
   bytes[1] = lengthBytes[0];
   bytes[2] = lengthBytes[1];
}

/**
 * Inserts a vector of interface SA.
 * @param isaVector vector of <code>InterfaceSA</code>
 */
public void insertPPLRvector(Vector pplrVector)
{
  numberOfPPLRs += pplrVector.size();
  bytes[NUMBER_OF_PPLRS_INDEX] = numberOfPPLRs;

  for (int i = 0; i < pplrVector.size(); i++)
   {
    PathPacketLossRate tempPPLR = (PathPacketLossRate) pplrVector.elementAt(i);

    bytes = Array.concat(bytes, tempPPLR.getBytes());
    PPLRvector.add(tempPPLR);

  }//end for loop

  //change the length field of the byte array -- two-byte integer
  //Note that the message length doesn't factor in the "type" and "message length" fields
  byte [] lengthBytes = PrimitiveConversions.getBytes((short) (bytes.length - 3));
  bytes[1] = lengthBytes[0];
  bytes[2] = lengthBytes[1];

}//end insertISAVectors()
```

```
 /**
  * Returns the vector of Path packet loss rates.
  * @return vector of <code>PathPacketLossRate</code>
  */
 public Vector getPPLRvector()
 {
  return PPLRvector;
 }


 /**
  * Returns the number of Path packet loss rates.
  * @return numberOfPPLRS
  */
 public byte getNumberOfPPLRs()
 {
  return numberOfPPLRs;
 }

 /**
  * Returns a new instance of PathPacketLossRate.
  * @return new instance of PathPacketLossRate
  */
 public PathPacketLossRate createPathPacketLossRate(Integer IntPathID, short lossRate)
 {
  return new PathPacketLossRate(IntPathID, lossRate);
 }

 /**
  * Returns a new instance of PathPacketLossRate.
  * @return new instance of PathPacketLossRate
  */
 public PathPacketLossRate createPathPacketLossRate(byte [] newBytes)
 {
  return new PathPacketLossRate(newBytes);
 }

 /**
  * Returns the string representation of this message.
  * @return string representation of <code>BestEffortPerformanceUpdate</code>
  */
 public String toString()
 {
  return "Best Effort Performance Update Message";
 }


}//end of class BestEffortPerformanceUpdate
```

APPENDIX C: MODIFICATIONS TO BEST EFFORT MANAGER SOURCE CODE

```
// [ba] TEST PURPOSES ONLY
  /**
   * [ba]
   * This is for test purposes
   * Processes EdgeNotification messages.
   * @param   edgeNotif  the message
   */
  protected void processEdgeNotificationTest (EdgeNotification edgeNotif)
  {
    //initialize localResolutionTimeout variable
    if (!lrtInitialized)
    {
     localResolutionTimeout = ce.NUM_OF_BUCKETS * myServer.getAC_cyclePeriod();
     lrtInitialized = true;
     panel.appendMessage("\nLocal resolution timeout is " + localResolutionTimeout + "ms.");
    }

    int count = 0;//used below to figure out how much information is new

    panel.appendMessage("\nProcessing edge notification.");

    IPv6Address interfaceAddress = edgeNotif.getEdgeInterfaceAddress();
            IPv6Address edgeRouterAddress = edgeNotif.getEdgeRouterAddress();


    //Xie-darpa
    BasePIB.InterfaceInfo edgeInterfaceInfo = (BasePIB.InterfaceInfo) myBasePIB.htInterfaces.get(interfaceAddress.toString());
    if (edgeInterfaceInfo == null)
    {
     panel.appendMessage("\n  PIB is not ready; quit processing the edge notification message.");
     return;
    }


    //is this a newly discovered edge router?
    if (!(vBestEffortRouters.contains(edgeRouterAddress.toString())))
    {
     panel.appendMessage("Adding " + edgeRouterAddress.toString() + " to edge routers vector.");
     vBestEffortRouters.add(edgeRouterAddress.toString());
     count++;
            }

            //is this a newly discovered destination interface?
    if (!(vBestEffortDestAdds.contains(interfaceAddress.toString())))
    {
     panel.appendMessage("Adding " + interfaceAddress.toString() + " to edge interfaces vector.");
     vBestEffortDestAdds.add(interfaceAddress.toString());
     count++;
    }

    BasePIB.InterfaceInfo edgeRouterInterfaceInfo = (BasePIB.InterfaceInfo)
myBasePIB.htInterfaces.get(edgeRouterAddress.toString());

    int srcNodeID = edgeRouterInterfaceInfo.getNodeID().intValue();
    IPv6Address srcRouterID = (IPv6Address) myBasePIB.htNodeIDtoRouterID.get(new Integer (srcNodeID));

            //IPv6Address srcRouterID = edgeRouterAddress;
            //Integer srcNodeID = (Integer) myBasePIB.htRouterIDtoNodeID.get(edgeRouterAddress);

            int destNodeID = edgeInterfaceInfo.getNodeID().intValue();
    IPv6Address destRouterID = (IPv6Address) myBasePIB.htNodeIDtoRouterID.get(new Integer (destNodeID));
```

```
            //Vector pathVector = new Vector();

            BasePIB.Path bePath1 = myBasePIB.routingAlgorithm.findPath(srcRouterID,
                                        destRouterID,
                                        null,
                                        myBasePIB.routingAlgorithm.SHORTEST_WIDEST_PATH);
 if (bePath1 != null)
 {
  //pathVector.add(bePath1);
              Integer bePathID1 = bePath1.getPathID();
  if (!bePath1.bCreated)
  {
   myBasePIB.setupPath(bePath1, bePathID1.intValue(), FlowRoutingTableEntry.INSTALLED_FOR_BE);
   bePath1.bCreated = true;
  }
  panel.appendMessage("Path " + bePathID1.intValue() +
                  " deployed as primary for (" + srcNodeID +
                  "," + destNodeID + ").");


            //SHORTEST WIDEST MOST DISJOINT PATH  is used for the alternate path
  BasePIB.Path bePath2 = myBasePIB.routingAlgorithm.findPath(srcRouterID,
                                        destRouterID,
                                        bePath1,
                                        myBasePIB.routingAlgorithm.SHORTEST_WIDEST_MOST_DISJOINT_PATH);
  if (bePath2 != null)
  {
   //pathVector.add(bePath2);

              Integer bePathID2 = bePath2.getPathID();
   if (!bePath2.bCreated)
   {
    myBasePIB.setupPath(bePath2, bePathID2.intValue(), FlowRoutingTableEntry.INSTALLED_FOR_BE );
    bePath2.bCreated = true;
   }
   panel.appendMessage("Path " + bePathID2.intValue() + " deployed as alternate.");
  }
  else
  {
   bePath2 = bePath1;
   panel.appendMessage("No alternate path available.");
  }

            //send table entries to source Router
            //for (int i = 0; i < pathVector.size(); ++i)
  //{
            myServer.sendBETUpdate(srcRouterID, interfaceAddress, bePath1.getPathID().intValue(), 0, 0);
  bePath1.initiateBestEffortTraffic();
                        //if (i == 0)
                        //{
            bePath1.timeBEinitiated -= 1;
                        //}
  //}
            myServer.sendBETUpdate(srcRouterID, interfaceAddress, bePath2.getPathID().intValue(), 0, 0);
  bePath2.initiateBestEffortTraffic();


 }//end if

}//end of processEdgeNotificationTest()


/**
 * Processes EdgeNotification messages
 * @param   edgeNotif  the message
 */
protected void processMATEEdgeNotification (EdgeNotification edgeNotif)
{
```

```
int countRouter = 0;//used below to figure out whether a new router is added
        int countInterface = 0; //used below to figure out whether a new router is added

panel.appendMessage("\nProcessing edge notification.");

IPv6Address interfaceAddress = edgeNotif.getEdgeInterfaceAddress();


          bestEffortAgentType = "MATETable";
          panel.appendMessage("\n  MATE is active");


//Xie-darpa
BasePIB.InterfaceInfo edgeInterfaceInfo = (BasePIB.InterfaceInfo) myBasePIB.htInterfaces.get(interfaceAddress.toString());
if (edgeInterfaceInfo == null)
{
 panel.appendMessage("\n  PIB is not ready; quit processing the edge notification message.");
 return;
}

int nodeID = edgeInterfaceInfo.getNodeID().intValue();
IPv6Address routerID = (IPv6Address) myBasePIB.htNodeIDtoRouterID.get(new Integer (nodeID));

//is this a newly discovered edge router?
if (!(vBestEffortRouters.contains(routerID.toString())))
{
 panel.appendMessage("Adding " + routerID.toString() + " to edge routers vector.");
 vBestEffortRouters.add(routerID.toString());
 countRouter++;
}
//is this a newly discovered destination interface?
if (!(vBestEffortDestAdds.contains(interfaceAddress.toString())))
{
 panel.appendMessage("Adding " + interfaceAddress.toString() + " to edge interfaces vector.");
 vBestEffortDestAdds.add(interfaceAddress.toString());
 countInterface++;
}

//A new interface and a new router received. Deploy paths from all best effort routers
        //to this new router and from this router to all other best effort routers
        if ( (countRouter > 0) && (countInterface > 0) )
        {
          panel.appendMessage("Updating MATE best effort topology...");
  byte action = NEWROUTER_NEWINTERFACE;
          updateMATEBEtopology(action);
  panel.appendMessage("Completed updating MATE best effort topology.");
        }
        //A new interface of an already known router is received. Deploy paths from all other best
        //effort routers to the router of this new interface (send only the new interface as destination adress)
        else if ( (countRouter == 0) && (countInterface > 0) )
        {
          panel.appendMessage("Updating MATE best effort topology...");
  byte action = ONLY_NEWINTERFACE;
          updateMATEBEtopology(action);
  panel.appendMessage("Completed updating MATE best effort topology.");
        }
        else if ( (countRouter > 0) && (countInterface == 0) )
        {
          panel.appendMessage ("New router without new interface?, this is weird!");
        }
        else
{
 panel.appendMessage("No new information; best effort topology still accurate.");
}
}//end of processMATEEdgeNotification()


// [ba] TEST PURPOSES ONLY
```

```
/**
 * this is for test purpose
 * Processes EdgeNotification messages
 * @param   edgeNotif  the message
 */
protected void processMATEEdgeNotificationTest (EdgeNotification edgeNotif)
{
  int countRouter = 0;//used below to figure out whether a new router is added
            int countInterface = 0; //used below to figure out whether a new router is added

  panel.appendMessage("\nProcessing edge notification.");

  IPv6Address interfaceAddress = edgeNotif.getEdgeInterfaceAddress();
            IPv6Address edgeRouterAddress = edgeNotif.getEdgeRouterAddress();
            panel.appendMessage("Edge Router Address is " + edgeRouterAddress.toString());

            //possibility
              bestEffortAgentType = "MATETable";
              panel.appendMessage("\n  MATE is active");


  //Xie-darpa
  BasePIB.InterfaceInfo edgeInterfaceInfo = (BasePIB.InterfaceInfo) myBasePIB.htInterfaces.get(interfaceAddress.toString());
  if (edgeInterfaceInfo == null)
  {
   panel.appendMessage("\n  PIB is not ready; quit processing the edge notification message.");
   return;
  }

  //is this a newly discovered edge router?
  if (!(vBestEffortRouters.contains(edgeRouterAddress.toString())))
  {
   panel.appendMessage("Adding " + edgeRouterAddress.toString() + " to edge routers vector.");
   vBestEffortRouters.add(edgeRouterAddress.toString());
   countRouter++;
  }

   //is this a newly discovered destination interface?
  if (!(vBestEffortDestAdds.contains(interfaceAddress.toString())))
  {
   panel.appendMessage("Adding " + interfaceAddress.toString() + " to edge interfaces vector.");
   vBestEffortDestAdds.add(interfaceAddress.toString());
   countInterface++;
  }

            BasePIB.InterfaceInfo edgeRouterInterfaceInfo = (BasePIB.InterfaceInfo)
myBasePIB.htInterfaces.get(edgeRouterAddress.toString());

   int srcNodeID = edgeRouterInterfaceInfo.getNodeID().intValue();
  IPv6Address srcRouterID = (IPv6Address) myBasePIB.htNodeIDtoRouterID.get(new Integer (srcNodeID));

            //IPv6Address srcRouterID = edgeRouterAddress;
            //Integer srcNodeID = (Integer) myBasePIB.htRouterIDtoNodeID.get(edgeRouterAddress);

            int destNodeID = edgeInterfaceInfo.getNodeID().intValue();
  IPv6Address destRouterID = (IPv6Address) myBasePIB.htNodeIDtoRouterID.get(new Integer (destNodeID));

            Vector MATEPathsVector = myBasePIB.routingAlgorithm.findMATEPaths(srcRouterID, destRouterID);


            //for test purposes only
            //Routers that generates cross traffic should send their traffic only via primary path(SWP).
            String router3Address = "99.99.99.99.17.0.0.0.0.0.0.0.0.0.0.1";
            String router4Address = "99.99.99.99.9.0.0.0.0.0.0.0.0.0.0.1";
            String router5Address = "99.99.99.99.19.0.0.0.0.0.0.0.0.0.0.1";
            if (edgeRouterAddress.toString().equals(router3Address) || edgeRouterAddress.toString().equals(router4Address) ||
               edgeRouterAddress.toString().equals(router5Address))
            {
```

```
                BasePIB.Path alternatePath = (BasePIB.Path)MATEPathsVector.lastElement();
                MATEPathsVector.removeElement(alternatePath);
              }

             if ((!MATEPathsVector.isEmpty()) && (((BasePIB.Path)MATEPathsVector.elementAt(0)) != null))
    {
     panel.appendMessage("Following MATE paths are deployed for (" + srcNodeID +
                          "," + destNodeID + ") : ");
              for(int i = 0; i < MATEPathsVector.size(); ++i)
     {
                Integer MATEPath = ((BasePIB.Path)MATEPathsVector.elementAt(i)).getPathID();
                panel.appendMessage(MATEPath.intValue() + "");
      if (!((BasePIB.Path)MATEPathsVector.elementAt(i)).bCreated)
       {
        myBasePIB.setupPath(((BasePIB.Path)MATEPathsVector.elementAt(i)), MATEPath.intValue(),
FlowRoutingTableEntry.INSTALLED_FOR_BE);
        ((BasePIB.Path)MATEPathsVector.elementAt(i)).bCreated = true;
       }
     }

              panel.appendMessage("Roters in htMATERouterstoPaths are:");
              Enumeration thisEnum = htMATERouterstoPaths.keys();
              while (thisEnum.hasMoreElements())
              {
                        String routerID = ((String)thisEnum.nextElement());
                        panel.appendMessage(routerID);
              }
              panel.appendMessage("srcRouterID is " + srcRouterID.toString());

              //add sourceRouterID and corresponding paths to htMATERouterstoPaths hash table
              if (htMATERouterstoPaths.containsKey(srcRouterID.toString()))
              {
               ((Vector)htMATERouterstoPaths.get(srcRouterID.toString())).addAll(MATEPathsVector);

                        panel.appendMessage(srcRouterID.toString() + " is already in htMATERouterstoPaths.");
                        panel.appendMessage("Following paths are in htMATERouterstoPaths");
                        Enumeration enum  = ((Vector)htMATERouterstoPaths.get(srcRouterID.toString())).elements();
                        while (enum.hasMoreElements())
                        {
                          int pathID = ((BasePIB.Path)enum.nextElement()).getPathID().intValue();
                          panel.appendMessage("" + pathID);
                        }
              }
              else
              {
               htMATERouterstoPaths.put(srcRouterID.toString(), MATEPathsVector);

                        panel.appendMessage(srcRouterID.toString() + " is newly added to htMATERouterstoPaths.");
                        panel.appendMessage("Following paths are in htMATERouterstoPaths for this router");
                        Enumeration enum = ((Vector)htMATERouterstoPaths.get(srcRouterID.toString())).elements();
                        while (enum.hasMoreElements())
                        {
                          int pathID = ((BasePIB.Path)enum.nextElement()).getPathID().intValue();
                          panel.appendMessage("" + pathID);
                        }
              }

              //send table entries to source Router
              for (int i = 0; i < MATEPathsVector.size(); ++i)
    {
                myServer.sendBETUpdate(srcRouterID, interfaceAddress,
((BasePIB.Path)MATEPathsVector.elementAt(i)).getPathID().intValue(), i, 0);
      ((BasePIB.Path)MATEPathsVector.elementAt(i)).initiateBestEffortTraffic();
     }



              //start timer to periodically send <code>BestEffortPerformanceUpdate<code> message
```

```
            if (bBEPUTimerStarted == false)
            {
              bBEPUTimerStarted = true;
              startBestEffortPerformanceUpdate();
            }
           }//end if
  }//end of processMATEEdgeNotificationTest()


/**
   * [ba] This is for MATE
   * Every time a new edge router is discovered, the BE topology is updated
   * and new paths are deployed as necessary.
   */
  private void updateMATEBEtopology(byte action)
  {
    try
    {
     switch (action)
        {
          case NEWROUTER_NEWINTERFACE:
          panel.appendMessage("In case NEWROUTER_NEWINTERFACE");
          Enumeration eSources = vBestEffortRouters.elements();
          while (eSources.hasMoreElements())
          {
            IPv6Address srcRouterID = IPv6Address.getByName((String) eSources.nextElement());
            Integer srcNodeID = ((BasePIB.InterfaceInfo) myBasePIB.htInterfaces.get(srcRouterID.toString())).getNodeID();

           //Last element of the vBestEffortRouters vector is the newly added vector. Deploy paths
           //between all other routers and this new router.
          IPv6Address interfaceAddress = IPv6Address.getByName((String) vBestEffortRouters.lastElement());
          Integer destNodeID = ((BasePIB.InterfaceInfo) myBasePIB.htInterfaces.get(interfaceAddress.toString())).getNodeID();
          IPv6Address destRouterID = (IPv6Address) myBasePIB.htNodeIDtoRouterID.get(destNodeID);

          Vector MATEPathsVector = myBasePIB.routingAlgorithm.findMATEPaths(srcRouterID, destRouterID);

          if ((!MATEPathsVector.isEmpty()) && (((BasePIB.Path)MATEPathsVector.elementAt(0)) != null))
            {
             panel.appendMessage("Following MATE paths are deployed for (" + srcNodeID.intValue() +
                                                                  "," + destNodeID.intValue() + ") : ");
            for(int i = 0; i < MATEPathsVector.size(); ++i)
            {
               Integer MATEPath = ((BasePIB.Path)MATEPathsVector.elementAt(i)).getPathID();
               panel.appendMessage(MATEPath.intValue() + "");
               if (!((BasePIB.Path)MATEPathsVector.elementAt(i)).bCreated)
               {
                myBasePIB.setupPath(((BasePIB.Path)MATEPathsVector.elementAt(i)), MATEPath.intValue(),
                 FlowRoutingTableEntry.INSTALLED_FOR_BE);
                 ((BasePIB.Path)MATEPathsVector.elementAt(i)).bCreated = true;
               }
            }

panel.appendMessage("Roters in htMATERouterstoPaths are:");
                                          Enumeration thisEnum = htMATERouterstoPaths.keys();
                                          while (thisEnum.hasMoreElements())
                                          {
                                                  String routerID = ((String)thisEnum.nextElement());
                                                  panel.appendMessage(routerID);
                                          }
                                          panel.appendMessage("srcRouterID is " + srcRouterID.toString());

                                          //add sourceRouterID and corresponding paths to htMATERouterstoPaths hash table
                                          if (htMATERouterstoPaths.containsKey(srcRouterID.toString()))
                                          {

((Vector)htMATERouterstoPaths.get(srcRouterID.toString())).addAll(MATEPathsVector);
```

```
                                                panel.appendMessage(srcRouterID.toString() + " is already in
htMATERouterstoPaths.");
                                                panel.appendMessage("Following paths are in htMATERouterstoPaths");
                                                Enumeration enum =
((Vector)htMATERouterstoPaths.get(srcRouterID.toString())).elements();
                                                while (enum.hasMoreElements())
                                                {
                                                  int pathID = ((BasePIB.Path)enum.nextElement()).getPathID().intValue();
                                                  panel.appendMessage("" + pathID);
                                                }
                                        }
                                        else
                                        {
                                         htMATERouterstoPaths.put(srcRouterID.toString(), MATEPathsVector);

                                                panel.appendMessage(srcRouterID.toString() + " is newly added to
htMATERouterstoPaths.");
                                                panel.appendMessage("Following paths are in htMATERouterstoPaths for
this router");
                                                Enumeration enum =
((Vector)htMATERouterstoPaths.get(srcRouterID.toString())).elements();
                                                while (enum.hasMoreElements())
                                                {
                                                  int pathID = ((BasePIB.Path)enum.nextElement()).getPathID().intValue();
                                                  panel.appendMessage("" + pathID);
                                                }
                                        }
                sendTableEntries(srcRouterID, destRouterID, MATEPathsVector, action);

                                        //start timer to periodically send <code>BestEffortPerformanceUpdate<code> message
                                        if (bBEPUTimerStarted == false)
                                        {
                                          bBEPUTimerStarted = true;
                                          startBestEffortPerformanceUpdate();
                                        }
                                       }//end if


                                       //Now deploy the reverse paths. Paths from new router to all other routers
                                       Vector MATEReversePathsVector =
myBasePIB.routingAlgorithm.findMATEPaths(destRouterID, srcRouterID);
                                       if ((!MATEReversePathsVector.isEmpty()) &&
(((BasePIB.Path)MATEReversePathsVector.elementAt(0)) != null))
                {

                 panel.appendMessage("Following MATE paths are deployed for (" + destNodeID.intValue() +
                                                        "," + srcNodeID.intValue() + ") : ");
                                          for(int i = 0; i < MATEReversePathsVector.size(); ++i)
                {
                                            Integer MATEPath =
((BasePIB.Path)MATEReversePathsVector.elementAt(i)).getPathID();
                                            panel.appendMessage(MATEPath.intValue() + "");
                  if (!((BasePIB.Path)MATEReversePathsVector.elementAt(i)).bCreated)
                  {
                    myBasePIB.setupPath(((BasePIB.Path)MATEReversePathsVector.elementAt(i)), MATEPath.intValue(),
FlowRoutingTableEntry.INSTALLED_FOR_BE);
                    ((BasePIB.Path)MATEReversePathsVector.elementAt(i)).bCreated = true;
                  }
                }

                                       panel.appendMessage("Roters in htMATERouterstoPaths are:");
                                       Enumeration thisEnum = htMATERouterstoPaths.keys();
                                       while (thisEnum.hasMoreElements())
                                       {
                                                String routerID = ((String)thisEnum.nextElement());
                                                panel.appendMessage(routerID);
                                       }
                                       panel.appendMessage("destRouterID is " + destRouterID.toString());
```

```
                                        //add sourceRouterID and corresponding paths to htMATERouterstoPaths hash table
                                        if (htMATERouterstoPaths.containsKey(destRouterID.toString()))
                                        {

((Vector)htMATERouterstoPaths.get(destRouterID.toString())).addAll(MATEReversePathsVector);

                                                panel.appendMessage(destRouterID.toString() + " is already in
htMATERouterstoPaths.");
                                                panel.appendMessage("Following paths are in htMATERouterstoPaths");
                                                Enumeration enum =
((Vector)htMATERouterstoPaths.get(destRouterID.toString())).elements();
                                                while (enum.hasMoreElements())
                                                {
                                                  int pathID = ((BasePIB.Path)enum.nextElement()).getPathID().intValue();
                                                  panel.appendMessage("" + pathID);
                                                }
                                        }
                                        else
                                        {
                                         htMATERouterstoPaths.put(destRouterID.toString(), MATEReversePathsVector);

                                                panel.appendMessage(destRouterID.toString() + " is newly added to
htMATERouterstoPaths.");
                                                panel.appendMessage("Following paths are in htMATERouterstoPaths");
                                                Enumeration enum =
((Vector)htMATERouterstoPaths.get(destRouterID.toString())).elements();
                                                while (enum.hasMoreElements())
                                                {
                                                  int pathID = ((BasePIB.Path)enum.nextElement()).getPathID().intValue();
                                                  panel.appendMessage("" + pathID);
                                                }
                                        }
            sendTableEntries(destRouterID, srcRouterID, MATEReversePathsVector, action);

                                        //start timer to periodically send <code>BestEffortPerformanceUpdate<code> message
                                        if (bBEPUTimerStarted == false)
                                        {
                                          bBEPUTimerStarted = true;
                                          startBestEffortPerformanceUpdate();
                                        }
                                       }//end if

          }//end of while
          break;

         case ONLY_NEWINTERFACE:
                                    panel.appendMessage("In case ONLY_NEWINTERFACE");
                                    Enumeration enumSources = vBestEffortRouters.elements();
          while (enumSources.hasMoreElements())
          {
            IPv6Address srcRouterID = IPv6Address.getByName((String) enumSources.nextElement());
            Integer srcNodeID = ((BasePIB.InterfaceInfo) myBasePIB.htInterfaces.get(srcRouterID.toString())).getNodeID();

                                        IPv6Address interfaceAddress = IPv6Address.getByName((String)
vBestEffortDestAdds.lastElement());
            Integer destNodeID = ((BasePIB.InterfaceInfo) myBasePIB.htInterfaces.get(interfaceAddress.toString())).getNodeID();
            IPv6Address destRouterID = (IPv6Address) myBasePIB.htNodeIDtoRouterID.get(destNodeID);

                    Vector MATEPathsVector = myBasePIB.routingAlgorithm.findMATEPaths(srcRouterID, destRouterID);

                                        if ((!MATEPathsVector.isEmpty()) &&
(((BasePIB.Path)MATEPathsVector.elementAt(0)) != null))
            {
             panel.appendMessage("Following MATE paths are deployed on router " + srcNodeID.intValue() +
                                                " for destination interface address " + interfaceAddress.toString() + ": ");
                                         for(int i = 0; i < MATEPathsVector.size(); ++i)
             {
```

```
                                        Integer MATEPath = ((BasePIB.Path)MATEPathsVector.elementAt(i)).getPathID();
                                        panel.appendMessage(MATEPath.intValue() + "");
            if (!((BasePIB.Path)MATEPathsVector.elementAt(i)).bCreated)
            {
              myBasePIB.setupPath(((BasePIB.Path)MATEPathsVector.elementAt(i)), MATEPath.intValue(),
FlowRoutingTableEntry.INSTALLED_FOR_BE);
              ((BasePIB.Path)MATEPathsVector.elementAt(i)).bCreated = true;
            }
           }

                                        //add sourceRouterID and corresponding paths to htMATERouterstoPaths hash table
                                        if (htMATERouterstoPaths.containsKey(srcRouterID.toString()))
                                        {
                                         Enumeration myEnum = MATEPathsVector.elements();
                                                while (myEnum.hasMoreElements())
                                                {
                                                 BasePIB.Path pathID = (BasePIB.Path)myEnum.nextElement();
                                                 if
(!((Vector)htMATERouterstoPaths.get(srcRouterID.toString())).contains(pathID))
                                                 {
                                                  ((Vector)htMATERouterstoPaths.get(srcRouterID.toString())).add(pathID);
                                                        panel.appendMessage(pathID.getPathID().intValue() + " is added
to htMATERouterstoPaths hash table for router " + srcRouterID.toString());
                                                        panel.appendMessage("Following paths are in
htMATERouterstoPaths");
                                                  Enumeration anotherEnum =
((Vector)htMATERouterstoPaths.get(srcRouterID.toString())).elements();
                                                   while (anotherEnum.hasMoreElements())
                                                   {
                                                    int thisPathID =
((BasePIB.Path)anotherEnum.nextElement()).getPathID().intValue();
                                                    panel.appendMessage("" + thisPathID);
                                                   }
                                                 }
                                                 else
                                                 {
                                                        panel.appendMessage(pathID.getPathID().intValue() + "is already
in htMATERouterstoPaths hash table");
                                                 }
                                                }
                                        }
                                        else
                                        {
                                         htMATERouterstoPaths.put(srcRouterID.toString(), MATEPathsVector);

                                                panel.appendMessage(srcRouterID.toString() + " is newly added to
htMATERouterstoPaths.");
                                                panel.appendMessage("Following paths are in htMATERouterstoPaths");
                                                Enumeration enum =
((Vector)htMATERouterstoPaths.get(srcRouterID.toString())).elements();
                                                while (enum.hasMoreElements())
                                                {
                                                 int pathID = ((BasePIB.Path)enum.nextElement()).getPathID().intValue();
                                                 panel.appendMessage("" + pathID);
                                                }
                                        }

          sendTableEntries(srcRouterID, destRouterID, MATEPathsVector, action);

                                        //start timer to periodically send <code>BestEffortPerformanceUpdate<code> message
                                        if (bBEPUTimerStarted == false)
                                        {
                                         bBEPUTimerStarted = true;
                                         startBestEffortPerformanceUpdate();
                                        }
                                       }//end if
        }//end of while
                                break;
```

```
                                    default:
                                      break;
        }//end of switch
    }
            catch (UnknownHostException uhe)
    {
     System.out.println("UHE thrown by updateMATEBEtopology() in BestEffortManager.");
    }

  }//end of updateMATEBEtopology()



/**
   * Starts best effort performance update for MATE routers.
   */
  private void startBestEffortPerformanceUpdate()
  {
    //Send first BestEffortPerformanceUpdate message.This message is sent before paths are deployed.
            //The loss rate values of the first path before shifting traffic on this path is used while
            //computing loss rate derivatives of the first path.
            //updateBestEffortPerformance();

            Runnable update = new Runnable()
    {
     public void run()
     {
      while (true)
      {
       try
       {
        synchronized (updateLock)
        {
         updateLock.wait();
        }
        updateBestEffortPerformance();
                        }
       catch (InterruptedException e)
       {
        System.out.println(e);
        System.exit(1);
       }
      }//end while
     }
    };

            performanceUpdater = new Thread(update, "performanceUpdater");
   performanceUpdater.start();

   panel.appendMessage("Auto configuration cycle is " + myServer.getAC_cyclePeriod());
   updateTimer = new Timer((UPDATE_PERIOD * myServer.getAC_cyclePeriod() * ce.getTimeScale()),
                    new ActionListener()
                    {
                     public void actionPerformed(ActionEvent event)
                     {
                      panel.appendMessage("Timer expired. update best effort performance for MATE routers...");
                      synchronized (updateLock)
                      {
                       updateLock.notify();
                      }
                                                                        }
                    }
                   );
   updateTimer.start();


  }//end startBestEffortPerformanceUpdate()
```

```
/**
 * Updates Best Effort Performance of the paths that are assigned to MATE routers
 */
private void updateBestEffortPerformance()
{
        panel.appendMessage("Updating loss rates on paths MATE routers use");
        Enumeration enumKey = htMATERouterstoPaths.keys();

  while (enumKey.hasMoreElements())
  {
        String thisRouterAddress = (String)enumKey.nextElement();
        Vector paths = (Vector)htMATERouterstoPaths.get(thisRouterAddress);
        Enumeration enumPath = paths.elements();

        Vector pathPacketLossRateVector = new Vector();
        BestEffortPerformanceUpdate.PathPacketLossRate pathPacketLossRate;
        BestEffortPerformanceUpdat e bepu = new BestEffortPerformanceUpdate();

        while (enumPath.hasMoreElements())
        {
                BasePIB.Path thisPath = (BasePIB.Path)enumPath.nextElement();
                Integer pathID = thisPath.getPathID();
                short packetLossRate = thisPath.getPathServiceLevelQoS(BasePIB.BEST_EFFORT).getPacketLossRate();

                pathPacketLossRate = bepu.createPathPacketLossRate(pathID, packetLossRate);
                pathPacketLossRateVector.add(pathPacketLossRate);
        }//end while

        try
        {
          IPv6Address routerAddress = IPv6Address.getByName(thisRouterAddress);
          myServer.sendBEPUpdate(routerAddress, pathPacketLossRateVector);
        }
        catch (UnknownHostException uhe)
   {
    System.out.println("UHE thrown by beNodePairAdmin() in BestEffortManager.");
   }

  }//end while

        panel.appendMessage("Finished updating Path packet loss rates");

} //end of updateBestEffortPerformance()
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX D: MODIFICATIONS TO BASEPIB SOURCE CODE

```
/**
 * [ba]
    * Finds paths for MATE routers using routing algorithms in place.
    * @param sourceRouterID the IPv6 address of the source router.
    * @param destinationRouterID the IPv6 address of the destination router.
    * @return the Vector of Path objects.
    */
   protected Vector findMATEPaths(
     IPv6Address sourceRouterID,
     IPv6Address destinationRouterID)
   {
     panel.appendMessage("\nfindMATEPaths()");
         panel.appendMessage("source address" + sourceRouterID.toString() +
                         "\ndestination address" + destinationRouterID.toString() );

         Vector MATEPathsVector = new Vector ();
         Path SWP = null;
         Path SWMDP = null;
         Path SWLCP = null;



         SWP = findPathSWP(sourceRouterID, destinationRouterID);
         MATEPathsVector.add(SWP);
         //panel.appendMessage("SWP is " + SWP.getPathID().intValue());

         SWMDP = findPathSWMDP(sourceRouterID, destinationRouterID, SWP);
         MATEPathsVector.add(SWMDP);
         //panel.appendMessage("SWMDP is " + SWMDP.getPathID().intValue());



         /*  possible future implementation to find all possible paths between
         an ingress and egress node pair.

      InterfaceInfo interfaceInformation =
         (InterfaceInfo) htInterfaces.get(sourceRouterID.toString());

      int sourceNodeID = ((InterfaceInfo) htInterfaces
                     .get(sourceRouterID.toString())).getNodeID().intValue();

      int destNodeID = ((InterfaceInfo) htInterfaces
                     .get(destinationRouterID.toString())).getNodeID().intValue();

      Path thisPath = null;

      Hashtable table = new Hashtable();

      for (int i = 1; i < MAX_HOP_COUNT; i++)
      {
       testMsg("Hop count = " + i);
       table = aPI[sourceNodeID][destNodeID][i];
       Enumeration enum = table.elements();

       if (enum.hasMoreElements())
       {
        //Cycle through each of the paths of the current hop count, between
        //source and destination nodes
        while (enum.hasMoreElements())
        {
         Integer currentPathID = (Integer) enum.nextElement();

         // Extract current path information
```

```
      thisPath = (Path) htPaths.get(currentPathID);
       MATEPathsVector.add(thisPath);

     } //end of while-loop
   } //end of if structure
 } //end of for-loop

*/

 if (!MATEPathsVector.isEmpty())
 {
   int srcNodeID = ((Integer)htRouterIDtoNodeID.get(sourceRouterID.toString())).intValue();
          int destNodeID = ((Integer)htRouterIDtoNodeID.get(destinationRouterID.toString())).intValue();

          panel.appendMessage("MATE paths between (" + srcNodeID + "," + destNodeID + ") are:\t");
          if (!MATEPathsVector.isEmpty())
          {
            for( int i = 0; i < MATEPathsVector.size(); i++)
            {
                        if (((Path)MATEPathsVector.elementAt(i)) != null)
                        {
                          panel.appendMessage("" + ((Path)MATEPathsVector.elementAt(i)).toString());
                        }
              else
                        {
                          panel.appendMessage("null");
                        }
            }
          }
          else
          {
            panel.appendMessage("MATEPathsVector is empty");
          }
 }
 return MATEPathsVector;
} //end of findMATEPaths()
```

LIST OF REFERENCES

[1] Wofford, Corey D., A Best Effort Traffic Management Solution for Server and Agent-Based Active Network Management, Computer Science Department, Naval Postgraduate School, Monterey, March 2002.

[2] Vrable, Dean J. and Jarger, John W., "The SAAM Architecture: Enabling Integrated Services", Computer Science Department, Naval Postgraduate School, Monterey, September 1999.

[3] Elwalid, Anwar, Cheng Zin, Stephen Low, and Indra Widjaja, "MATE: MPLS Adaptive Traffic Engineering", *IEEEINFOCOM*, April 2001.

[4] Kurose, F. J. and Ross, W.Keith, Computer Networking, A Top-Down Approach Featuring The Internet, Addison Wesley, 2001.

[5] Awduche, D., J. Malcolm, J. Agogbua, M. O'Dell, and J. McManus, "Requirements for Traffic Engineering Over MPLS", RFC 2702, September 1999.

[6] Rosen, E., Viswanathan, A., Callon, R., "Multiprotocol Label Switching Architecture", RFC 3031, January 2001.

[7] Wang, J., Patek, S., Wang, H., Liebeherr J., "Traffic Engineering with AIMD in MPLS Networks", Proceedings of 7th International Workshop on Protocols for High Speed Networks, April 22-24, Berlin, Germany.

[8] Elwalid, Anwar, Widjaja, Indra, "MATE: MPLS Adaptive Traffic Engineering" Internet draft <draft-widjaja-mpls-mate-01.txt>, October 1999.

[9] Turksoyu, Fatih, "Realistic Traffic Generation Capability for SAAM Testbed", Computer Science Department, Naval Postgraduate School, Monterey, March 2001.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia 22060

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California 93943

3. Professor Geoffrey Xie
   Department of Computer Science
   Naval Postgraduate School
   Monterey, California 93943

4. Mr. John Gibson
   Department of Computer Science
   Naval Postgraduate School
   Monterey, California 93943

5. Deniz Kuvvetleri Komutanligi
   Kutuphane
   Bakanliklar, Ankara, TURKEY

6. Deniz Harp Okulu Komutanligi
   Kutuphane
   Tuzla, Istanbul, TURKEY

7. Dz. Utgm. Birol AYVAT
   Envanter Kontrol Merkezi Komutanligi
   Golcuk,Kocaeli, TURKEY