Theses and Dissertations   1. Thesis and Dissertation Collection, all items

2002-03

A best effort traffic management solution for server and agent-based active network management (SAAM)

Wofford, Corey D.

Monterey, California. Naval Postgraduate School

https://hdl.handle.net/10945/6000

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A BEST EFFORT TRAFFIC MANAGEMENT SOLUTION FOR SERVER AND AGENT-BASED ACTIVE NETWORK MANAGEMENT (SAAM)

by

Corey D. Wofford

March 2002

Thesis Advisor:	Geoffrey Xie
Co-Advisor:	James Bret Michael

Approved for public release; distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE March 2002	3. REPORT TYPE AND DATES COVERED Master's Thesis	
4. TITLE AND SUBTITLE A Best Effort Traffic Management Solution for Server and Agent-based Active Network Management (SAAM)		5. FUNDING NUMBERS	
6. AUTHOR (S) Corey D. Wofford			
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, 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)*

Server and Agent-based Active Network Management (SAAM) is a promising network management solution for the Internet of tomorrow, "Next Generation Internet (NGI)." SAAM is a new network architecture that incorporates many of the latest features of Internet technologies. The primary purpose of SAAM is managing network quality of service (QoS) to support the resource-intensive next-generation Internet applications.

Best effort (BE) traffic will continue to exist in the era of NGI. Thus SAAM must be able to manage such traffic. In this thesis, we propose a solution for management of BE traffic within SAAM. With SAAM, it is possible to make a "better best effort" in routing BE packets. Currently, routers handle BE traffic based solely on local information or from information obtained by link-state flooding which may not be reliable. In contrast, SAAM centralizes management at a server where better (more optimal) decisions can be made. SAAM's servers have access to accurate topology and timely traffic-condition information. Additionally, due to their placement on high-end routers or dedicated machines, the servers can better afford computationally intensive routing solutions. It is these characteristics that are exploited by the solution design and implementation of this thesis.

14. SUBJECT TERMS Next Generation Internet, Quality of Service, Best Effort Traffic, Networks, Routing, Resource Management			15. NUMBER OF PAGES 163
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

A BEST EFFORT TRAFFIC MANAGEMENT SOLUTION FOR SERVER AND AGENT-BASED ACTIVE NETWORK MANAGEMENT (SAAM)

Corey Wofford
Lieutenant, United States Navy
B.S., Michigan State University, 1995

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

**NAVAL POSTGRADUATE SCHOOL
March 2002**

Author: Corey Wofford

Approved by: Geoffrey Xie, Thesis Advisor

                  James Bret Michael, Co-Advisor

                  Chris Eagle, Chairman
                  Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Server and Agent-based Active Network Management (SAAM) is a promising network management solution for the Internet of tomorrow, "Next Generation Internet (NGI)." SAAM is a new network architecture that incorporates many of the latest features of Internet technologies. The primary purpose of SAAM is managing network quality of service (QoS) to support the resource-intensive next-generation Internet applications.

Best effort (BE) traffic will continue to exist in the era of NGI. Thus SAAM must be able to manage such traffic. In this thesis, we propose a solution for management of BE traffic within SAAM. With SAAM, it is possible to make a "better best effort" in routing BE packets. Currently, routers handle BE traffic based solely on local information or from information obtained by link-state flooding which may not be reliable. In contrast, SAAM centralizes management at a server where better (more optimal) decisions can be made. SAAM's servers have access to accurate topology and timely traffic-condition information. Additionally, due to their placement on high-end routers or dedicated machines, the servers can better afford computationally intensive routing solutions. It is these characteristics that are exploited by the solution design and implementation of this thesis.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGEMENTS

I give thanks to God for life, wisdom, and salvation through his Son.

I thank DARPA and NASA, whose funding made this project possible.

I would especially like to thank my advisor, Geoffrey Xie. He was there for me 100% and was my teacher from start to finish. Through the course of working with this man, I learned so much more than just technical knowledge. Thanks to him, I've come away not just with improved research skills, but a deeper passion for computer networks as well.

I greatly appreciate the time and efforts of Bret Michael, my co-advisor. He devoted enormous amounts of time and energy in helping me polish my final product. From him, I learned quite a bit about technical writing as well as greater discipline in research and development.

I also thank Cary Colwell without whom I would have had to sacrifice a great deal of quality. His coaching on the technical aspects of the SAAM architecture proved invaluable. I'll always remember those long days in the lab with Cary as my sole companion. I consider him a friend.

My wife and children supported me the whole way and for that I am grateful. Without their support, none of this would have been possible. Billie was my Proverbs 31 woman. My children's hugs after dog days kept my spirits up.

Last, but not least, I thank my parents Dale and Gayle. I have the deepest respect for their marriage and for them as people. They've stayed true to each other and they've stayed true to me. I draw a great deal of strength from them. They are a rock for me.

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

The Internet was born out of an experimental project in the late 1960's, funded by the Advanced Research Projects Agency (ARPA). Since then, the Internet has exploded into a worldwide phenomenon with millions of hosts to include private citizens, businesses, schools, governments, and of course, the DoD (the original customer).

The Internet came to be defined by an underlying core technology known as TCP/IP. This model/protocol has proved immensely scalable in terms of meeting the data transfer demands of the ever-growing amount of hosts, data, and links; it's simply a matter of adding more resources. Unfortunately, there are new demands today above and beyond "data transfer" that TCP/IP is not built to handle. These are the demands for Quality of Service (QoS). The TCP/IP model is a "best effort" model. That is, much like regular mail at the Post Office, the Internet will simply make a best effort to get the data to where it's addressed with no explicit guarantee of delivery time or success rate. For many of today's cutting edge applications in use by DoD and elsewhere, guarantees of service quality are required.

A project known as Server and Agent-based Active Network Management (SAAM) under the Next Generation Internet (NGI) initiative has as its goal the identification of solutions that could provide guaranteed QoS while still maintaining the simplicity and robustness of the underlying TCP/IP architecture. As of this writing, SAAM has done just that. After three years in research and development at the Naval Postgraduate School, SAAM has become a somewhat comprehensive solution that addresses not only issues of QoS, but also security, fault-tolerance, and policy management among others. One of the last areas to be developed for SAAM is its management of best effort traffic, the type of traffic that exists in the Internet today. Such interoperability with and management of best effort traffic is required in order for SAAM to viably deploy and integrate into today's Internet. That is the topic of this thesis.

This thesis develops a solution for best effort traffic management to incorporate into SAAM. First, best effort traffic management is researched through open sources for critical evaluation of previously developed ideas and whether or not they are applicable

to the SAAM model. Next, the SAAM architecture is studied in order to leverage SAAM's core strengths and ensure the best fit for a modular solution. Finally, the solution is designed, developed, integrated into SAAM, and tested for satisfactory performance. Conclusions are drawn and recommendations are made for possible future work in refining best effort traffic management for SAAM.

I. INTRODUCTION

A. MOTIVATION

The SAAM architecture has been and is currently being designed with NGI in mind for which increasingly sophisticated applications will be provided QoS guarantees for their data flows within a region of SAAM routers. Research on SAAM revolved around various QoS parameters and path constraints and how to manage those constraints in terms of Admission Control and Resource Management. If the Internet would instantly transform into the so-called NGI, this would be the end of the matter. All traffic would be preceded by resource reservation requests (a "flow request") to a SAAM server. Of course, as with most new computer network technology, it would be naïve to imagine a flip-of-the-switch technology conversion and forego any consideration of backwards compatibility.

If SAAM were deployed as is, on the assumption that all flows will be preceded by flow requests or reside within an aggregate Differentiated Service (DiffServ) level, a number of adverse consequences could arise. For one, a user internal to a SAAM region may be unnecessarily denied service until someone proficient with the SAAM client software can properly configure his/her device. For something as simple as viewing a static web page or sending an email message, such effort is questionable. Second, an autonomous system (AS) adopting the SAAM architecture may inadvertently violate service level agreements with neighboring AS's for carrying transit traffic. In reality, some of the applications on internal hosts and border routers in external Internet regions will continue to operate on the same assumption underlying today's Internet, the best effort model. The traffic these applications send and receive is known as best effort traffic. While specialized protocols and data exchange methods abound, it is universally understood and assumed that best effort (BE) traffic, when sent, will be handled according to uniform standards set forth by the Internet Engineering Task Force (IETF). Indeed, misbehavior by these standards can be punishable through "black holing," whereby Internet names and address entries are removed from the governing hierarchy tables, a logical severance from communications with the rest of the Internet.

The question to be explored here is the manner in which to handle BE traffic. Presumably, clients of a SAAM network provider will still have demand for BE services. While these may not generate as much revenue as QoS streams per unit of bandwidth consumed, SAAM must handle this traffic in a satisfactory manner to meet the needs of the customer and the expectations of the universal Internet. Further, the client will expect at least the same guarantee for BE traffic that the Internet already provides: a best effort (i.e., SAAM cannot arbitrarily drop or misroute packets unless constrained).

Until now, the question of how to characterize and handle BE traffic within a SAAM region has been left unanswered, but not locked out. That is, SAAM already has a couple of building blocks within its architecture into which BE traffic could be easily shoehorned. Indeed, these building blocks were designed as possible hooks with an eye towards reaching an ultimate solution in the future. Now that the cement is drying on the fundamental QoS portion of the architecture (SAAM's charter), a look towards how to now incorporate BE management into the total solution is appropriate. One possibility has always been SAAM's DiffServ solution. Conventionally, DiffServ has been divided into aggregate levels of standard QoS guarantees. One division, familiarly known as the "Olympic model of DiffServ" is Gold, Silver, and Bronze. If Bronze's QoS guarantee was simply best effort with no regard for delay, bandwidth, jitter, or loss rate, than this could be the service level to encapsulate BE traffic. The SAAM region would have to be pre-configured such that every end-to-end path (or point-to-point link in a slightly different scheme) would have an existing approved Bronze flow. The second solution is static provision of links. SAAM currently sets initial bandwidth reservations on each link for Control Traffic, Integrated Service (IntServ), DiffServ, Best Effort, Unallocated, and Out-of-Profile. As shown in Figure 1, BE receives an initial 15% of the link in this scheme. While the provisions for IntServ and DiffServ are dynamic (they change with load conditions), the provision for best effort is fixed at 15%. This allocation is a minimum since BE can also use whatever bandwidth QoS is not fully utilizing at the time. In earlier SAAM design efforts, BE's provision was set at 20% in order to prevent starvation of BE traffic [8]. Later, this level was reduced to 15% as part of the inter-service borrowing solution developed in [9]. No testing has been conducted on any

Internet service provider (ISP) to verify the suitability of this provision. However, historical BE utilization levels on the Internet's backbone are known to be approximately 10%. More importantly, these are just example allocations for use within the current prototype.
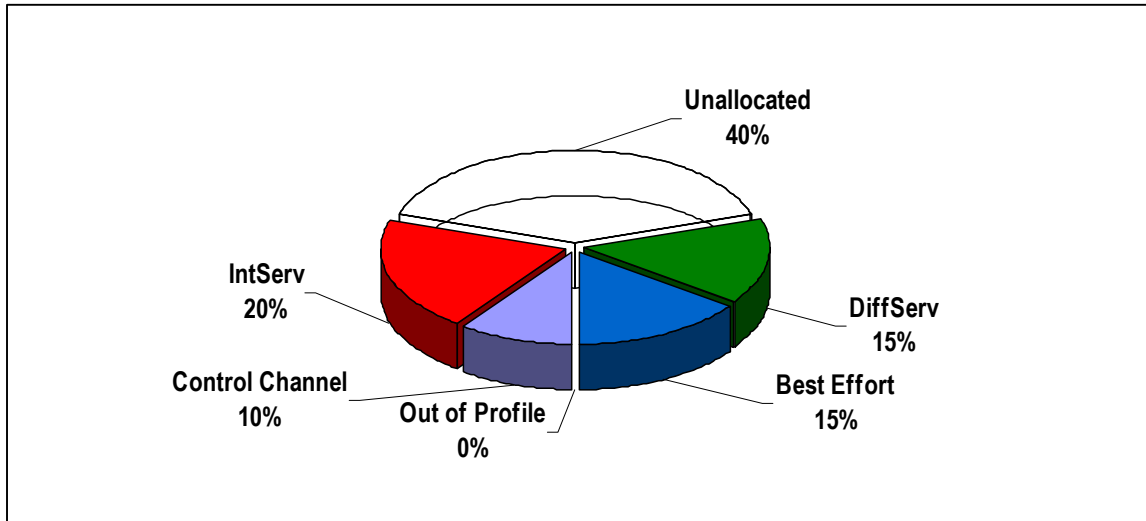
Figure 1.   Base Allocation per Service Level (From [9])

Essentially, BE has a network unto itself, a carbon copy of the underlying topology with at least 15% of the bandwidth. Within the SAAM research group, one solution proposed has been to let the legacy routing protocols and programs of today stay resident and handle that part of the network management. This has some obvious drawbacks. Primarily, it would relegate the SAAM solution to being an add-on module and prevent it from being a total integrated solution. The ability to function as a module on a device rather than completely "owning" the device necessitates at least one more software interface layer and more complexity for the underlying OS to manage calls to the hardware. Increased complexity usually translates to decreases in scalability, speed, or efficiency. In the area of computing, this is always true when another interface layer is inserted between an application and the underlying hardware.

SAAM needs to be a total solution so that it can completely control an underlying network and thereby maximize the speed and efficiency, not to mention the overall simplicity of that network's architecture. SAAM' is agent-based whereby specialized agents can be deployed at run-time to devices needing to handle special situations. Therefore, it is tempting to argue that SAAM can be deployed lacking certain

compatibilities since, as the Internet and technologies change, new agents can simply be developed, released, and deployed as needed. This may be advisable for those obscure technologies and protocols that are still emerging or in decline. For something as essential as handling BE traffic, however, functionality should be provided with the initial release.
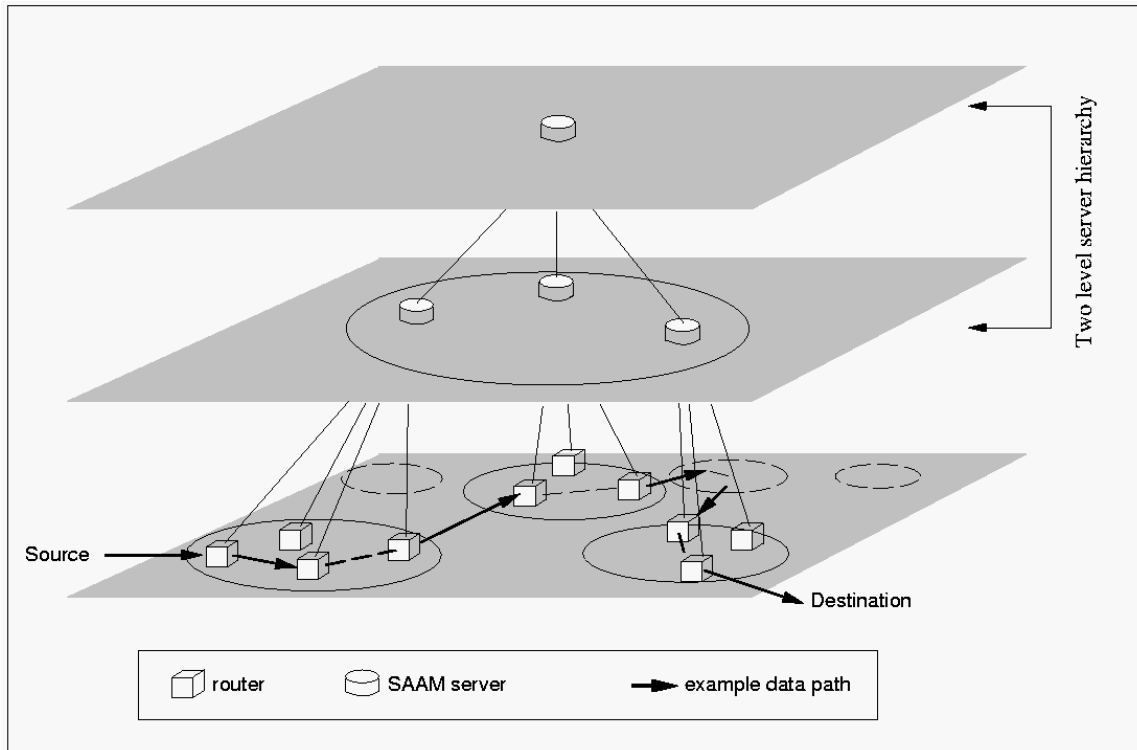
Figure 2.   SAAM Hierarchy (From [12])

As seen in Figure 2, SAAM is organized into a hierarchy of controlling servers over underlying SAAM regions of routers. The servers provide the points of centralized management, while the routers provide the labor for routing packets. The servers are the heavyweight decision makers that maintain large amounts of data and perform the processing required to make computationally intensive decisions based on that data. The routers, on the other hand, are lightweight performing simple packet forwarding and reporting tasks. From the ground up, this division of labor has been designed with QoS management in mind. The hierarchy of servers coordinates to meet QoS guarantees by coordinating resource usage within and between the SAAM regions. The routers simply install agents or routing table entries sent to them by the servers. While it seems a simple

task at the router level, they are an integral part of a coordinated and complex decision (made by a server above them) process that intelligently provides QoS.

Traditionally, BE traffic management has been performed in an opposite manner. Routers are alone and must act autonomously on decisions as to where they should forward packets. The process centers on nothing more than a single packet at a single moment in time. That is, the decision doesn't involve any variable or historic data other then a single entry in a lookup table. The router builds this table from either a human administrator or from a peer router that is directly neighboring or intraregional. Still, the routers remain distributed autonomous systems answering to no one except those with access to the physical device. As such, these routers tend to behave in a greedy fashion when it comes to routing BE traffic. Their routing decisions will be that which seems best based on their own router-centric algorithm. This decision may not be the best interest for the greater network or for the end users who are depending on a particular data flow. Enter SAAM with its sophisticated management capabilities for a network region and the Internet at large. Surely, there must be some way for SAAM to leverage its centralized management to handle BE traffic in a better fashion than the organized chaos of distributed routers in a network acting autonomously.

If SAAM did take on the task of recognizing and managing BE traffic flowing within its regions, there would be limitless possibilities for the method of accomplishment. While a router in today's Internet must make routing decisions based on late, sometimes inaccurate information, SAAM maintains comprehensive information about the regions it manages that is only slightly time-late. Minus any NP-complete problem, therefore, the server should be able to make better decisions than a distributed router operating with incomplete information. SAAM could do something as simple and unimaginative as using a shortest path (SP) algorithm to route its BE traffic, as is done by most routers today. If nothing else, this method would have the advantage of making routers more lightweight. However, as occurs in networks using this scheme, this can cause undue congestion on interior, critical (defined in [17]) links, whose bandwidth may be inadequate to support periods of high demand. It also fails to capitalize on most of the information that is maintained by the server. Finally, it is a static solution not becoming

5

of SAAM where information is kept up-to-date allowing dynamic measures, both proactive and reactive.

The current Internet BE model is built on simply making a best effort to get packets to their ultimate destination. As David Isenberg puts it in [7], "just deliver the bits, stupid." This issue is a small one that exists at that single point in time and space when a nameless router somewhere in the Internet receives a faceless packet and decides how to send it on its way. Perhaps SAAM could go further and tackle some of the broader issues as well. SAAM tracks traffic network-wide, providing for refereeing traffic condition and ensuring fairness in scheduling. Since it maintains global information about network resources, SAAM should also be able to ensure better utilization of those resources than a collection of routers running greedy algorithms with local information. Finally, when a failure occurs, a SAAM router has an ally an Internet router does not: the SAAM server. The SAAM region should be able to deal better with issues of fault tolerance than a stand-alone router.

B.   PROBLEM STATEMENT AND APPROACH

The development of SAAM is nearly complete. SAAM has the capabilities to take control of a region of routers and handle QoS traffic by establishing servers and deploying router agents. These features answer the challenge of SAAM's original charter. Still, even as Internet Protocol Version 4 (IPv4) has yet to be replaced (or even slightly displaced) by IPv6, so also it is recognized that the need to handle BE traffic will exist for the foreseeable future.

The overarching research question we address here is that, if it were to do so, how should SAAM handle BE traffic? More specifically, how can SAAM handle BE traffic while maintaining its guarantees to QoS flows and minimizing congestion of the BE traffic? Further, since all nodes are potentially equal customers with equal rights (excluding those who have purchased QoS guarantees), how can SAAM enforce some degree of fairness in the event of insufficient resources for high volumes of BE traffic?

The approach to answering these questions begins with research. First, broad research is conducted with a general survey of past and current methods for routing BE traffic. Next, research focuses on those solutions that are being developed today for

problems similar to SAAM's: handling BE traffic over QoS networks. Finally, plausible solutions are studied in detail for concepts applicable to a SAAM implementation.

Once promising concepts are identified and isolated, the next step is to build a solution that will fit in conceptually and be able to integrate practically into the SAAM project. This starts with conceptual mapping of concepts into design diagrams followed by a parsing of that abstract representation into object schema. From there, it is simply a matter of translating identified objects into actual software objects to run on the SAAM test-bed. For most SAAM development, such translation is accomplished through the Java programming language with which all of SAAM's core components have been developed.

Once the solution is built into SAAM, then simulation testing begins by building test scenarios which will validate certain key portions of the BE traffic management solution. These scenarios are constructed using the Extensible Markup Language (XML). Easily modifiable XML files facilitate the iterative design process by allowing concrete identification of goals through actual scenarios that present the problem situation(s).

Finally, results of testing are analyzed to determine whether any further changes are necessary to the solution. Once the iterative design process is complete, final results are collected and analyzed for conclusions. Evaluating the project as a whole with the new BE traffic solution, recommendations are made for areas of future research and improvement.

C. THESIS SCOPE

The scope of this thesis will be a research survey of existing solutions for BE traffic over QoS in NGI-type models and then development of a creative and superior solution that builds on the unique capabilities provided by the SAAM architecture. This new solution will then be implemented and tested in the existing SAAM prototype.

Specifically, this solution examines management of BE traffic beginning and ending in the ISO OSI Network layer in routers within a SAAM region. This thesis does not develop a solution for routing BE packets beyond the borders of the SAAM region,

7

though consideration is given to making a follow-on solution to address this easy to implement. In addition, this thesis does not develop an interface for handling BE traffic on a SAAM router that originates within or is destined for the ISO OSI Application layer outside of the SAAM software.

D. CONTRIBUTIONS OF THIS THESIS

1. Major Contributions

A solution for managing BE traffic has been developed and incorporated into the DARPA and NASA sponsored SAAM project at Naval Postgraduate School. The solution developed is presented as "a" BE solution for SAAM; no claims are made to the effect of being "the" best solution possible. Several areas invite further research and/or possible improvement.

2. Minor Contributions

The simulation environment for SAAM has been enhanced to facilitate the testing conducted as part of this thesis. First, the flow generator developed in [11] has been modified to allow packet count as a test parameter. Second, a simulation delay time has been incorporated into the test code that causes data rate to accurately match link speed. Finally, the server code has been modified to make the server better recognize network failures during simulation.

E. THESIS ORGANIZATION

The remainder of the thesis is organized as follows:

- Chapter II provides the background to this thesis work. Underlying concepts fundamental to traditional routing are discussed as well as more recent work in the field.

- Chapter III details the thought process behind building the BE solution for SAAM.

- Chapter IV describes the implementation details of incorporating the developed solution into SAAM.

- Chapter V presents the testing and results obtained in evaluating the solution.

- Chapter VI contains the conclusions and recommendations based on the overall thesis work.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND

A. BEST EFFORT VS. QUALITY-OF-SERVICE TRAFFIC

In today's Internet, traffic is largely of a datagram based form known as best effort [13]. As the Internet evolved from a small network of government and educational hosts into the sprawling internetwork of hosts it is today, the single guiding design principle has been connectivity. New hosts, routers, and domains were continually added in an ad hoc fashion with the only requirement being to maintain connectivity. The rest of the Internet needed a physical and logical connection to these new identities and vice versa. There was no widespread concern for bandwidth, security, QoS, or extensibility of underlying protocols.

Generally, guidelines and technology decisions for the Internet's development were sanctioned and published by non-profit, non-authoritative organizations such as the Internet Engineering Task Force (IETF). Submission to these guidelines was largely voluntary. The one de facto paradigm that emerged was best effort, the only expectation that a user can have of the Internet as a whole. Sites on the Internet are connected and intermediate routers will make a best effort to get each packet to its destination. In other words, connectivity and minimum cooperation in routing are the only things that can be counted upon in today's Internet.

For simple data traffic, where the only concern is having packets traverse from point A to point B *eventually*, this best effort model has proven adequate. However, there are many other forms of traffic that are rapidly growing in volume for which a best effort is not sufficient. 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.

When the only promise a network can make is a best effort, QoS traffic encounters several classic performance problems. A BE network cannot guarantee bandwidth and so high volume traffic applications may find themselves bottlenecked. A

11

BE network will only take the packet from point A to point B with no guarantee of trip time. Obviously, for traffic that is time sensitive, this is unsatisfactory. In a BE network, all packets are treated equally so that, at a congested point in the network, packets will be arbitrarily dropped. For most BE applications running over TCP, these dropped packets will be detected and resent with little or no degradation of performance in the end user application. For some QoS applications, however, exceeding a threshold loss rate (as low as zero in some cases) can degrade performance significantly.

If no expenses are spared, bandwidth and delay requirements can always be met through providing additional network resources. However, this does not solve the problem of delay jitter. For a radio signal in the air, continuous information transmitted at discrete intervals will be received with the same intervals between them. This is important for tightly synchronized information streams among others. Unfortunately, BE traffic over the Internet can be much more erratic than a radio signal and exhibit jitter (also referred to as delay variation) where there may be significantly different delay intervals between successive packets in a data stream from source to destination.

B. BEST EFFORT ROUTING IN TODAY'S INTERNET

In today's Internet, information is broken up into and then transmitted as packets. Appended to these packets are headers, which contain, among other things, source and destination addresses. These packets are then routed to their destinations by routers. Routers perform the task (for BE traffic) of examining a packet's header and then sending it to the next hop in the network based on an entry in the router's routing table.

For all the research conducted and sophistication introduced into this simple model of table lookup, it remains a question of how to populate these routing tables with more intelligent, or optimal, information than a human being's manually configured best guess. Many routers continue the dumb task of examining and forwarding packets while presumably some intelligence, human or machine, is populating these tables to make all the routing decisions happened with the greater good (global optimization) in mind. Some routers are more intelligent, incorporating a routing protocol that enables them to modify their routing tables based on information received from their peers.

The problem of routing can be framed a number of ways. On a static basis, there are underlying graph theory problems. A network is expressed as a graph with nodes and links between nodes. Well-known algorithms can be applied to the graph expression to compute solutions to sub-problems such as shortest path, maximum flow, or minimum spanning tree. The routing problem can also be viewed as one of artificial intelligence where each router must autonomously gather information and make inferences, ultimately building a routing table from scratch. Finally, there is the dynamic aspect of traffic engineering. Traffic engineering generally refers to optimization of changing levels of traffic over routes that are themselves dynamic. For networks, traffic engineering is concerned with the performance optimization of operational networks [1].

To date, no single routing solution has transcended the entire Internet. As a whole, a static solution is difficult to develop because the Internet's topology is ever changing. Similarly, a dynamic solution would require a large amount of computing power to manage the vast number of nodes and links along with a global mechanism for implementing the solution. In theory, either of these solutions is possible (e.g., use of an oracle with infinite computing resources). However, at present, real systems do not have the resources or enforcement mechanisms to do so. No single solution is required as the Internet has naturally grown into a form amenable to a divide-and-conquer approach: that of multiple interconnected domains, or AS's. Within one of these AS's, an intradomain solution is applied that can be uniformly implemented within the domain. Since the entire AS is often known and administered as a whole, very specialized solutions can be developed without assumptions being made about the physical hardware or level of cooperation among the nodes. Between AS's, however, a more robust, interdomain solution has to be applied whereby border routers communicate to share simple reachability information, either informally or through service level agreements (SLA's).

1.   **Intradomain Routing**

a.   Static Configuration

The oldest and least imaginative solution for BE routing within a domain is static configuration, where a router's table entries are manually set and do not change automatically. This requires complete knowledge of the network and the labor intensive

task of logging into each router and through a series of system commands, setting its table to route packets as desired. There is no room for error as a single incorrect digit in a single routing table can logically disconnect the entire network. Further, the network cannot be reconfigured (adding or changing position of nodes and links) without repeating the entire static configuration.

b.   *Distance Vector Protocols*

Of the automated protocols, the first major classification is known as "distance vector." With these simple protocols, a router updates a vector data structure for each of its neighbors, which contains entries for destination address versus a single metric, usually 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 shortest advertised hop count to a destination), updates its routing table entries for each destination. As 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) has enjoyed preeminence during the early growth of the Internet. RIP continues to be widely implemented today despite new proprietary technologies from Cisco and other companies that are superior in some technical aspects. RIP uses a simple hop count metric for route computation and in some implementations, specialized techniques known as split horizon and poison reverse to eliminate loops and hasten convergence after topology changes. RIP's major drawbacks are its hop count limit of 16 and simplistic path weighting (1 hop = 1 distance unit). This has spurred development of more sophisticated distance vector protocols that address these shortcomings. Still, 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.

c.   *Link State Protocols*

The other major classifications of automatic routing protocols are the 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, a router will obtain information from every other router in the network. Second, rather than selecting routes based on a simple metric comparison, a router will build a graphical representation of the network from the information it receives and then use 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 [3]. 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 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 [3]. 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. Finally, OSPF introduces a technique known as traffic splitting where traffic is divided along multiple equivalent (i.e., equal metric) paths.

2.   Interdomain Routing

Intradomain routing solutions handle traffic that is sent and received within a single AS. However, in many networks, particularly those belonging to commercial ISP's, such traffic is not the prevalent form. Most packets are part of network sessions with another host somewhere else in the greater Internet, not in this AS. These packets must first seek out one of their AS's border routers, also known as border gateways. From there, these packets typically travel through one or more other AS's to the Internet backbone and then through another one or more AS's before reaching its destination. As

the Internet is a much more loosely organized collection of hosts, a robust protocol is needed that does not require the trust or cooperation that RIP or OSPF does.

The two most prevalent protocols for interdomain routing are the Exterior Gateway Protocol (EGP) and the Border Gateway Protocol (BGP). The protocols are similar in that they operate through neighboring AS border routers sharing reachability information. Essentially, when a border router receives an outbound packet from its AS, it needs to determine by which neighboring border router that destination is reachable. EGP was the precursor to BGP and is rapidly being phased out due to scalability issues. Specifically, EGP assumes the Internet is still arranged (it once was) in a tree-like topology. EGP fails in those regions of the Internet that are now mesh-like. 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 (64MB, for example, is no longer sufficient without some form of route aggregation).

Reachability information consists of enumerated long lists of sequential domains that are next hops 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 does not prevent selecting a long route when shorter routes exist, only loops. Still, the protocol's current version has controls such as Local Preference and Multiexit Discriminator, which allow for the prioritization of route information. Finally, as the Internet continues its exponential growth and exhausts a router's limited memory, BGP allows route aggregation where it can generalize downstream information. A consequence of this might be lower preference from those border routers implementing the most-specific-first method of route selection.

C. THE STATE OF THE ART

The Internet is still a BE, IPv4 internetwork. Until organizations upgrade their networks to use IPv6 or an overall QoS-based architecture emerges, neither of which are guaranteed to happen, research and development will continue work on designing new solutions and improving existing solutions for BE traffic in the Internet's current

architecture. Even the Internet2 consortium acknowledges that their solution will not replace the Internet [6].

1. Routing Solutions

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.

Shortest path protocols such as OSPF typically use a well-known routing algorithm. Among these, Dijkstra's algorithm is the most popular and more importantly, unexceeded in terms of computational efficiency (scales on order of $n\log n$). The algorithm visits each node in turn and builds a shortest-path tree to all other nodes by sequentially linking the nearest node, updating the computed distance of the unlinked nodes, and repeating continually until the tree is built. In its simplest form, these distances are single, static values that describe something about the links between nodes. In this form, the problem is tractable. However, if the distance values change from scalar to vector or if a single path constraint is applied, the problem becomes NP-hard, or intractable. Obtaining near-optimal solutions in these instances requires the application of heuristics, or settling for some best solution obtained after a computational time limit.

Therefore, mathematical research focuses on how to choose these heuristics. Operationally, research is done on how best to tune a network by adjusting path weights. 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 where a network undergoes large swings in traffic conditions while attempting to correct itself; it never converges on a new solution. Part of this is 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 amongst themselves, 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.

Some of the other routing algorithms developed are sequential selection of different metrics over single-metric algorithms such as Dijkstra's. For example, the shortest widest path (SWP) algorithm is actually Dijkstra's algorithm with inverse bandwidth as the metric. "Shortest" refers to the tie-breaking aspect of selection among equally wide paths. Widest shortest path (WSP) is the reverse. Another method applied in these instances is to develop a new metric that is a function of multiple inputs. For example, a metric may be developed that is a function of a link's bandwidth and delay. Once these two are combined into a single value, the problem becomes tractable with an optimal solution obtainable through Dijkstra's or other algorithm.

Besides modifying the algorithms or their inputs, it is also possible to change how outputs are used. 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 one path. One of the first alternatives to be proposed was a method called equal cost multipath (ECMP) wherein if the routing algorithm found multiple shortest routes of equal cost, then traffic would be evenly split along those paths. One of the main drawbacks of this method is the potential for fragmenting a TCP session where supposedly equal paths actually have delay differentials large enough to cause packets to arrive out of order. One suggestion [2] to correct this is to split traffic based on a hashed IP address to ensure same-session traffic follows the same route. Besides ECMP, other solutions involve splitting traffic to a destination along the next shortest route(s).

2. Management Solutions

Another area of research is the management aspect of BE traffic. Currently, there is no management in the Internet without specialized devices. The Internet's de facto hardware consists of IP routers. The lowest common denominator IP router has no management capability. It simply examines IP headers of packets, performs a table look-up, and forwards those packets as fast as its switch fabric or line speed will allow.

For this reason, TCP/IP ascended to be the combined standard relied upon by hosts communicating over the Internet. TCP performs the end-to-end management of a

18

session that does not exist in the underlying IP protocol. Whereas without TCP, a user can have no expectations about packet loss rate or timing, TCP will step in front of IP and present 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 when needed. This is management on the user end.

TCP also performs an important form of management for the Internet at large known as 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. While at first seeming a luxury, this single mechanism is the only thing protecting the Internet from congestion collapse [4]. Congestion collapse describes the situation where a network's bandwidth is almost exclusively occupied by packets that are discarded because of congestion before they reach their destination.

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 unimpeded and take advantage of the TCP applications' selflessness. Today, the majority of applications run on TCP. 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 [14].

The questionable reliance on cooperation continues to drive the end-to-end versus router-based management debate. End-to-end management is simpler. By refining the TCP protocol and hoping that everyone will use it, congestion can be managed by the hosts at each end of the session. The IETF has even mandated that non-TCP applications be TCP-friendly. That is, those applications that do not use TCP should not offer more traffic load than a similar TCP application would under like conditions. The issue here is fairness. Unfortunately, there is no current way to enforce this mandate. IP routers treat traffic as aggregate and do not currently incur the processing or overhead required to determine if anyone is behaving in an unfriendly manner. Therefore, malicious users can deliver barrages of non-TCP packets with impunity. Even with TCP, a malicious user

can launch redundant sessions or parallel sessions to grab more than his/her fair share despite rate control.

Router control, on the other hand, could potentially enforce some degree of fairness. At the very least, it would require a switch from aggregate scheduling to per-flow queuing. Even then, a flow would have to be reclassified to an IP address-to-address level rather than a TCP port-to-port level to overcome the problem mentioned above. For a router take on management functions, it would have to track state. Most IP routers are treated as being stateless in that they do not keep track of past state. Conversely, stateful router tracks some state over time and requires memory to do this. This state may be data about a specific data flow, class of flows, or type of traffic.

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 the menial task of packet forwarding.

3.   Traffic Engineering

For most of the Internet's lifetime, technicians have tackled problems at the physical level and described those problems in very concrete terms. If there was any engineering involved, it was electrical engineering. As the Internet grew and the amount of data exchange increased, problems were alleviated by a "bigger, better, faster, more" strategy through provision of equipment. The physical layer was the focus and the actual data traffic was treated like weather: something that must be dealt with but cannot be controlled in and of itself.

Recently, a new construct and vernacular have emerged to frame the problem differently. It is called traffic engineering. Network traffic engineering is a more strategic approach than tactical hardware upgrades. The IETF has demonstrated their commitment to this approach by establishing the Traffic Engineering Working Group (TEWG) with its Internet Traffic Engineering Charter.

The TEWG has continued to develop and refine the concept through publication of various Requests for Comment (RFC's) and Internet Drafts. In May of 2001, the TEWG released an Internet Draft entitled "A Framework for Internet Traffic Engineering" in an attempt to solidify some of the concepts and terminology for the subject.

> Internet 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 [19].

Numerous research efforts have focused on traffic engineering and how to incorporate it into existing protocols through extensions or existing routers through management-information-base (MIB) design.

Multiprotocol label switching (MPLS) 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 readily lends itself to this schema through its simple mechanism of packet labeling and label-switched paths (LSP's). In this way, packets can be labeled according to their administrative handle and links can be colored according to their administrative policy. For BE traffic, emphasis has been placed on intelligent labeling and relabeling.

> Traffic engineering is needed in the Internet mainly because current interior gateway protocols always use the shortest paths to forward traffic. Using shortest paths conserves network resources, but may also cause the following problems:
>
> The shortest paths from different sources overlap at some links, causing congestion on those links.
>
> The traffic from a source to a destination exceeds the capacity of the shortest path, while a longer path between these two routers is underutilized [16].

These authors and many others have proposed methods using MPLS to overcome these problems.

Traffic engineering has led to the introduction of another concept: network engineering. In July of 2001, the TEWG published an Internet Draft entitled "A Framework for Internet Network Engineering." Like traffic engineering, network engineering addresses the problem of handling Internet traffic at a macroscopic level. Whereas the focus on traffic engineering was a move away from physical solutions and towards programmed solutions, network engineering is a move back towards the physical in that it examines network provisioning. According to the TEWG, they go hand in hand. In their original treatise on network engineering, the TEWG puts their difference simply: traffic engineering is putting traffic where the capacity is while network engineering is putting capacity where the traffic needs it [19].

III. BUILDING A BETTER BEST EFFORT SOLUTION

A. CURRENT STATE OF SAAM PROTOTYPE

Prior to the work reported in this thesis, SAAM had very little BE capability. In fact, BE traffic was treated as just another QoS level within the SAAM QoS architecture. Contrary to what an end user might expect, SAAM required that all BE traffic be part of a flow and that flow be preceded by a flow request to reserve resources. Once the request was accepted, then the BE traffic would flow through the SAAM region in a manner similar to other QoS flows with an assigned flow ID and path ID. At outbound interfaces, it would be discriminated by the type of service (ToS) bits in its header. This would channel the BE packets through a lower priority outbound queue (IntServ and DiffServ traffic take precedence).

With the flow request model, a number of issues arise. The first is that it is a step back from what is already offered by the Internet for BE traffic. From a user standpoint, the Internet is "instant on" for BE traffic. Once configured with a proper IP address and connected to the Internet, nothing more is required to send and receive traffic. With the flow request model in SAAM, however, the user is burdened with the added requirement of a flow request and the forethought necessary for determining the parameters of that request.

Second, there is the issue of waste. For a revenue-generating QoS flow, there is no waste. The market forces result in a price-driven model in which every request and hence, resource reservation, is paid for at a price agreeable to the service provider and user. Unused reservations continue to generate revenue from whoever reserved them. Unreserved resources retain the potential to be sold in a flow request. Best effort, on the other hand, which is presumably a free or flat fee service, would waste resources within its granted provision. All the while, it could be sharing its unused bandwidth with another BE flow on the same link. Indeed, from an economic standpoint, the entire provision is a waste if there is no cost for BE flows.

Finally, SAAM's requirement of flow requests for all traffic makes it incompatible with the rest of the Internet. In the Internet, an IP router will examine an incoming packet's destination IP address and then make a best effort to forward it, dropping the packet only if the destination is invalid or unreachable. A SAAM router will not recognize, let alone route a packet, unless it has been appropriately labeled according to a previously approved flow request.

Fairness and fault tolerance are two issues that SAAM addresses for QoS traffic. For both IntServ and DiffServ, fairness is a matter of service contract. Better QoS parameters are negotiated at a price. SAAM provides reliable fault tolerance for IntServ flows through advanced rerouting techniques detailed in [15]. For BE however, there is no fairness or fault tolerance. Equal paying customers may encounter substantially different BE performance in a SAAM region and SAAM will make no effort to correct the disparity. As for fault tolerance, a single failure on a BE flow's path will permanently disrupt that flow.

B. REQUIREMENTS FOR A BETTER SOLUTION

1. Security

In any effort to make SAAM more interoperable with the Internet's BE traffic, security must not be compromised. Specifically, if router behavior is modified such that non-SAAM packets are recognized and examined, consideration should be given to the accompanying security problems that may occur such as denial-of-service or any attack that would use unsolicited BE traffic as a vehicle.

2. Light-Weight Routers

"SAAM consists of light-weight routers and a small set of heavy-weight servers" [12]. This is one of SAAM's foundational concepts and is sometimes referred to as the "smart server, dumb router" model. Any new BE solution must not place significant computational or memory burden on the router and destroy this model. This eliminates solutions such as making an OSPF SAAM agent. Router-based algorithms, such as those required by OSPF, are unacceptable. Therefore, computation and bookkeeping should be kept at the server whenever possible and any new router functionality should be fairly simplistic.

3.   Interoperability

Whereas SAAM's current BE flow request system makes it incompatible with the current Internet, the new solution should promote interoperability. Specifically, SAAM routers should recognize and handle generic BE traffic that does not have any QoS labeling. The ultimate achievement of compatibility would be a solution that allows a SAAM region to function as a transit AS, allowing transit traffic based on SLA's with neighboring AS's.

4.   Fault Tolerance

The new solution should afford some degree of fault tolerance for BE traffic. The SAAM server maintains a complete picture of its region and there is no good reason why this knowledge should not be exploited for enhanced fault tolerance. At the very least, the server's knowledge could be used to detour a traffic flow around a known failure.

5.   Fairness

Fairness is one aspect of BE traffic management that the Internet lacks. SAAM is able to provide this with its servers acting as arbiters among competing interests. The new BE solution should tackle the fairness problem along these lines. Specifically, an attempt should be made to maintain fairness both among flows and among users. In this context, fairness is defined as ensuring adverse effects are borne as equally as possible during periods of network congestion.

6.   Adaptive Response Mechanisms

Network congestion can sometimes be avoided altogether if the network adapts to changing conditions by taking preventive or corrective action. Since the SAAM server maintains an accurate picture of its region and is continuously operating, a new solution should incorporate an adaptive response mechanism in order to prevent congestion or alleviate congestion that arises.

7.   Stability

Any time a dynamic solution is considered, stability becomes an issue. Therefore, in designing any type of new intelligence for BE traffic in SAAM that is adaptive and/or

dynamic, stability must be guaranteed. Specifically, time constants for new processes should be considered in order to ensure convergence and prevent over-corrections.

8. Intelligent Provisioning

SAAM has complete knowledge of the characteristics of every link and node in the network. Therefore, an attempt should be made to develop a better solution for BE provisioning than just assigning shortest-hop paths. One area in particular that should be examined is how to make use of unused resources during periods of heavy demand.

9. Scalability

Any new solution should not be based on assumption of a SAAM region having an arbitrarily small number of nodes. Rather, it should allow for growth without internal processes and data structures becoming unmanageable. Specifically, if a solution based on underlying BE flows is considered, then a full interior mesh topology should be avoided if possible.

10. Packet Recovery

For solutions involving edge router discovery, incoming packets should not be needlessly dropped. Rather, an attempt should be made to requeue and retransmit later if possible.

C. INSPIRATION

Significant research was conducted prior to translating these requirements into a design. Many methods of managing BE traffic were surveyed. Those solutions that involved managing BE traffic in a QoS network were examined closely in a search for concepts that might be applied to the SAAM architecture for its BE solution. While many general principles and engineering practices were gleaned as background from multiple sources, the fundamental idea for SAAM's design solution comes from one source in particular.

In 2001, a group of researchers proposed a set of multipath adaptive traffic engineering (MATE) algorithms designed for traffic engineering in MultiProtocol Label Switching (MPLS) networks [2]. MPLS is similar to SAAM in that it is a path-based routing scheme and can be used for QoS management through use of flow labels.

The developers of MATE over MPLS point out that traffic engineering has been attempted before through use of shortest-path algorithms, but such solutions suffer from several limitations:

- Load sharing cannot be accomplished among paths of different costs.

- Traffic/policy constraints are not taken into account.

- Modifications of link metrics to readjust traffic mapping tend to have network-wide effects causing undesirable and unanticipated traffic shifts.

- Traffic demands must be predictable and known *a priori* [2].

The MATE developers then propose their state-dependent traffic engineering mechanism with features to include:

- distributed adaptive load balancing

- end-to-end control between ingress and egress nodes

- no new hardware or protocol requirement in intermediate nodes

- no knowledge of traffic demand required

- no assumption of scheduling or buffer management schemes at nodes

- optimization decision based on path congestion measure

- minimal packet reordering

- no clock synchronization between nodes [2]

Of all the methods for BE over QoS that have been studied, it is MATE over MPLS that provides the foundation for the SAAM solution. Overall, the BE solution for SAAM is designed to incorporate the routing that is common to RIP and OSPF and the traffic engineering accomplished by MATE. The main difference between conventional implementations of these methods and SAAM's implementation is that SAAM centralizes intelligence at the server. Whereas an OSPF router calculates routes on its own, SAAM's servers calculate the routes in a SAAM region. Whereas a MATE router determines congestion with probe packets and makes load balancing decisions by itself, a

SAAM server determines congestion through LSA's and decides for the routers when to do load balancing.

D.   DESIGN

The design of the new BE traffic management solution capitalizes on the SAAM architecture; that is, it relies on specialized deployed agents to perform specialized tasks. Additionally, the server is given a new management module to coordinate the deployment and operation of these agents. Finally, new messages are created to facilitate the specialized communications necessary between the servers and agents.
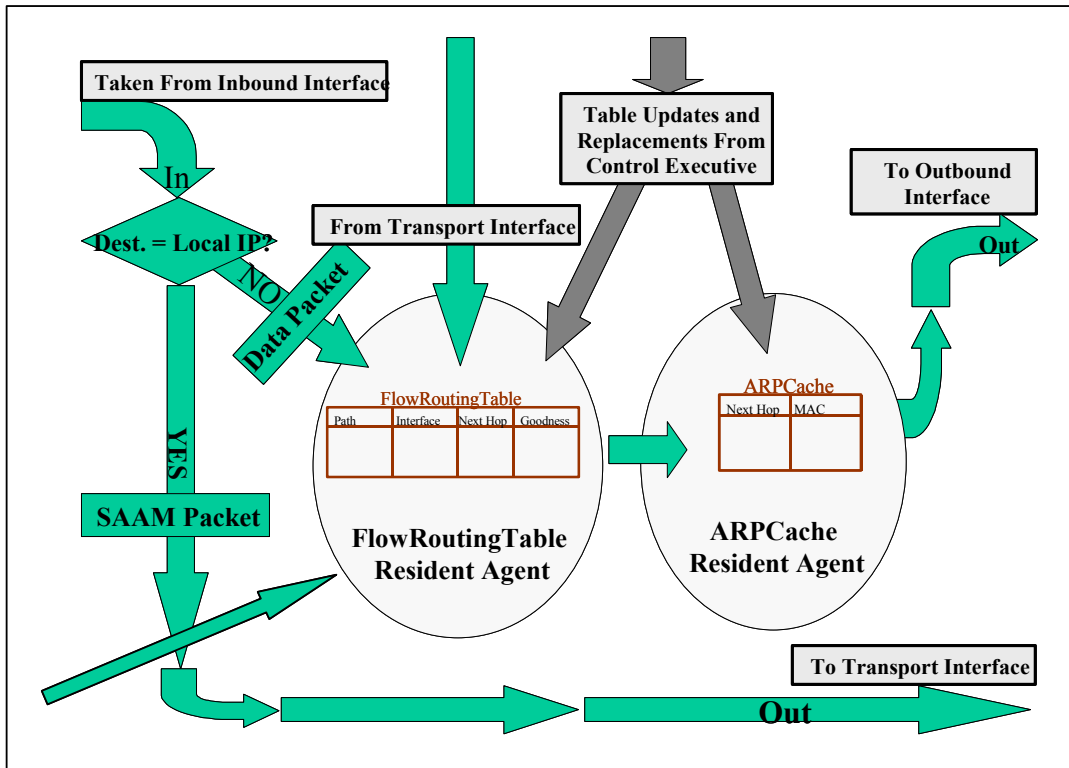
Figure 3.   Old Routing Method (modified from [12])

Previously, only two agents were involved in routing: FlowRoutingTable and ARPCache. As seen in Figure 3, incoming packets are examined to determine destination IPv6 address. If the destination is local (i.e., this router), the packet is sent to the Transport layer.   Otherwise, the RoutingAlgorithm relies on agents FlowRoutingTable and ARPCache to determine the appropriate outbound interface on which to forward the packet.   First, the RoutingAlgorithm calls the FlowRoutingTable to determine the next hop IPv6 address. Based on that address,

another call is made, this time to the `ARPCache`, to determine which outbound interface that address maps to. This solution does not handle BE traffic. As seen in Figure 3, the call to the `FlowRoutingTable` requires that the packet have a flow label containing path ID information, which can only be obtained through flow requests.
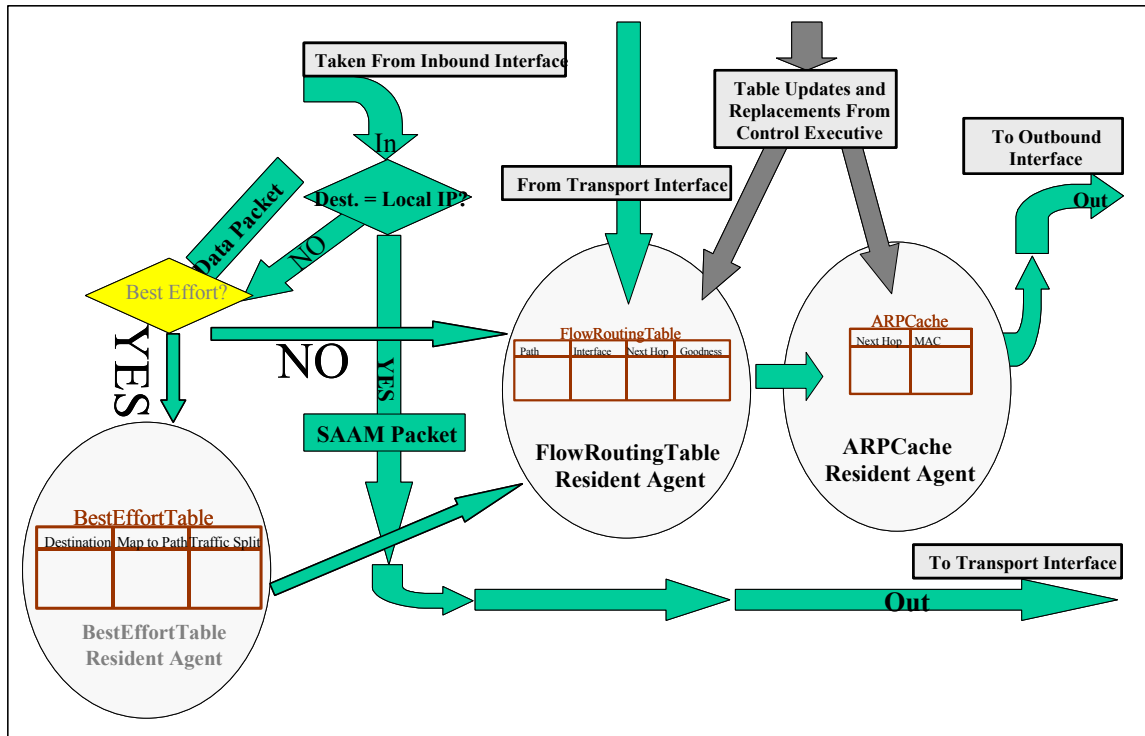
Figure 4.   New Routing Method (modified from [12])

At the router level, a new agent is all that is needed to provide BE functionality. Now, a simple check to see whether or not the packet is unlabeled BE is made before calling the `FlowRoutingTable`. If the packet is unlabeled BE, then the `BestEffortTable` will be called first to obtain the missing flow label (path ID) that the `FlowRoutingTable` needs. This only needs to happen once. When a BE packet receives its label through this process at an ingress to a SAAM region, the label is written into its IPv6 header and used for the rest of its travel through the region. The routing that SAAM uses is path-based rather than hop-by-hop. Essentially, the BE packet gets mapped onto a preinstalled path through the region at the ingress point. That path will terminate at an interface within the region or at one of the border routers. All intermediate SAAM routers will recognize that path by means of a preinstalled entry in their `FlowRoutingTable`.

29
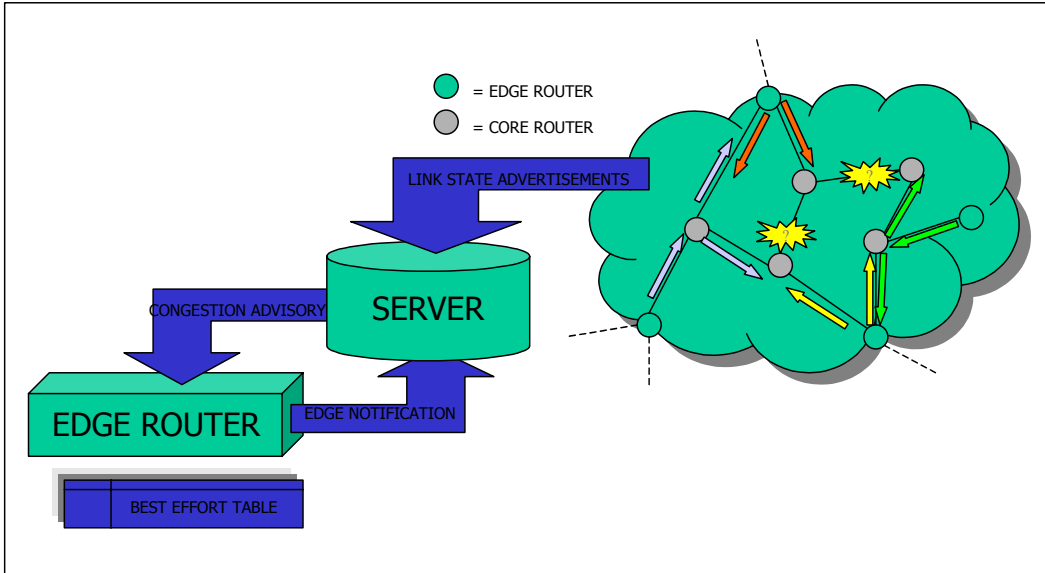
Figure 5.   Server / Edge Router Communication

The solution developed in this thesis requires constant communication between the SAAM server and the deployed `BestEffortTable` agents. This happens through the message exchange shown in Figure 5. LSA's (a message previously developed in [12]) are received from all SAAM routers, keeping the server apprised of network conditions. `EdgeNotification` messages are received from edge routers to key the server to BE topology changes. Finally, the server controls edge router BE handling behavior through `CongestionAdvisory` messages.

All of these new components will now be explained more fully.

1.   Best Effort Table Agent

The `BestEffortTable` agent was developed to be the sole additional agent necessary at a router to handle BE traffic. At its core, the `BestEffortTable` agent is a simple lookup table, similar to the `FlowRoutingTable` agent. However, the `BestEffortTable` has the capability to track multiple entries for a single destination and perform load-balancing among them when directed by the server.

Dest IPv6 Address	Map to Path	Traffic Split
99.99.99.99.3.0.0.0.0.0.0.0.0.0.0.2	92	10
99.99.99.99.3.0.0.0.0.0.0.0.0.0.0.2	82	90

Figure 6.   A Best Effort Traffic Routing Table

30

The `BestEffortTable` works in concert with the `FlowRoutingTable` to route BE traffic. As seen in Figure 6, the `BestEffortTable` by itself has insufficient information to determine a packet's next hop. `RoutingAlgorithm` will recognize BE traffic at the ingress edge route by virtue of it having no flow label or path ID. At that time, it will ask the `BestEffortTable`, if resident, how to handle the packet. The `BestEffortTable`, however, will return a `pathID` based on the packet's destination IPv6 address. With a path ID, the `RoutingAlgorithm` will then be able to route the packet like regular QoS traffic by referring to the `FlowRoutingTable` for an actual next hop address and then its `ARPCache` for the outbound interface.

The previously unlabeled packet is now labeled in the manner for use by the SAAM routers. The packet will travel through the region possibly using the same path shared concurrently by QoS flows. It will be discriminated against in priority queues, but this is a preexisting mechanism designed to meet QoS traffic bandwidth guarantees. It only needs special handling at the single ingress edge router. This traffic would have originated from either an application on the edge router or from outside the domain if the edge router is also a border router. Either way, it follows a path to an egress edge router in the SAAM region, which, again, may host a receiving application or be a border gateway to outside the domain.

a. *Destination Management*

The `BestEffortTable` manages traffic on a per-destination basis. Whenever the `BestEffortTable` receives new BE routes from the server, it will either recognize a new destination to manage that it hasn't seen before or it will add those routes to its data structure for that pre-existing destination. The `BestEffortTable` has room for a full complement of spare routes for each destination. The current implementation uses just two active routes to each destination, so a 4-route array is used to hold the active and spare routes. At any given time, the `BestEffortManager` will load balance between its two active routes. If sent a third route, that route will be added to the table with a split of 0% for that destination. If a fourth route (which may be identical to the third in some cases) is then sent, the `BestEffortManager` will recognize this as the alternate path for a congestion bypass. Consequently, the first two

31

routes will be set to 0% split, while the previously sent third route is set to handle 100% of the traffic as the new primary route. The first two routes are retained and remain in the table. However, if the server sends more routes for this destination, the table wraps around and these entries are written over.

b. Load Balancing

The `BestEffortTable` receives multiple entries to a destination in order to perform load balancing among different parts of the network. It accomplishes this through the dual behaviors of redirection and reversion. Initially, `BestEffortTable` maps all traffic to a destination along a single primary path. This path is usually shorter and/or wider than the alternate paths by virtue of the performance-based path-finding algorithm the Best Effort Manager uses for primary paths. Therefore, it is used preferentially for performance reasons. Once the `BestEffortTable` is notified by the server of congestion towards a destination, it begins redirection of traffic. Redirection involves iteratively shifting traffic to alternate paths.
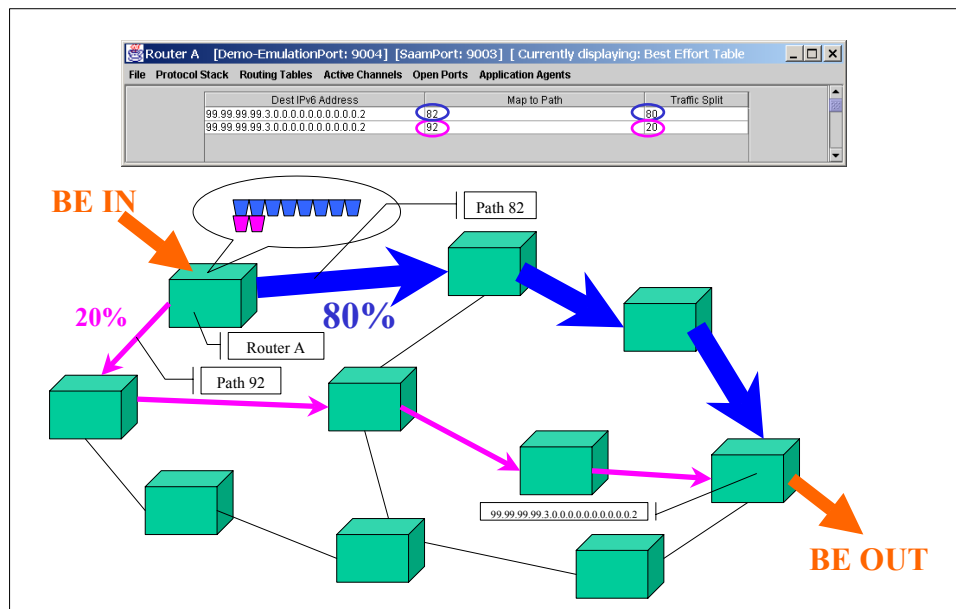
Figure 7.   Multipath Load Balancing

When the server notifies the `BestEffortTable` that the congestion has cleared, the `BestEffortTable` will initially hold the current split. Then, it will begin reversion of traffic. This involves iteratively shifting traffic back to the primary path, which is preferential for performance reasons.

Redirection and reversion happen with different frequencies. Principally, redirection is rapid in order to avoid a sustained congestion condition. Reversion is less urgent as the only concern is gradual optimization during conditions that are already acceptable and so happens less frequently. Choice of redirection and reversion intervals is discussed later in this section.

The `BestEffortTable` has an additional method of load balancing that is not a programmed behavior as redirection and reversion are, but rather a phenomenon resulting from the manner in which it installs new table entries. Whenever the `BestEffortTable` receives a new complement of routes toward a destination, it resets the split for that destination to 100/0. This allows the server to do something that will be termed here as switchback. By resending the same entries that the `BestEffortTable` already has, the server can cause the `BestEffortTable` to instantly shift all of its traffic back to the primary path. Otherwise, this would only happen through numerous reversion intervals.

c. Fault Tolerance

While primarily used for load balancing, the redundant paths are also used for fault tolerance. When the server notifies the `BestEffortTable` of a failure in a path to a destination, the `BestEffortTable` will immediately switch all traffic onto the surviving paths.

2. Best Effort Manager

The `BestEffortManager` is a new component in the SAAM server which handles the management of BE traffic from the server end. The `BestEffortManager` relies on three main programmed hooks in the server code to function during operations. The first is a call whenever the server discovers a new edge router. `BestEffortManager` updates its bookkeeping to track the new router and any attendant BE traffic. Next is a call whenever the server is receiving an updated link state advertisement from a SAAM router. `BestEffortManager` will examine the reported loss rate for BE traffic and respond accordingly. Lastly, whenever a network failure occurs, `BestEffortManager` is notified immediately for fault tolerance purposes.

`BestEffortManager` always functions in one of two modes: reactive monitoring or proactive monitoring. Within these modes, its operations consist of maintaining a BE topology, deploying paths for BE traffic, sending congestion advisories, and tracking statistics.

a. Best Effort Topology Maintenance

One key to the overall BE traffic management solution is to be able to distinguish between edge routers and core routers. It would be simpler to consider every SAAM router as a potential sender and receiver of BE traffic. This necessitates deploying enough paths to create a full interior mesh topology because traffic can originate and end anywhere. This approach is not only wasteful; it hurts the overall scalability of SAAM. First, not all of these paths are necessary since many, if not most of the routers, will not be origins or destinations of BE traffic. Second, a mesh involves deploying $2 * N * (N - 1)$ paths for a region of N routers which is on the order of N^2 growth as new routers are added. N^2 growth can portend that a solution will not scale well; if N increases by an order of magnitude, then the solution resources will increase by two orders of magnitude.

Therefore, the `BestEffortManager` carefully tracks and manages just a subset of the overall SAAM topology, which is the BE topology. It builds this topology from scratch through a process of edge router discovery. Every router is a core router initially The SAAM server is notified if a router previously regarded as a core router needs to be promoted to edge router to handle BE traffic. The SAAM server is also notified when a regional interface address appears in the IPv6 destination field of BE packets. In both cases, `BestEffortManager` updates a BE topology comprised of these routers and interfaces. The BE topology is not just a subset, but also an abstraction of the complete topology maintained by the server. Whereas the server's topology is comprised of nodes and links between nodes, the BE topology is comprised of nodes and paths between nodes. These paths are logical representations of nodes and links not specifically included in the topology.

In the strictest sense, this BE topology still scales as N^2, just as a mesh topology would. However, N now maps to just edges rather than all nodes. In the

34

Internet, edges are comprised of routers serving as an ISP's point of presence or a border gateway. Generally, these routers comprise less than half of the router population. Therefore, a mesh consisting only of these routers significantly reduces the size of the topology to be managed, which enhances scalability.

The topology is built by connecting all BE origins to all BE destinations through two paths. Origins are edge routers. Destinations are interfaces addressed in BE packets. From this point, the topology is kept up to date in event of new nodes or interfaces, new path deployment, or path failures.

b. Reactive Monitoring

Reactive monitoring is the default mode of the `BestEffortManager`. During reactive monitoring, the `BestEffortManager` monitors link state advertisements for 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 single `BestEffortTable` agent out in the SAAM region.

When congestion is detected between two BE nodes, `BestEffortManager` will notify the `BestEffortTable` agent at the source router to begin load balancing in an attempt to resolve the congestion locally. If it becomes apparent that load balancing will not resolve the congestion, then the `BestEffortManager` will initiate a procedure called congestion bypass. This entails the server deploying new paths to that `BestEffortTable`, which bypass the congested region of the network. The old paths expire for 30 minutes and will not be reused for BE traffic during that time. If no available paths remain for congestion bypass, then the `BestEffortManager` declares the network to be globally congested and initiates global congestion resolution procedures.

c. Proactive Monitoring

`BestEffortManager` shifts to proactive monitoring during periods of global congestion. Proactive monitoring involves continuous statistics tracking and deliberate actions to effect fairness during times of resource contention. `BestEffortManager` uses statistics to determine which BE traffic is being inordinately penalized and which BE traffic is receiving more than its fair share.

After this, `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 less than one standard deviation less than the mean packet loss rate. Similarly, the "poor" are those experiencing a packet loss rate greater than one standard deviation above the mean packet loss rate.

In robbing from the rich, `BestEffortManager` will reclaim that flow's least widest path, freeing those resources to alleviate congestion in more heavily congested flows. In giving to the poor, `BestEffortManager` first goes through the graveyard of expired BE paths (from congestion bypass procedures) and reclaims those paths which are now congestion free prior to their 30 minute quarantine. Then, `BestEffortManager` determines which of two methods might increase a flow's available bandwidth: switching back to the primary path or deployment of a reclaimed path. If it is switching back to the primary path, `BestEffortManager` instructs that `BestEffortTable` agent to initiate a switchback. If it is deployment of a reclaimed path, then `BestEffortManager` will deploy reclaimed paths to increase bandwidth and alleviate congestion. In either case, if an action is taken, `BestEffortManager` will wait one local resolution timeout to allow the problem to resolve before taking further action. If no action is possible or if all loss rates fall within a single standard deviation of the mean loss rate, then `BestEffortManager` continues with proactive monitoring until either global congestion disappears or a flow qualifies as rich or poor.

3. Routing Algorithms

Several new routing algorithms have been developed which are designed specifically with to include BE management capabilities. Specifically, these algorithms encompass load balancing, congestion bypass, redundancy, fault tolerance, congestion avoidance, and performance.

Each of these routing algorithms is based on linear search and path comparison of the server's path database and applying selective filters. The first-shortest-path (FSP)

algorithm developed earlier in the SAAM project ([5]) did not meet the needs of the BE management scheme discussed here.

a. Shortest Widest Path (SWP)

SWP selects 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 among those is chosen.

SWP is the algorithm used to deploy initial primary paths for BE traffic flows. Barring any congestion and assuming sufficient network provision, this is the algorithm of choice since bandwidth is the number one criteria. Bandwidth is more valued for BE traffic due simply to the prime goal of congestion avoidance. Delay is not a concern for BE traffic since if it was, that traffic should have been put into a QoS flow. The only concern for BE traffic is that it gets to its destination with minimum congestion along the way. Congestion is best avoided by selecting wide paths from the beginning.

b. Shortest Widest Most Disjoint Path (SWMDP)

SWMDP selects the shortest of the widest of the paths most disjoint compared to some reference path between two nodes. It does this by first finding the paths that have the least nodes and links in common with some reference path between two nodes. Ideally, a path will be found with nothing in common with the reference path except the first and last node. If multiple paths are found equally most disjoint, then the widest path is selected. For equal widths, the shortest path is selected.

SWMDP is the algorithm used to deploy all alternate paths (both initial provision and congestion bypass). Regardless of the algorithm used to deploy the primary path for a given node pair, SWMDP will be used to select the alternate. The reason for this is that alternate paths are valued above all else for their use in load balancing and fault tolerance. Load balancing is more effective between two paths with no common node or link between source and destination. Indeed, if the two paths did share a link, and congestion were to occur on that link, then load balancing between the two paths would do nothing to combat that congestion. Secondly, less fault tolerance is provided by an alternate path that shares nodes and links with the primary path. Fault

tolerance dictates that the paths be as independent as possible so that a single failure might not disable both.

c. *Shortest Widest Least Congested Path (SWLCP)*

SWLCP selects the shortest of the widest of the least congested paths. It does this by first determining the least congested path in terms of packet loss rate. If there are multiple paths of equal congestion, it chooses the widest of those. If necessary to choose among equal widths, the shortest path will be selected.

SWLCP is used when the `BestEffortManager` performs congestion bypass. If congestion has been detected in the region, then lack of congestion becomes the number one priority in choosing paths in order to bypass the congestion that exists. Whereas prior to congestion detection, congestion levels in the region are ignored, congestion now becomes the main criterion. Ideally, this will result in the under-utilized network periphery being used in periods of peak demand, easing the load that tends to develop on the interior critical links.

4. Traffic Splitting

In order to accomplish load balancing, one of the main features of the `BestEffortTable` agent, some method of splitting traffic must exist. That is, if the current traffic split is 30/70 to a primary and alternate path to destination, how is the 30/70 split actually accomplished? The simplest solution would be round robin where three packets would be sent on the primary path and then seven on the alternate. However, this method can incur performance penalties for applications requiring packets to arrive in the order they were sent, since the primary and alternate path may have significantly different delay times. TCP, for example, will hold packets arriving on the receiving end prior to presenting them to the Application layer. This process takes time and processing power.
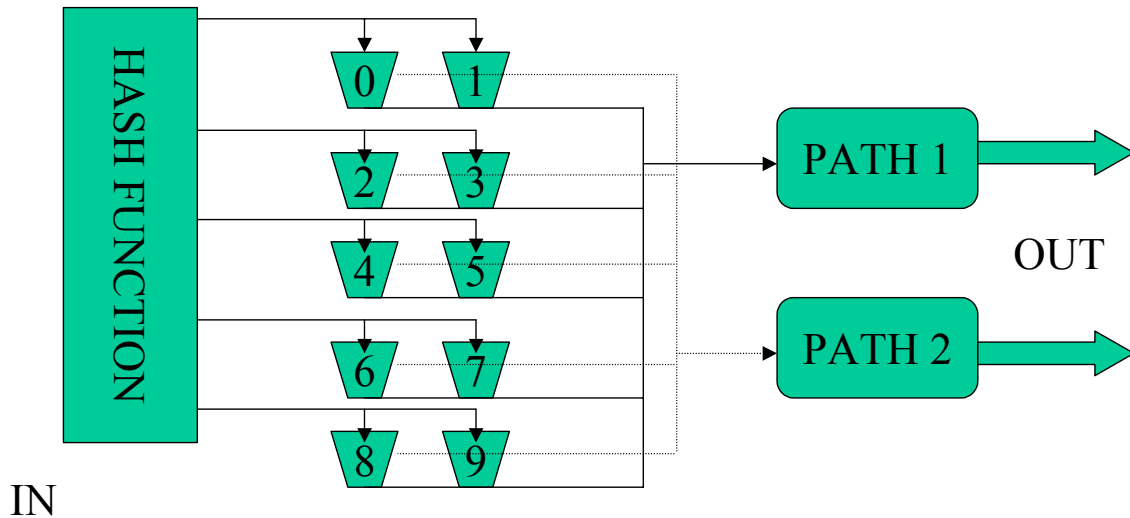
Figure 8.   Hashing Traffic into Buckets

For BE traffic management in SAAM, a better solution has been developed based on the suggestion in [2]. Packets are sorted into ten buckets based on a hash of their source IP address through a modulo-N function (N=10 in this case). These buckets are then shifted discretely so that a single TCP session, for example, will never know it has been rerouted. It certainly will not be split. Either the entire session will proceed on the primary path or the alternate path, never a split between the two. This means that a 30/70 split might not really be 30% and 70%. If, for example, that BE traffic flow is comprised of two equal traffic sessions between two host pairs, then it must be either 100% on one path or 50%/50%.
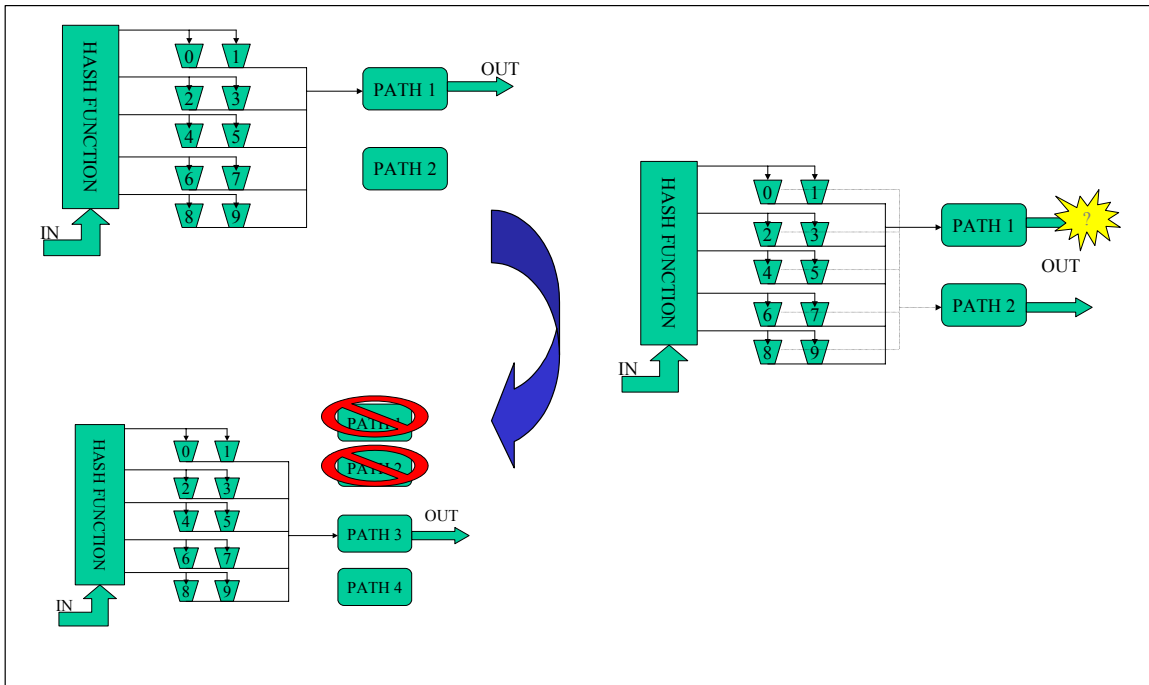
Figure 9.   Load Balancing By Bucket Path Mapping

Hashing traffic into buckets results in a different behavior of the system. When the `BestEffortTable` is shifting traffic during load balancing, it is not shifting 10% at a time, but rather a bucket at a time. If no traffic is in a bucket, then shifting a bucket has no effect. Conversely, if all the traffic is in a single bucket, shifting that bucket shifts all of the traffic. With similarly sized buckets, tunability is limited only by number of buckets and load balancing is realized. If one bucket is much larger than the others, however, then the solution allows for no more balancing than a single path solution would since practically, this has the same effect of switching a path for an entire flow.

5.   Control Timing

In order to build a stable solution, consideration must be given to timing of the corrective controls. On one hand, it is useless to act too quickly if the effect of the last corrective action has not been fully assessed. Another corrective action might be unnecessary. On the other hand, it is desirable to correct problems as quickly as possible without needless hesitation. Therefore, there is always a tradeoff between rushing to correct a problem and waiting to fully evaluate the effect of previous corrections. Over-correction can lead to instability where a cycle begins of correcting a problem only to create a new one in the other direction that needs further correction.

40

The BE traffic management solution for SAAM presents another concern through its use of distributed adaptive load balancing. With each of the routers acting independently to correct congestion, is it possible for destructive interference to occur? This problem has been noted in other solutions (such as OSPF) where all routers immediately unload a congested route and move traffic elsewhere, creating a new congested route.

SAAM's BE solution mitigates this in three ways. First, while SAAM's routers act autonomously during load balancing, they are not synchronized. Therefore, their corrective actions are usually staggered and less likely to result in network-wide route flapping. Second, SAAM's routers do not shift an entire traffic flow at once, but portion by portion in incremental stages. Lastly, if global congestion occurs, SAAM's server takes over, ending distributed autonomous operation and instituting centralized micromanagement.

The authors in [2] showed that stability can be guaranteed when node pairs operate asynchronously in a MATE scheme. SAAM's routers are not as asynchronous as MATE assumes, but neither are they synchronized. Two effects govern their degree of synchronicity. First, if the server instructs load balancing to take place on multiple routers at the same time, the actual time of initiation will only differ by their delay time from the server. Routers at the same distance from the server will be synchronized. This synchronism is mitigated by the fact a node pair interrupts its traffic shifting whenever that traffic flow ceases, even for a single packet interval. This would cause the previously synchronized pairs to become asynchronous. Should the pairs remain synchronous due to continuous traffic, however, potential for instability is mitigated by the fact that the autonomous load balancing at the router only proceeds in one direction at short time intervals: primary to alternate. Only the server can move traffic from alternate to primary at short intervals. This happens by switchback and the server never initiates more than one switchback at a time. Together, these design characteristics should minimize the chance of the SAAM region failing to converge for a given network condition, which would be greater if the routers were allowed to balance in either direction. At longer intervals, the router will attempt to shift back to primary, but only

one bucket of traffic at a time. If such a shift causes the congestion to reoccur, the router immediately resumes the previous traffic split.

When SAAM's server takes control during global congestion, stability is not an issue. There is congestion everywhere and SAAM's main concern is to ensure fairness among contending flows. Still, convergence is attempted in this case by the server waiting at least one local resolution timeout between sets of corrective actions. Table 1 shows the timing used for BE traffic controls.

Auto-configuration Cycle = 1 ACC	200 ms (nominal)
Number of Traffic Buckets = NB	10
Load Balancing – Redirection = 1 ACC	200 ms
Load Balancing – Reversion	30 min.
Congestion Bypass = NB ACC's	2 s
Path Expiration	30 min.
Failure Detection / Response = 2 ACC's	400 ms
Local Resolution Timeout = NB ACC's	2 s

Table 1.   Best Effort Traffic Control Timing

a.   *Auto-configuration Cycle Time*

The entire BE traffic management timing scheme depends upon the auto-configuration cycle time of the SAAM region. It is this cycle time that governs the exchange of periodic "I'm alive" type messages. Added to these messages are the link state advertisements (LSA's) from each router. The auto-configuration cycle time is a tunable parameter generally set to be greater than the round trip time between the server and most distant SAAM router. A lower setting causes auto-configuration cycles to initiate without evaluating information from LSA's in the previous cycle as well as causing more control channel overhead in the network. A higher setting makes the SAAM server less responsive to changes in network conditions. For the current SAAM prototype, auto-configuration cycle time is 200 ms. This means that the SAAM server is

apprised of the complete network condition every 200 ms. It also means that, at any given time, the information the SAAM server has about its region may be up to 200 ms old.

b. Redirection Interval

Once congestion is detected, `BestEffortManager` notifies affected `BestEffortTable` agents to begin load balancing, which initially involves redirection. During redirection, the `BestEffortTable` agent shifts one bucket of traffic at a time from the primary to alternate path each redirection interval. The redirection interval is equal to the auto-configuration cycle time. The reason for this is that the server knows the status of that congestion no sooner and no later than the next LSA, which is sent once per auto-configuration cycle. Therefore, the `BestEffortTable` agent performs redirection at the same interval until notified by the server that congestion has cleared.

c. Reversion Interval

Once congestion has cleared, traffic that has been redirected remains on the alternate path. This is not desirable in the long term for two reasons. First, as mentioned earlier, the alternate path may not have the performance characteristics that the primary path does. Second, there is less room to maneuver in the event of future congestion in the region. Ideally, when congestion occurs, every node pair is on its primary path and collectively, the congestion will resolve by one or more routers redirecting traffic to its alternate paths. However, if too many node pairs are already routing most traffic on their alternate paths, then load balancing may be ineffective and the server will have to perform congestion bypass.

In the case of redirection, where congestion is being combated, time is of the essence. This is not the case with reversion, where the goal is to gradually restore the primary path topology and gain minor performance increases. Indeed, if reversion were to happen too quickly, traffic may be reverted back to an area where congestion has not subsided. Therefore, the reversion interval of 30 minutes was chosen such that network conditions may have appreciably changed. Graphs in [10] show that wide-area Internet traffic patterns only change appreciably over time periods on the order of hours. For

SAAM deployment to a region smaller than the greater Internet, the reversion area may be tuned to match that region's expected usage patterns. Smaller reversion intervals may allow BE traffic to resume on high performance links earlier, but may also cause unnecessary cycles of congestion if the previous congestion has not cleared. As discussed earlier, stability becomes a concern as reversion interval time approaches redirection interval time. The sole consequence of longer redirection intervals is unnecessary lingering on lower performance alternate paths while the primary path is cleared. If congestion results on the alternate during this period, however, the server will initiate a switchback to place all traffic back on the primary path.

d. Congestion Bypass Time

On detection of congestion, `BestEffortManager` will immediately notify `BestEffortTable` agents on affected routers so that adaptive distributed load balancing will begin. `BestEffortManager` will allow load balancing to run its course in the hope that alternate paths can relieve the primary paths enough to eliminate congestion. However, if the deployed primary and alternate paths are both congested, then `BestEffortManager` will have to perform congestion bypass. The server knows this is necessary after a time equal to the number of buckets multiplied by auto-configuration cycle time. By this time, all traffic would have been redirected to the alternate path for that node pair. So, for the current prototype, the `BestEffortManager` would wait two seconds (10 x 200ms) and then perform congestion bypass if the node pair is still congested.

e. Path Expiration Interval

When a region of the network becomes congested, all paths traversing that region are affected. `BestEffortManager` will begin to unload this part of the network and move traffic to unused periphery. It accomplishes this through a system of path expiration. Once congestion bypass is performed on a node pair, that pair's old primary and alternate paths are marked as expired and will not be reused for BE traffic until the path expiration interval elapses, which is 30 minutes. In the near term, this prevents subsequent congestion bypass attempts from placing an alternate path in the congested region. In the short term, it allows network usage of that region to subside

44

prior to adding more BE traffic. Finally, these expired paths become available for selective early reuse in the fairness procedures of `BestEffortManager`.

f. *Failure Detection / Response Time*

As explained in [15], the SAAM server will detect a failure in two auto-configuration cycles, which is 400 ms for this prototype. The response time is the time it takes for the `BestEffortManager` to inform affected `BestEffortTable` agents of the failure, which happens immediately after detection. In the interim, packets traveling on the affected path will be lost. Once that `BestEffortTable` receives notification of the failure, all traffic will be transferred to the unaffected paths.

g. *Local Resolution Timeout*

During global congestion, `BestEffortManager` performs global congestion resolution and proactive monitoring. This involves any number of positive actions as detailed above in the rob-from-the-rich and give-to-the-poor fairness scenarios. The local resolution timeout is equal to the congestion bypass time of two seconds for the same reasons. It allows local resolution to proceed before taking any global action. In this case, the local resolution in question is the new bandwidth that has been taken from the rich and/or given to the poor. The timeout is also used as a delay between initiation of proactive monitoring and any server action to combat congestion. This is necessary since it may be that local resolution initiated just prior to the global congestion condition will be enough to resolve the congestion. This timeout also adds to the stability of the global congestion resolution scheme by preventing hasty action. The network is allowed to settle after each global congestion resolution action is taken; only then are the rich and poor classes recalculated for consideration of further action. The statistics for calculating rich vs. poor are based on inputs that are exponentially smoothed, not instantaneous measures.

6. Messages

A number of new messages are necessary to support the coordination of building the BE topology and communication between `BestEffortManager` and `BestEffortTable` agents.

a. *Edge Notification*

1	2	16
Message Type	**Message Length**	**Edge Router ID or Destination Interface Address (IPv6)**
70	16	X.X.X.X.X.X.X.X.X.X.X.X.X.X.X.X

Table 2.   Edge Notification Message Format

The BE topology is built through a process of edge router discovery. When the SAAM region first stands up and the overall topology is built through auto-configuration messages, all routers are assumed to be core routers that handle only labeled QoS traffic. `EdgeNotification` messages are used by routers to alert the server of either a new router or a new BE destination address. In the first case, a core router will send an edge notification on receipt of its first BE traffic. This is a request to be recognized as an edge router. If it is already an edge router, but it receives a BE packet with an unresolvable destination address, it will send an `EdgeNotificiation`, this time with the field containing the unknown IP address; this constitutes a request for a new table entry. Table 2 shows the message format for `EdgeNotification`. The size, name, and value of each message field are presented respectively in the three rows.

b. *Best Effort Table Entry*

1	2	16	4	4	4
Message Type	**Message Length**	**Destination Address (IPv6)**	**Path ID**	**Serial Number**	**Split**
69	28	X.X.X.X.X.X.X.X.X.X.X.X.X.X.X.X	X	X	X

Table 3.   Best Effort Table Entry Message Format

The `BestEffortManager` exercises overall control of all BE traffic through deployment of table entries for the `BestEffortTable` agents. These table entries are sent in the message format shown in Table 3. The destination address field carries those addresses through which the `BestEffortManager` has become aware by edge notification messages. Path ID is determined by the server and corresponds to the path that traffic will map to in that router's `FlowRoutingTable`. The serial number field is reserved for future use. The split field is not currently used by the server. Split is set on receipt and further adjusted autonomously by deployed `BestEffortTable`

46

agents. Future designs may allow the `BestEffortManager` to send table entries with an initial split setting.

c.   *CongestionAdvisory*

1	2	4	1
Message Type	**Message Length**	**Path ID**	**Color Code**
71	5	X	Red=1 / Yellow=2 / Green=3

Table 4.   Congestion Advisory Message Format

`CongestionAdvisory` messages are sent from `BestEffortManager` to `BestEffortTable` agents. The message contains two pieces of data. The first is the path ID. `CongestionAdvisory` messages are always sent in reference to a particular path, though in some cases, the `BestEffortTable` agent will act based on the destination address the message refers to. The second piece of data is the message's color code, which describes the nature of the advisory.

- `CongestionAdvisory`-RED means that the path in question has failed. The `BestEffortTable` agent is to reroute all traffic to the remaining path and await redundancy restoration.

- `CongestionAdvisory`-YELLOW means that the path is congested. The `BestEffortTable` agent is to begin load balancing immediately.

- `CongestionAdvisory`-GREEN means that congestion has cleared on all paths toward a destination. The `BestEffortTable` agent is to terminate load balancing and maintain the current split ratio.

E.   STATE ANALYSIS

The BE management system here can be further explained through a state transition analysis. Several states have been discussed above that are being tracked in the BE management solution and used to make decisions. First, the edge routers are tracking destinations, which are viewed as either congested or congestion-free with a current traffic split. Servers are tracking paths, labeling them red, yellow, or green based on their congestion condition. Servers are also tracking edge router pairs in noting their paths

47

available, congestion condition, and time in local resolution procedures. Finally, the entire SAAM region is viewed as either being globally congested or locally resolvable.

1. Single Node Pair Management
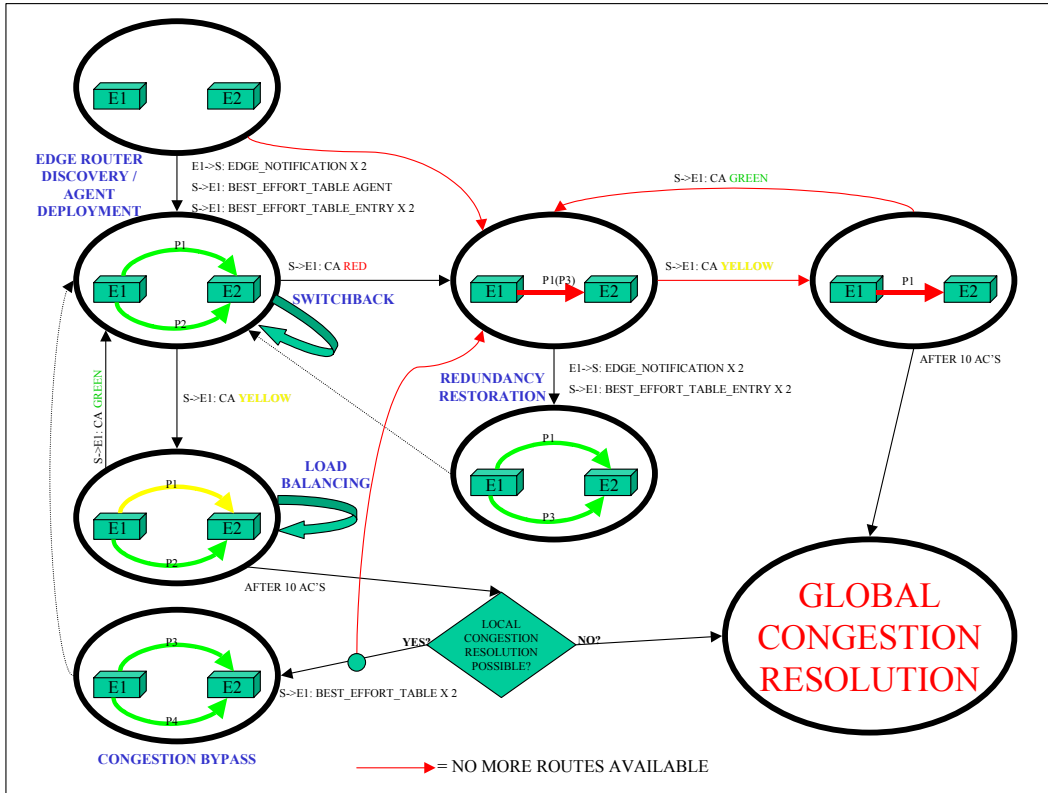
Figure 10.   Best Effort Management State Diagram

Figure 10 show the possible states that exist in the management between `BestEffortManager` and a single pair of edge routers from a server's perspective. E1 and E2 are a pair of edge routers with P1-P4 being possible paths between them. As discussed before and shown in the figure, the existing states and their transitions depend on path congestion condition, path availability, and time. Ideally, load balancing, congestion bypass, redundancy restoration, and/or switchback will always be successful in restoring a state of congestion free flow between the node pair through multiple paths. However, as shown in the figure, there are two ways to reach the undesirable end state where global congestion is declared.

2. Destination Management
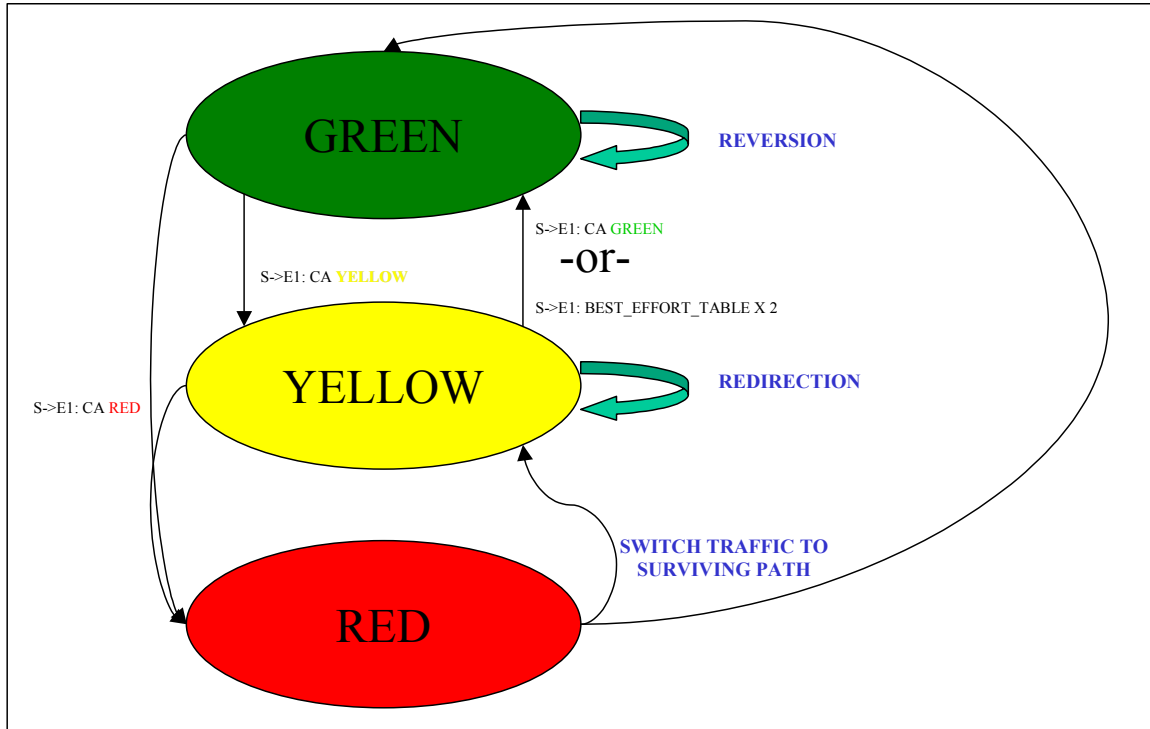
Figure 11.   Destination Management State Diagram

On the router end, BE traffic is managed on a per-destination basis. As shown in Figure 11, destinations are colored either red, yellow, or green based on messages from the server. During conditions green and yellow, the `BestEffortTable` is constantly shifting traffic. During condition red, all table entries are modified so traffic is carried by a surviving path and then the previous condition is resumed. Therefore, unless the server sends new table entries in this case, reversion and redirection accomplish nothing other than modifying the Split field in otherwise identical table entries.

3. Path Management
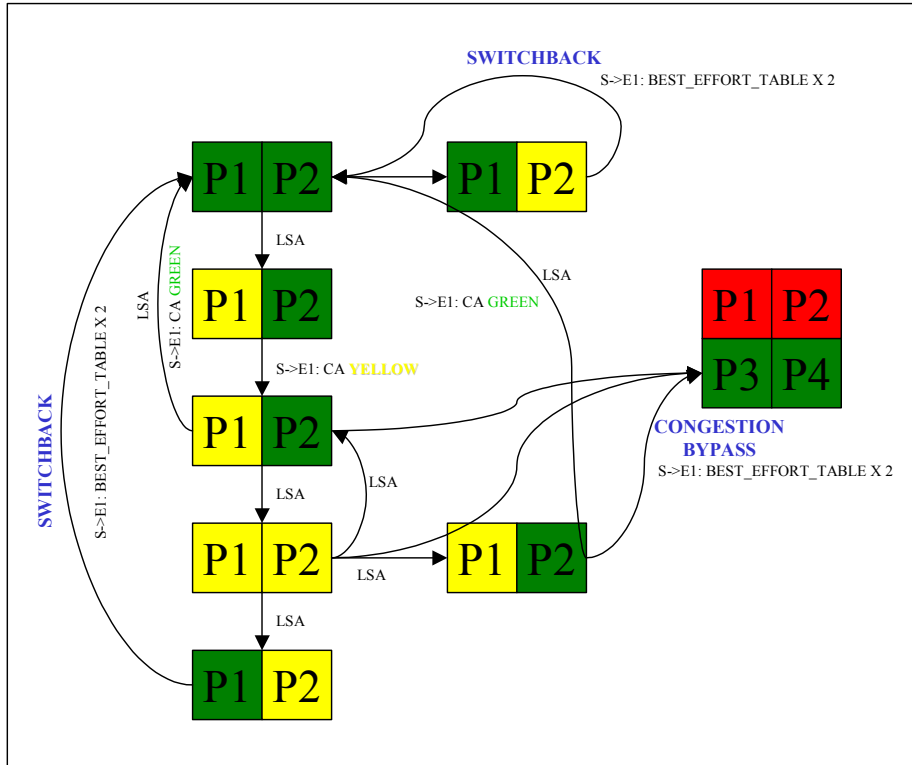
Figure 12. Path Management State Diagram

Figure 12 shows the BE path management state diagram for the general case of no path failures and two routes available for congestion bypass. P1-P4 are paths between a pair of edge routers. Paths are colored green, yellow, or red to indicate whether they are congestion-free, congested, or expired, respectively. LSA's drive the state transitions causing the server to take actions. Fundamentally, whenever congestion occurs on the primary path P1, congestion bypass will be performed unless that congestion clears within the local resolution timeout time. Switchback is performed in two cases: congestion starts on alternate path P2 while P1 is congestion-free or congestion clears on primary path P1 while P2 is congested.

F. DESIGN TRADEOFFS

The BE traffic management solution described here is one of an infinite number that could have been chosen for SAAM. Throughout the course of development and design, many tradeoffs have been made which require explanation.

1. Maintaining BE Link Provision

Recently, SAAM added a new capability called inter-service borrowing, a feature detailed in [2]. Inter-service borrowing allows IntServ and DiffServ, SAAM's QoS traffic, to borrow from each other if necessary to meet QoS demand. Such an idea could have been extended to BE traffic. Here, it was decided that rather than focusing on adding complexity to the inter-service borrowing method, the focus would be on how to maximize use of the existing provision. This solution's method of moving traffic to underutilized periphery during periods of high demand increases the potential amount of traffic that SAAM can handle within its 15% allotment for BE. Philosophically, QoS management is the primary function of SAAM and temporary borrowing of QoS bandwidth for BE is contrary to that.

2. Degree of Management Centralization

SAAM's architecture consists of lightweight routers and heavyweight servers [12]. Therefore, a mandate already exists for general division of labor. Still within these bounds, decisions must be made about where to place each function of a management scheme. Keeping the routers lightweight becomes the main constraint. The design proposed in this thesis, therefore, attempted to place the majority of added burden on the server end. Indeed, the software solution, while written in Java, aimed to be easily translatable to a lower level language or even hardware. For the functionality required in the edge router proposed in this thesis, the router needs just two extra data structures (best effort table and destination status table) and some simple behavior that is driven by time stamp checking (redirection and reversion).

3. Management Focus

The solution designed in this thesis does not involve the server as much as it potentially could. In the case of distributed adaptive load balancing, the server is unaware of the current traffic splits in node pairs throughout the network though this is information that could be reported. It only knows which pairs are currently load balancing and which pairs are not. This is an issue of management focus. The reasons that this solution chose the standoffish approach are threefold. First, if the server is going to manage all load balancing down to the bucket for each node pair, significantly more communication would be required. Second, the job of the server becomes much more

51

intensive in that it would now have to make a computation-based decision every auto-configuration cycle for each node pair in load balancing. Finally, the solution is no longer distributed and adaptive for the case of localized congestion. Instead, it becomes a centralized intractable problem requiring the same heuristic that is applied with the distributed approach (shift traffic to a congestion-free path). Granted, there is room to include a minimum interference algorithm at this point, but being able to compute a near optimal solution in the time of one auto-configuration cycle is doubtful. Overall, the decision made to allow the load balancing to proceed autonomously at routers is because the effort required at the server otherwise is considered not to be worth it for marginal (if any) time reduction in resolving local congestion.

4. Granularity of Load Balancing

One arbitrary number that hasn't been fully discussed yet is the number of buckets used for load balancing. No operational tests were performed on any real-life network, but ten buckets was generally viewed as being at least within an order of magnitude of what a best number might be for an average network. If there are very few buckets, load balancing may be too coarse to be effective as there is a greater chance of a single bucket's bandwidth exceeding either available path. If there are very many buckets, then the response time to resolve congestion becomes less acceptable.

5. Granularity of Fairness Enforcement

One of the goals for this solution was to provide fairness, which is a subjective term. Therefore, for implementation, fairness had to be defined and put into logical terms for software development. Ultimately, it was decided that during global congestion, traffic flows whose loss rate lies within one standard deviation of the mean were not being treated unfairly. The double negative of "not being treated unfairly" is used here to allow fairness to be further defined. Therefore, SAAM takes no special action for those flows. For those flows outside a standard deviation, however, measures are taken to either reward or punish flows. This broad fairness range was chosen not on the basis of hard science or field studies, but rather the realization that SAAM could be deployed on vastly different sizes of networks. Statistically, populations of different sizes can be normalized through parametric analysis and this serves as a starting point for a common solution. Perhaps for specialized solutions in the future, fairness will be defined in more

tunable, concrete terms such as "loss rate less than X," but for now a scalable solution is used.

Still, one standard deviation can be a very large range for data that exhibits sizeable scatter. Indeed, in test runs, it was found that mean plus or minus standard deviation sometimes defined a range with head or tail outside of loss rate bounds 0 and 100 percent. This would preclude either a rich or poor class of flows, seemingly eliminating some of the control that could have been exerted with a narrower range. The guiding philosophy here is avoiding what is viewed as unnecessary degree of control. That is, any sustained loss rate above 0% is generally regarded as unacceptable. There aren't enough degrees of unacceptability to warrant finer control measures than the broad ones used here.

THIS PAGE INTENTIONALLY LEFT BLANK

IV.   SAAM IMPLEMENTATION DETAILS

In order to add BE traffic management capabilities to SAAM, some modifications and additions have been made to the existing SAAM code. First of all, changes have been made to the existing code for design, testing, or interface issues. Then, new capabilities are added, carefully maintaining the existing code's modularity, maintainability, and overall hierarchical structure.

A.   CHANGES TO EXISTING SAAM CODE

Many changes have been made to the existing SAAM code for a number of reasons. First, the new BE solution reflects a fundamental design change where BE traffic in its current form is integrated seamlessly and no longer requires a flow request. Second, changes have been made to allow simulation testing of the new solution. Finally, changes have been made to allow the new solution to properly interface with the existing software components.

1.   Flow Generator

a.   Elimination of BE Flow Requests

The previous version of `FlowGenerator` was based on the flow request model where the agent would send a flow request, wait for the flow response, and then initiate the BE flow. This has been changed such that no flow request is sent and no flow response is needed for BE flows. For a BE flow, the agent will simply initiate the flow on its own after a short delay.

b.   Random Source Addressing for BE Packets

The previous version of `FlowGenerator` did not place source addresses in packets, relying instead on the Network layer components to perform this task. Now, capabilities have been added so that each packet receives a random source address to simulate that it came to this router from somewhere else. This feature is necessary in order to test the traffic splitting and load balancing features of the overall solution.

2. Server Agent

`ServerAgent` now recognizes and handles `EdgeNotification` messages. This is necessary in order to accomplish the discovery of edge routers and BE traffic destinations.

3. Control Executive

a. *Router Classification Data Members and Get Methods*

New data members have been added into `ControlExecutive` in order to properly classify the router in the overall BE management scheme. These members are:

- `isEdgeRouter`

- `isCoreRouter`

- `isProtectedCore`

- `isBorderRouter`

All are private Booleans which indicate whether this router is an edge router, core router, protected core router, or border router, respectively. New public get methods have been added which will obtain the value of these members. They are:

- `isEdgeRouter()`

- `isCoreRouter()`

- `isProtectedCore()`

- `isBorderRouter()`

b. *Router Status Display*

The method `displayRouterStatus()` has been changed to reflect whether this router is an edge, core, or border router.

c. *BE Management*

Two methods have been added for BE management capabilities. These are:

- acceptEdgeTraffic()

- sendEdgeNotification()

The method `acceptEdgeTraffic()` allows the resident `BestEffortTable` agent to control when and if the router is BE-capable based on communication with the server. The method `sendEdgeNotification()` is the means by which `RoutingAlgorithm` can notify the server in event of first BE packet or BE packet with unresolvable destination.

4. Demonstration Initiation Information

The class `DemoInitInfo` has been changed to add `BestEffortTable` to the list of core agents.

5. Flow Request

`FlowRequest` has been modified to reflect that best effort is no longer one of the possible types of flow requests.

6. Flow Routing Table Entry

Flow routing table entries have a field entitled "goodness." Previously, this field was initialized based on whether it was for a primary or backup route for a QoS flow. A new constant, *INSTALLED_FOR_BE*, has been added to reflect an entry created initially to carry BE traffic.

7. Message

New constants have been added to the superclass *Message* to reflect the identifiers for the new messages in the BE management system: *BEST_EFFORT_TABLE_ENTRY*, *EDGE_NOTIFICATION*, *CONGESTION_ADVISORY*.

8. Routing Algorithm

a. BE Packet Recognition and Handling

Previously, BE traffic was only recognized as part of an assigned flow having been approved after a previously submitted flow request. Then, packets without a proper flow label were discarded. Now, `RoutingAlgorithm` has been changed to recognize general BE traffic. Specifically, if the ToS bits reflect BE and the path ID portion of the flow label is zero, that packet will be treated as an ingress BE packet. `RoutingAlgorithm` will hash the packet's source address, modulo 10 the hash value,

and then call the `BestEffortTable` to obtain the path ID to map that flow onto. In the case where BE traffic has not been handled at this router before or if the packet destination is unrecognized, `RoutingAlgorithm` will have the `ControlExecutive` send an `EdgeNotification` message to the server.

b.   Requeueing Capabilities

Previously, `RoutingAlgorithm` would drop all packets for which it could not obtain routing information, including unlabeled BE traffic. Now, requeuing capabilities have been added for certain situations. Specifically, for new BE traffic to a core router or BE traffic toward a new destination, SAAM will not punish that traffic flow by dropping packets while awaiting edge router promotion or route deployment. Instead, a new `requeueBestEffPkt()` method will be called which will send the packet back to an inbound queue.

9.   Transport Interface

The method for making a BE flow request has been removed since this is no longer part of the SAAM model.

10.   Base Path Information Base

a.   Interfaces for Best Effort Manager

`BasePIB` is the class within the Server package that contains most of the server intelligence. It is also where `BestEffortManager` ties in to handle its important functions. `BestEffortManager`'s main interaction is in `refreshPathQoS()`. After *BasePIB* processes the latest LSA and updates QoS parameters for paths in its database, it calls `BestEffortManager` from within this method to report on BE loss rate for each path. It will make one of two calls based on whether or not global congestion is occurring. In the case of global congestion, it calls `BestEffortManager`'s `proactiveMonitor()` method. Otherwise, it calls `reactiveMonitor()`.

b.   New Routing Algorithms

Several new routing algorithms have been added to `BasePIB`'s inner class `RoutingAlorithm`. These have been added as methods:

- `findPathSWP()`

- `findPathSWMDP()`

- `findPathSWLCP()`

These methods return paths obtained by the SWP, SWMDP, and SWLCP algorithms respectively.

c. *Inner Class Path*

`BasePIB`'s inner class `Path` is the object representation of the paths that `BasePIB` manages. `BestEffortManager` includes new management capabilities requiring new constants, data members, and methods for `Path`. The constants classify the path's current usage in the best effort scheme as a color: `GRAY`, `GREEN`, `YELLOW`, `RED`. The new data members are:

- `bConnected`

- `bBestEffortLossRate`

- `bestEffortTrafficCondition`

- `bestEffortLossRate`

- `timeBEinitiated`

- `timeLastAdvisorySent`

- `timeConditionRed`

Together, these Boolean and numeric variables enable `BestEffortManager` to track and evaluate each path for BE management. Actual manipulation of the variables occurs with the following new methods:

- `initiateBestEffortTraffic()`

- `terminateBestEffortTraffic()`

- `newCongestion()`

- `congestionCleared()`

- `expireBEpath()`

59

- unexpireBEpath()

d.   *Access Modifiers*

Previously, all path management and route deployment took place within `BasePIB`. In accordance with the design principle of information hiding, this enabled complete usage of *private* access for associated inner classes and data structures. `BestEffortManager` was created as a separate, external class from `BasePIB` based on the software design principle of modularity. In order to allow the necessary visibility, the following members of `BasePIB` had their access changed from *private* to *protected*:

- Path

- PathQoS

- InterfaceInfo

- RoutingAlgorithm

This enables `BestEffortManager` and all other classes within the Server package to see these inner classes.

11.   Server

a.   *Auto-Configuration Cycle Sharing*

Previously, `Server` was given its auto-configuration cycle time by the `DemoStation` during configuration. No further sharing of this information was required and it was kept as a *private* member. `BestEffortManager`, however, needs this data to synchronize several of its methods which key on the periodicity of LSA's, which is equal to the auto-configuration cycle period. Therefore, a new method, `getAC_cyclePeriod()`, was added to `Server` to share this information.

b.   *Communications*

`BestEffortManager` requires two new server-to-router messages: congestion advisories and best effort table entries. These are sent via the `Server` through new methods `sendCongestionAdvisory()` and `sendBETUpdate()`, respectively.

60

B.   ADDITION OF NEW CAPABILITIES

1.   Best Effort Manager

BestEffortManager is a new class created in the saam.server package that handles all the BE management on the server end.   Functionally, ControlExecutive is running on a device. If the device is acting as a server, then the ControlExecutive has a Server agent installed, which has a BestEffortManager as one of its components.

```
┌──────────────────────────────────────────┐
│ saam.server.BestEffortManager             │
├──────────────────────────────────────────┤
│ alternatePathForThisNodePair()            │
│ calculateFairnessVariables()              │
│ computeMean()                             │
│ computeStdDev()                           │
│ expireBEpaths()                           │
│ getThisNodePairsBEpaths()                 │
│ giveToThePoor()                           │
│ globalCongestionIsOccurring()             │
│ handleBEpathFailure()                     │
│ initiateGlobalCongestionResolution()      │
│ lossRateFromThisNodePair()                │
│ primaryPathForThisNodePair()              │
│ proactiveMonitor()                        │
│ processEdgeNotification()                 │
│ reactiveMonitor()                         │
│ reclaimExpiredBEpaths()                   │
│ restoreRedundancy()                       │
│ robFromTheRich()                          │
│ switchback()                              │
│ terminateGlobalCongestionResolution()     │
│ twoBEroutesActive()                       │
│ unexpireBEpaths()                         │
│ updateBEtopology()                        │
└──────────────────────────────────────────┘
```

Figure 13.   BestEffortManager Class Structure

2.   Best Effort Table

BestEffortTable is a new class within the saam.agent.router package. BestEffortTable extends Hashtable from the java.util library and implements the TableResidentAgent and MessageProcessor interfaces.
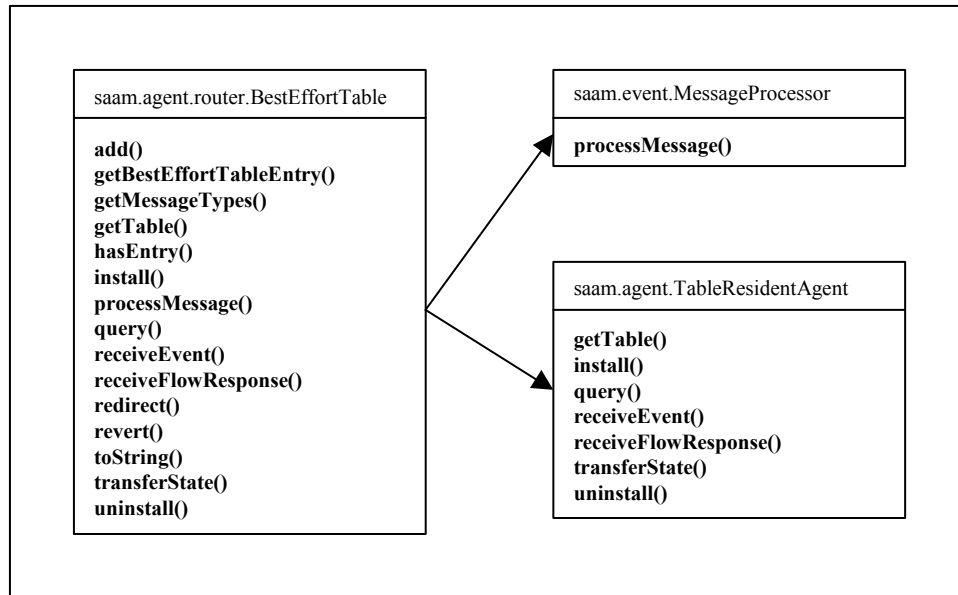
Figure 14.   BestEffortTable Class Structure

BestEffortTable agent has an inner class TrafficDestination to manage BE traffic on a per-destination basis rather than per-path.

3.   Messages

a.   *Best Effort Table Entry*

BestEffortTableEntry is a new class within the *Message* package that extends *Message*.

b.   *Edge Notification*

EdgeNotification is a new class within the *Message* package that extends *Message*.

c.   *CongestionAdvisory*

CongestionAdvisory is a new class within the *Message* package that extends Message.

V. TESTS AND RESULTS

The BE traffic management solution proposed in this thesis is complex. The solution promises many capabilities in managing BE traffic in a SAAM region. To conduct initial validation and verification of these capabilities, a series of tests have been performed. All tests were performed using the software simulation of SAAM's `DemoStation` *[XML SAX Parser] version 1.0* with custom test topologies (written as XML files) and the flow generator and sink agents developed in [11]. In order to not overload the processing capabilities of the host computer and to facilitate data capture, simulation-to-real time scales of 200-500 were used in all tests (i.e. 200-500 seconds was required to simulate one second of real time).

The overall focus of testing was more qualitative than quantitative. Primarily, the tests are used to show that the solution built from scratch performs as designed without concern for optimization. More exhaustive data analysis testing to fine-tune internal design parameters on operational networks is available as possible future work.

A. EDGE ROUTER DISCOVERY AND PACKET REQUEUEING

The most fundamental test of the overall solution is that it can stand up and begin operating. The solution was developed with the idea that BE routers and traffic destinations will be discovered at run-time. Initially, a SAAM region will be composed entirely of core routers. Once BE traffic begins to flow, the affected core routers should be promoted to edge routers with their `BestEffortTable` agents receiving routing entries for the BE traffic. Further, the BE packets that initiate the flow should not be dropped during the discovery process, but rather, requeued until the router is able to handle them.

1.   **Test**
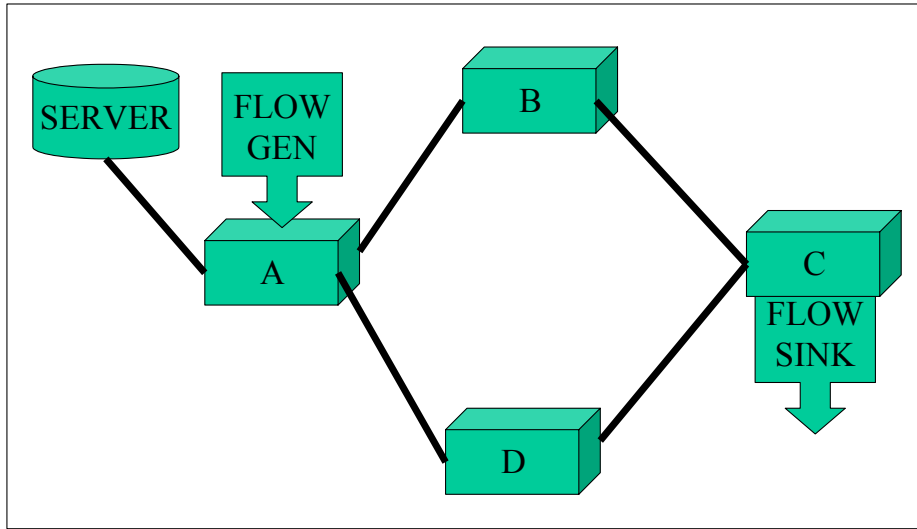
Figure 15.   Discovery / Requeueing Test Topology

The test topology shown in Figure 15 was developed and loaded into the simulator. There is a single source of BE traffic entering the region at Router A, destined for Router C. Therefore, Routers A and C should be promoted to edge routers. Router A's `BestEffortTableAgent` should receive two table entries reflecting the two routes to Router C.

2.   **Results**

The test was successful in all areas. First, the discovery of routers and interfaces was successful.
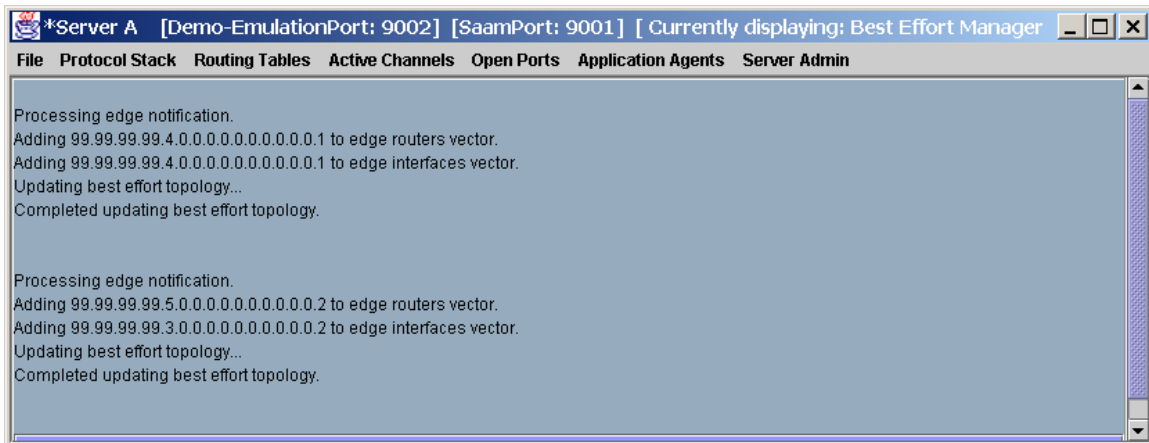
Figure 16.   Edge Discovery

Second, routes were successfully deployed for BE traffic to Router A. Matching flow routes were also deployed to Router A's flow routing table and traffic was successfully carried from Router A to C.

Dest IPv6 Address	Map to Path	Traffic Split
99.99.99.99.3.0.0.0.0.0.0.0.0.0.0.2	82	80
99.99.99.99.3.0.0.0.0.0.0.0.0.0.0.2	92	20

Figure 17.   BE Route Deployment

Path ID	Interface Number	Next Hop IPv6 Address	Goodness
2	0	99.99.99.99.1.0.0.0.0.0.0.0.0.0.0.1	2
82	2	99.99.99.99.2.0.0.0.0.0.0.0.0.0.0.2	2
92	1	99.99.99.99.4.0.0.0.0.0.0.0.0.0.0.2	2

Figure 18.   Flow Routing Entries to Support BE

Finally, to verify the requeuing mechanism was working, the same test was run with a controlled number (100) of packets. Due to successful requeueing, every packet made it to Router C despite being initially delayed at ingress Router A during edge promotion and route deployment.

Figure 19.   100 Packet Generation

Figure 20.   Receipt of 100th Packet

B.   LOAD BALANCING

Simply routing BE traffic is fundamental. In order to deliver on its promise of providing a better best effort, the solution must successfully perform its built-in load balancing capabilities.

1.   Test
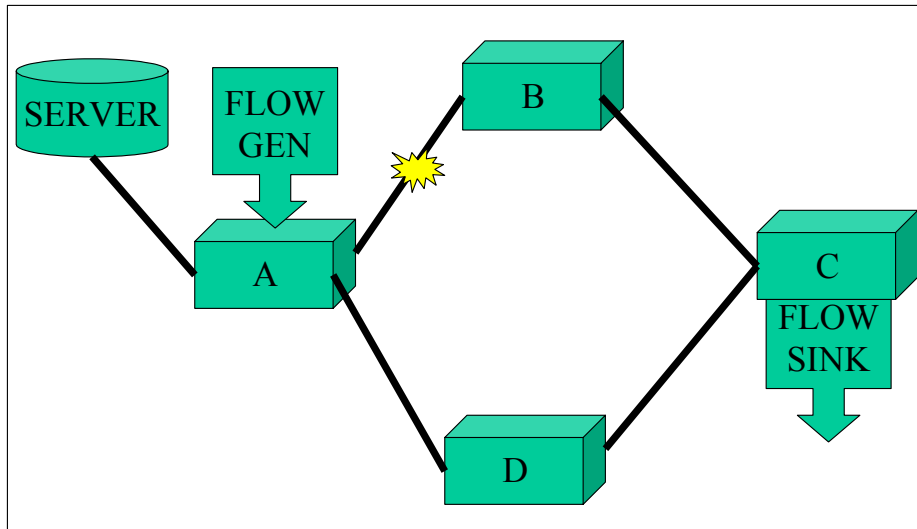
Figure 21.   Load Balancing Test Topology

The topology shown in Figure 21 was loaded into the simulator. This topology is identical to that of Figure 15, which was used during the discovery test, with one exception. The primary (widest) route from Router A to Router C will be insufficient to carry all of the BE traffic between them. Therefore, the Server should discover this through an LSA message and notify Router A's `BestEffortTable` agent to begin load balancing. Load balancing should consist of shifting traffic from the primary to alternate route until congestion clears followed by a gradual reversion of traffic to the primary path.
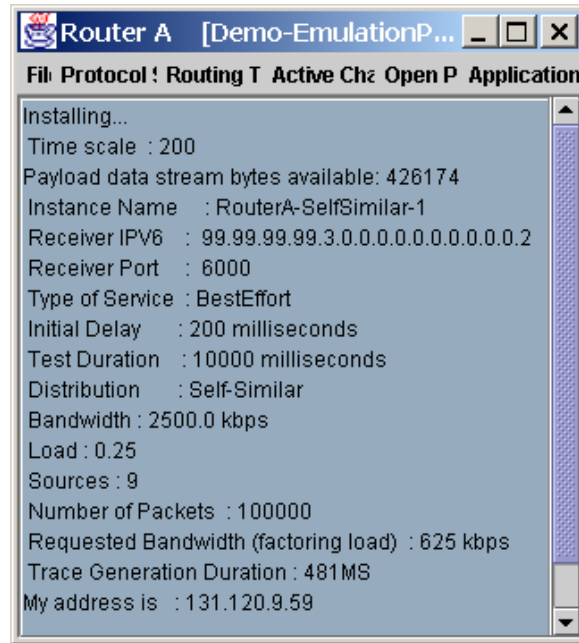
Figure 22.   Test Agent for Load Balancing

Specifically, the agent shown in Figure 22 was loaded, which requests 625 kbps of bandwidth, using a pseudo-random self-similar packet distribution. Links AB and AD (see Fig.21) were edited to each have a capacity of 622 kbps. Not only will this force congestion when all traffic is on a single link, it will test to see whether load balancing will find the solution that exists (traffic split so that neither link is congested).

2.   Results

The simulation proceeded as expected and load balancing was successful in finding a lossless routing solution involving both paths.
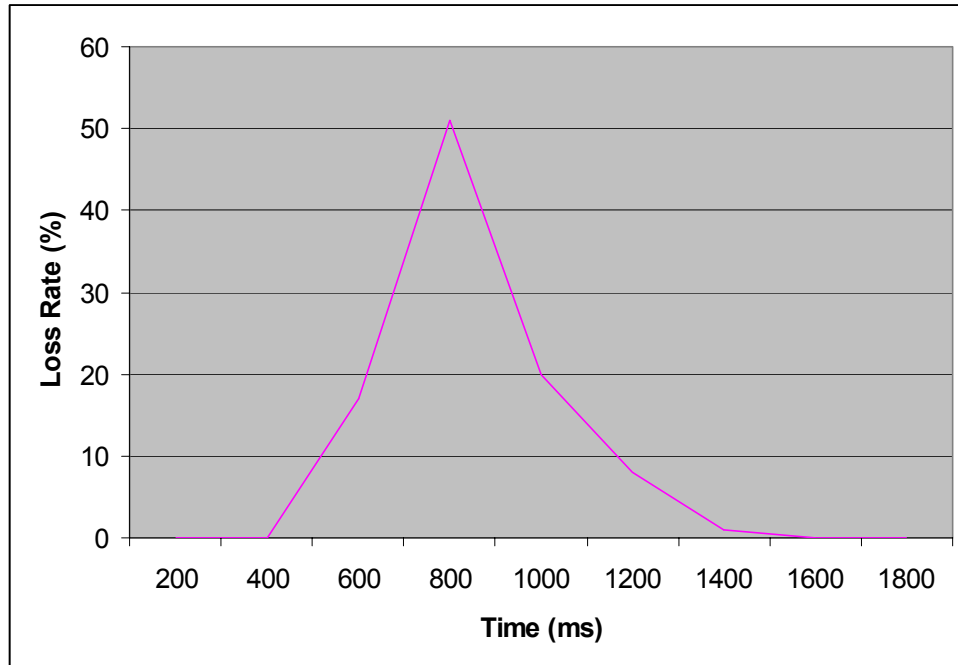
Figure 23.   Primary Path Loss Rate vs. Time for Load Balancing Test

Once the 625 kbps flow generating agent-initiated traffic, loss rates shown in Figure 23 were observed. As expected, there was no loss rate initially as the outbound queues filled up and absorbed excess packets. At some time between 400ms and 600ms, the primary path's interface's outbound queue began to overflow resulting in a reported loss rate on the 600ms LSA. This caused the `BestEffortManager` to send a `CongestionAdvisory`-YELLOW message to initiate load balancing by Router A's `BestEffortTable` agent. Router A instantly shifted to a 90/10 split.

Next, the reported loss rate continues to increase for one more cycle and then starts to drop rapidly. Router A continues to shift traffic during the time of continued reported loss.   Finally,   at   1600ms,   loss   rate   has   decayed   to   zero.   A `CongestionAdvisory`-GREEN message was sent to Router A causing termination of load balancing and maintenance of current split.

Figure 24.   Load Balancing Test Final Split

At that time, Router A had reached a split of 50/50 between the primary and secondary paths. Due to the fact that reported loss rate is exponentially smoothed (with α = 0.7 over 100ms sampling intervals), Router A may have over-corrected as reported loss rate decayed below the threshold. If this was the case, then the reversion process should gradually place traffic back onto the primary path.

Unfortunately, a reversion interval of 30 minutes and a simulation time scale of 200 make for a very long (100 hours) test. Therefore, to test the reversion feature, the code was temporarily modified to make the reversion interval equal to the redirection interval of 200ms in order to verify reversion operates as designed. The same scenario was run with this modification in place. As expected, reversion did begin to take place once the initial congestion had cleared. Not surprisingly, the split reverted all the way back to its initial setting of 100/0, which, of course, caused the congestion to reappear since load had not changed. In the solution's real life implementation, this amount of reversion would have taken 2.5 hours (five buckets of traffic times 30 minutes), hopefully enough time for the network's usage to lessen. Otherwise, load balancing would be expected to resume and find a loss-free balance.

C. CONGESTION BYPASS

If congestion were to develop in a region affecting both a `BestEffortTable` agent's routes to a destination, then load balancing would be useless. This is where the overall solution's congestion bypass mechanism comes to the rescue.
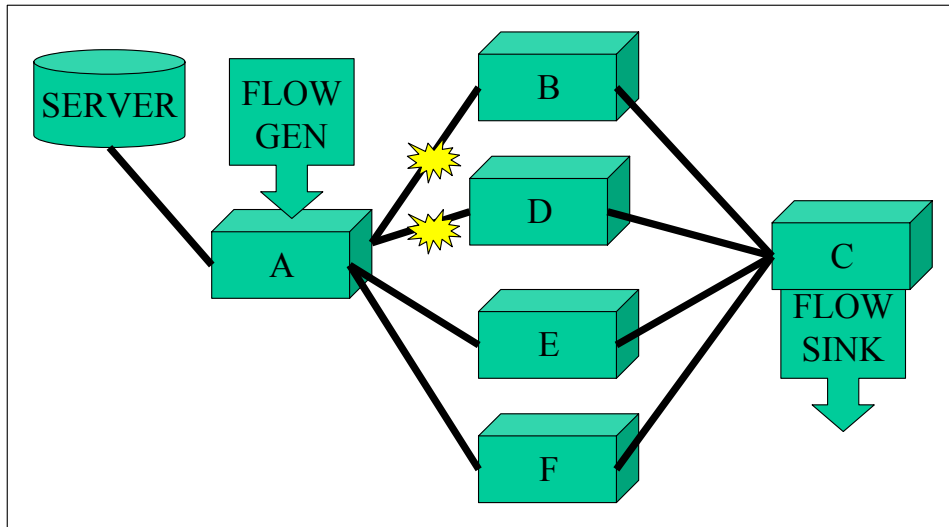
Figure 25.   Congestion Bypass Test Topology

The test topology shown in Figure 25 was developed and loaded into the simulator. A single BE flow originates on Router A destined for Router C. Congestion will develop on the initial two routes. This should cause load balancing to take place for 2 seconds, shifting all traffic to the alternate path. After this time (the local resolution timeout is 2 seconds), the `BestEffortManager` should perform congestion bypass and deploy the other two routes to Router C. Router A's `BestEffortTable` agent should install the new routes and direct 100% of the traffic to the new primary route.

2.   Results

The test was completed successfully.
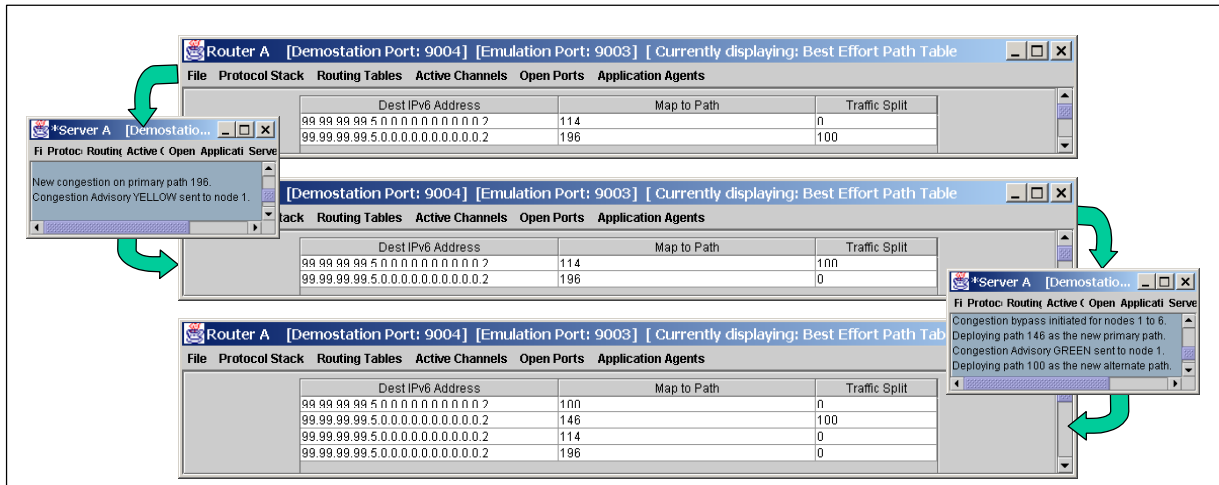
Figure 26.   Congestion Bypass Test Results

70

As seen in Figure 26, everything went as expected. Initially, paths 114 and 196 were deployed as the primary and alternate routes from Router A to Router C. As soon as congestion was noted on path 114, a `CongestionAdvisory`-YELLOW message was sent to Router A's `BestEffortTable` agent. Load balancing goes into effect for 10 auto-configuration cycles, by which time all traffic has been directed to alternate path 196. Now that the local resolution timeout has expired, the `BestEffortManager` initiates congestion bypass procedures and deploys routes 100 and 146, which avoid the congestion.

D.   FAIRNESS

Load balancing and congestion bypass work well when congestion is in isolated pockets of the network and can be avoided through active management by the server. When congestion is everywhere (global), all methods of congestion avoidance become useless. Here, the BE management solution turns its attention from congestion avoidance to fairness enforcement.

1.   Test

The fairness enforcement procedures cover an infinite number of situations ranging anywhere from two competing hosts to millions. It is impossible to develop every single scenario that could happen or even a representative scenario. Rather, the procedure is tested to see if it properly carries out its two main corrective actions: robbing from the rich and giving to the poor.
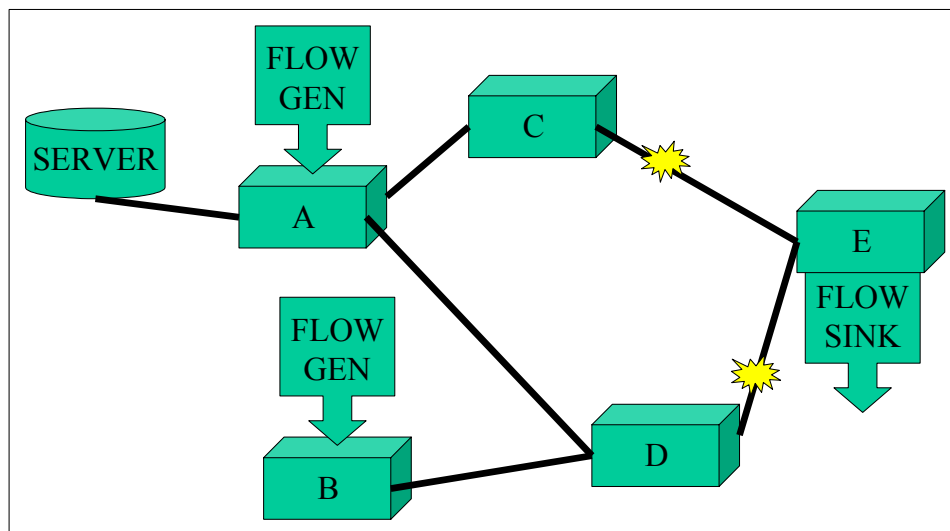
Figure 27.   Fairness Test Topology

71

The topology shown in Figure 27 was developed and loaded into the simulator. The flow generators for Routers A and B provide more load than the network can handle, forcing global congestion. When and if `BestEffortManager` recognizes the global congestion, test signals will be inserted marking the A-E flow as rich and the B-E flow as poor. `BestEffortManager` should take appropriate measures at that point to ensure fairness.

2. Results

The test was successful with the `BestEffortManager` taking correct actions.
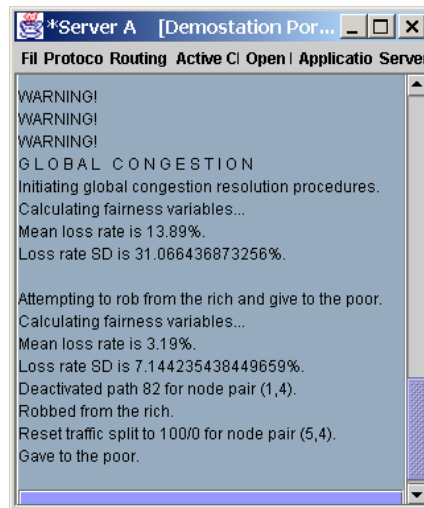
Figure 28.   Fairness Test Results

As programmed, `BestEffortManager` waited to see if the situation could be corrected by local resolution. After that time, `BestEffortManager` initiated proactive monitoring and global congestion resolution procedures to include an initial round of loss rate statistics. It then waited another local resolution timeout to ensure any local resolution measures enacted just before global congestion started were allowed to run their course. Indeed, by that time the loss rate had improved (see Fig. 28), but proactive measures were still in order. `BestEffortManager` successfully completed a rob-from-the-rich procedure for A-E and a give-to-the-poor procedure for B-E.

E. PERIPHERY UTILIZATION

One of the hopes for the new BE management solution is for a particular emergent behavior: network periphery utilization. The new system for BE traffic management should cause the network periphery to be utilized when the network center

is overloaded. Further, this behavior should emerge, as it were, from the simple underlying algorithm of load balancing, which is not at all concerned with centers or peripheries.

1.   Test
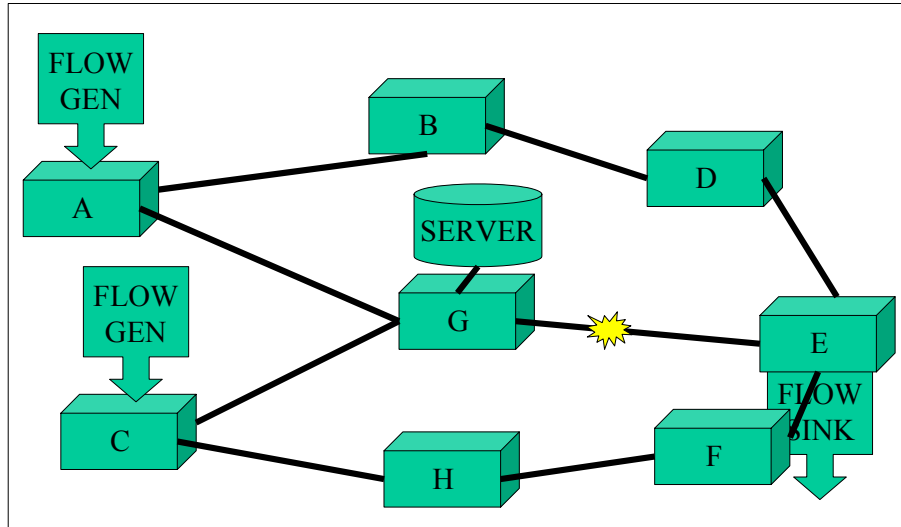
Figure 29.   Periphery Utilization Test

The topology shown in Figure 29 was developed and loaded into the simulator. All links have equal bandwidth causing all paths comprised of these links to have equal bandwidth as well. Therefore, the primary paths assigned for BE traffic from Routers A and C to Router E should both share Router G and the link GE. The combined offered load was purposely designed to overwhelm link GE and force load balancing to take effect. Once the congestion clears, load balancing should terminate. The final state of the network should be one in which the peripheral links share some of the load.

2. Results
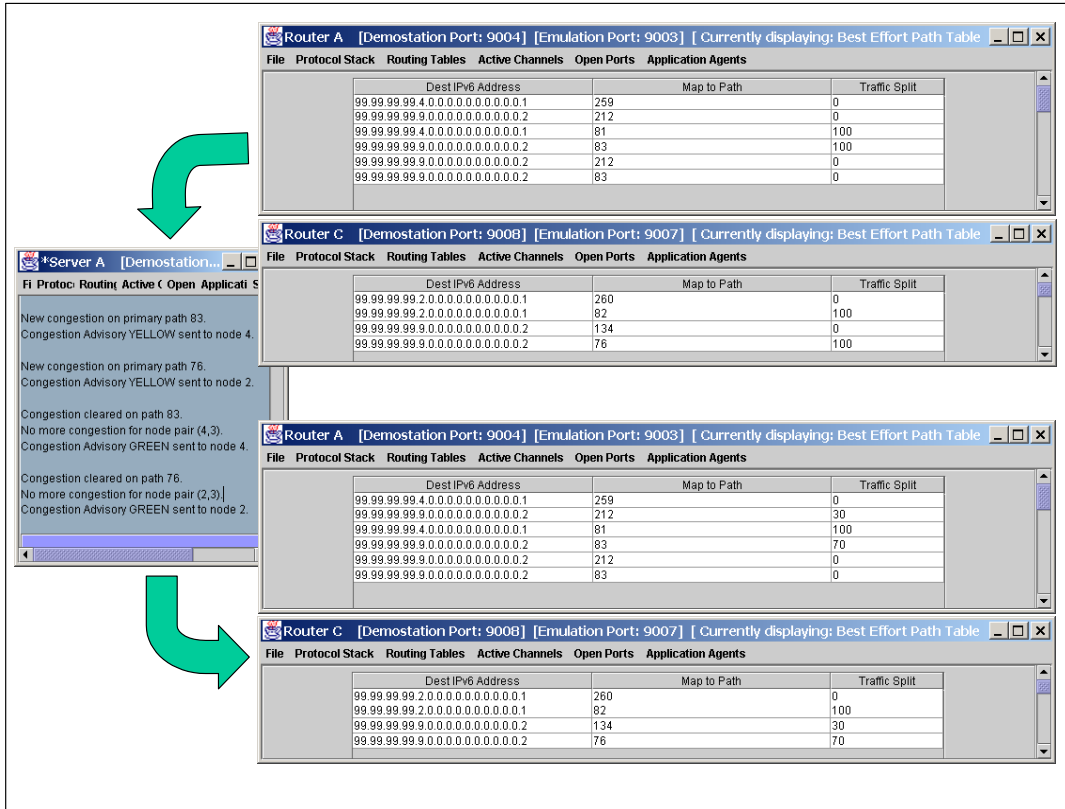
Figure 30.   Periphery Utilization Test Results

The test was completed successfully with results as expected. Initially, Routers A and C were given primary paths 83 and 76, respectively. As these were chosen by the SWP algorithm, they shared Router G and link GE. Once congestion developed, `BestEffortManager` sent `CongestionAdvisory` messages to effect load balancing. This caused Routers A and C to shift traffic to alternate paths 212 and 134, respectively. These paths were chosen by the SWMDP algorithm and contained links and nodes on the periphery of the network. Traffic cleared when 70% of the traffic had been shifted to peripheral paths.

F.   COMPARATIVE BENEFIT

The previous tests have provided a mostly qualitative perspective in examining an overall network. The last aspect that needs to be examined is a quantitative one that will examine a single flow for comparative benefit with the new BE solution.

1.   Test

The load-balancing test from before will be repeated to examine the difference between loss rates when load balancing is enabled or disabled.

Offered Load	750 kbps
Primary Path Bandwidth	622 kbps
Alternate Path Bandwidth	622 kbps

Table 5.   Load Balancing Test Parameters

The simulation will be run five times each for the cases with and without load balancing. In the load balancing case, loss rate should eventually return to zero as traffic is split between the primary and alternate paths. In the case without load balancing, a loss rate should persist indefinitely.

2.   Results

The results show an appreciable benefit from load balancing in this case.
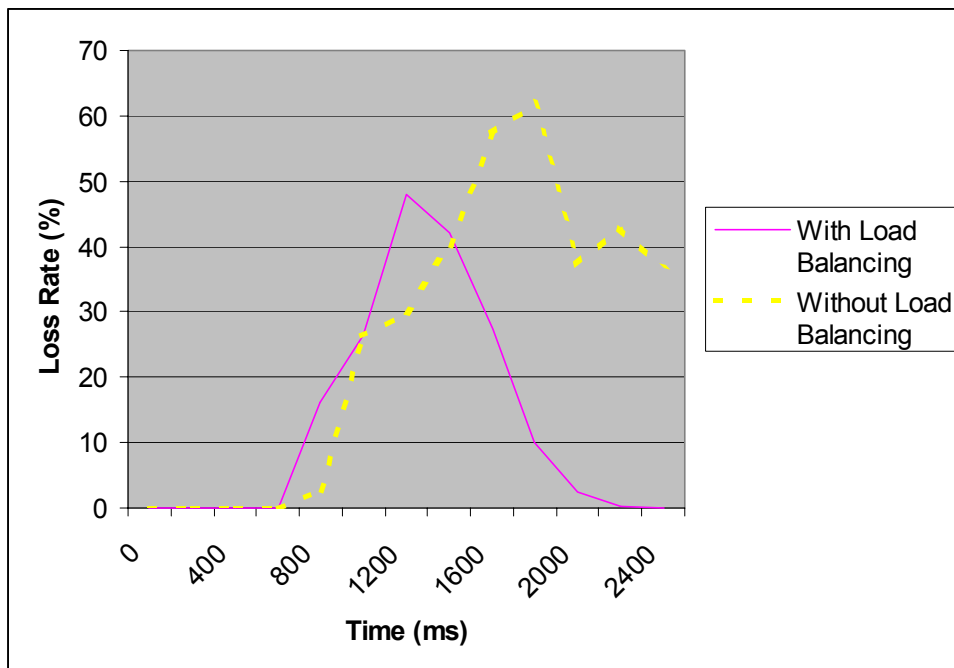
Figure 31.   Loss Rate Comparison With and Without Load Balancing

Figure 31 shows the average result for each case over five test runs. As expected, the performance is similar initially. Both flows experience no loss while the packet

queue is filling up. The queue overflows near 800ms causing packet loss. In the case without load balancing, the lossy condition persists. In the load balancing case, however, packet loss ceases after about 1.6 seconds.

VI. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

1. Requirements Revisited

The BE traffic management solution developed in this thesis addresses all previously stated requirements and meets those requirements fully or in part.

a. Security

While the solution developed herein offers no fully implemented security features for BE traffic, the underlying code allows future developers to set a software switch that will identify a router as a protected core router.

b. Light-Weight Routers

SAAM's routers remain lightweight as the servers of SAAM take on almost all of the memory and computational requirements associated with the developed solution.

c. Interoperability

Due to the handling added for unlabeled IPv6 packets, SAAM's interoperability has been enhanced with regard to BE traffic.

d. Fault Tolerance

The solution developed ensures complete fault tolerance for BE traffic with regard to connectivity. There would have to be no surviving path between a source and destination for traffic between the two to be permanently disrupted.

e. Fairness

The solution developed addresses fairness in times of resource contention. This fairness is only between node pairs, not flows. Further, it is of rather coarse granularity based on exponentially smoothed reported loss rates.

f. Adaptive Response Mechanisms

The solution developed has successfully incorporated adaptive response mechanisms, allowing automatic rerouting based on changing network conditions.

g. *Stability*

Stability has been addressed through careful selection of time constants. However, no testing has been performed in the area of convergence-time minimization through parameter tuning.

h. *Scalability*

Scalability will depend on memory and CPU speed of deployed routers. The solution developed herein has addressed scalability through its economical choice of when and where to deploy the table entries and activate the agents needed to handle BE traffic.

i. *Intelligent Provisioning*

Intelligent provisioning has been achieved by the solution's method of assigning high performance routes during periods of low load and then moving traffic to less utilized areas when congestion results in a region of high demand.

j. *Packet Recovery*

This solution provides for a number of packets up to the router's inbound queue size to be buffered while the router awaits BE characterization.

2. Overall

SAAM now has an effective, but not necessarily optimal, solution for managing BE traffic. This solution builds on the strengths of the SAAM architecture and stays true to its core paradigm. SAAM now has the potential to deploy as an integrated, comprehensive network solution rather than just a module for handling QoS traffic.

The Internet may provide a "best" effort, but SAAM delivers an even better effort through intelligent central management. BE users in a SAAM region can expect more than just the connectivity promised elsewhere. SAAM does indeed provide a "better best effort".

B. RECOMMENDATIONS FOR FUTURE WORK

This thesis has successfully brought BE traffic management to SAAM. In many ways, however, it should be viewed as "Version 1.0." There are several unanswered questions as well as areas for improvement. These are left open as areas to be researched and/or improved by future thesis students and SAAM developers.

1.   A Border Gateway Agent

The routers at the borders of SAAM regions will need to know where to route BE traffic leaving the region. One solution would be to develop a border gateway agent for those routers. This agent should handle an exterior gateway protocol such as BGP in order to exchange routing information with external, non-SAAM routers. This will simplify SAAM's interior BE routing as all BE traffic could be directed to the router(s) with border gateway agents.

2.   Security Features

SAAM's BE policy could be used as another avenue of attack by malicious users. For example, a denial-of-service attack could cycle the congestion handling mechanisms. Now that the BE solution for SAAM has been developed and implemented, classic BE security research should be undertaken.

3.   Deployable Agents

Currently, the new router agents developed in this thesis are among the "core agents," those that are preinstalled in every router. This means that they remain latent until that router receives BE traffic, if ever. One avenue that could be explored is the desirability of making the BE agents deployable and only sending to a router on a need basis.

4.   Fine Tuning of Parameters

In building this solution from scratch, many of the parameters involved were chosen more on the basis of reasonableness than anything else. For values such as the number of traffic buckets, reversion time, and path expiration time, more extensive data analysis could be performed with the intent of recommending a range of values for given situations. Convergence time is an important consideration for routing protocols and could be a possible starting point for further analysis involving the change of these parameters versus various network conditions.

5.   An Even Better Best Effort

There are several areas in the solution where the management schemes could be made more complex and possibly more effective. On the server end, total control could be assumed by developing an algorithm seeking to manipulate all path splits. Currently, the server just deploys paths and turns load balancing on and off. On the router end,

there is no limit to the number of paths that could be deployed to a destination. Perhaps an algorithm could be developed where each node pair has three paths. Even more, each router could be given a full set of disjoint paths to a destination. Again, the supporting algorithms would have to be developed for these cases in regard to how to accomplish load balancing in this more complex arrangement.

6. Implementation of Other Algorithms

While they do take network-wide congestion into account, the algorithms used in this solution do not take full advantage of all the SAAM server has to offer. For instance, all of the path finding algorithms described for this solution take a fundamentally greedy approach that focuses on a single node pair. SAAM's servers could potentially make better decisions through a class of path finding algorithms that focus on "minimum interference." Minimum interference algorithms take into account where other paths have been established and in some cases a heuristic approach attempting to minimize interference with some future path. As it is, the only consideration this solution gives to other deployed paths is taken through attention to congestion parameters.

7. Refinement of Fairness Approach

The fairness measures adopted by this solution are only on a per-aggregate-flow basis. A finer solution would be to provide fairness between hosts or sessions. Additionally, the methodology adopted of a Robin Hood do-good approach based on coarse statistical measures could be made more sophisticated. For one, fairness could be better defined than loss within a standard error of mean. Additionally, rather than giving and taking entire paths to node pairs, a rate control could be instituted which would allow for a measured sharing.

APPENDIX A: GLOSSARY

Address Resolution Protocol (ARP)

A protocol that dynamically discovers the physical address of a system, given its IP address.

Admission Control

The portion of QoS management concerned with whether or not to admit a flow and approve its flow request.

Application layer

One of the seven layers in the OSI model for computer communications. It is the layer in which applications and services run.

ARPCache

The class in the SAAM software that provides functionality roughly equivalent to an ARP cache table on a conventional router.

Autonomous System (AS)

An internetwork that is part of the Internet and has a single routing policy.

Best Effort (BE)

The term applied to traditional Internet traffic for which no QoS guarantees are made

Delay jitter

For a given packet, the amount by which the packet's delay varies from the mean delay for that stream of packets.

Delay variation

-see "delay jitter"-

Differentiated Service (DiffServ)

A model for handling QoS traffic in which multiple flows with equal service requirements are aggregated into a class of service.

Dijkstra's algorithm

A well-known algorithm named after Edsger Dijkstra that computes the shortest path between two nodes in a graph.

Flow label

Given to packets in an assigned SAAM QoS flow, the label concatenates a flow identification and a path identification for flow and path management.

Flow routing table

A table located in SAAM routers that routes packets based on flow labels.

Heavyweight

A term given to software components designed with no concern for processing power and/or memory requirements.

Hop

Refers to the travel between adjacent nodes in packet-switched networks.

Integrated Service (IntServ)

A model for handling QoS traffic in which state is maintained on each flow due to allowing flow-individualized QoS parameters.

Internet

When capitalized refers to the entity that is the worldwide computer internetwork. Lower cased usage is proper only when spelled out as internetwork referring to a part less than or separate from the worldwide portion.

Internet Draft

A document submitted as part of the Internet standards development process. It is for comment in open forum, but unlike the similar RFC document, it has an expiration date.

Internet Engineering Task Force (IETF)

A set of working groups made up of volunteers who develop and implement Internet protocols.

Jitter

-see "delay jitter"-

Lightweight

A term given to software components designed to be lean in terms of processing and memory requirements.

Link-state flooding

The process by which neighboring link-state protocol routers share information through occasional saturation of interior links with topology information. This saturation is necessary to ensure every router receives all sharing information from every other router.

Local Preference

In BGP, a purely internal path attribute that can be used to set preference for external routes.

Local resolution timeout

For SAAM's BE traffic management solution, it is the time equal to the maximum time it could take for a full cycle of load balancing to occur, which is equal to the number of buckets multiplied by the auto-configuration cycle time.

Multiexit Discriminator

In BGP, a path attribute that can be assigned by a neighboring AS that indicates preference among multiple routes between the two AS's.

Network layer

One of the seven layers in the OSI model for computer communications. It is the layer through which physical connections become abstracted and dissimilar entities can be viewed as a logical network of links and nodes.

Next Generation Internet (NGI)

A Presidential Initiative with primary goal of researching network technologies to enable the Internet to scale in size, speed, and reach.

Next hop

A term referring to the next sequential physical link a packet must traverse in a packet-switched network.

Poison reverse

A technique to prevent circular traffic in RIP domains in which neighbors tell their next hop for a route that the cost through them is infinity.

Quality of service (QoS)

A term referring to any combination of a number of performance metrics concerning data flow through a network.

Reachability

A term used in BGP that indicates whether or not a given destination is reachable.

Redirection

In SAAM's BE traffic management, the process by which traffic between a source and destination is redirected to the alternate path.

Request for Comment

A document submitted as part of the Internet standards development process. It is permanently archived and tracked to indicate the state of the technology development for its research area.

Resource Management

The portion of QoS management concerned with tracking resources and their current usage.

Reversion

In SAAM's BE traffic management, the process by which traffic between a source and destination is reverted back to the primary path.

Robin Hood

A mythical vigilante from Merry Olde England who robbed from the rich and gave to the poor.

Route flapping

The phenomenon in which a dynamic routing solution causes a given traffic flow to constantly oscillate between multiple routes.

RoutingAlgorithm

The class in the SAAM software that provides functionality roughly equivalent to the routing algorithm on a conventional router.

Server and Agent-based Active Network Management (SAAM)

A comprehensive network management solution being developed at the Naval Postgraduate School that seeks to provide management for QoS traffic while maintaining the underlying robustness of the TCP/IP network architecture.

Service Level Agreement (SLA)

An agreement between neighboring AS's or ISP's that concerns route information sharing and traffic carrying among other things.

Split horizon

A technique to prevent circular traffic in RIP domains in which neighbors do not report routes back to the next hop for that very route.

Switchback

In SAAM's BE traffic management, the process by which a server directs all traffic for a node pair to be switched back to the primary path.

TCP/IP

Refers to the dominant combination of protocols on the Internet today: Transmission Control Protocol (TCP) and Internet Protocol (IP). Together, IP provides the connectivity and TCP the end-to-end flow control.

Transport layer

One of the seven layers in the OSI model for computer communications. It is the layer that provides methods of flow control, ordering of received data, and acknowledgement of correctly received data.

Utilization

For a network link, the percent usage as measured by data throughput over throughput capacity.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B: LIST OF ACRONYMS

ARPA

Advanced Research Projects Agency

AS

Autonomous system

BE

Best effort

BGP

Border Gateway Protocol

DARPA

Defense Advanced Research Projects Agency

DiffServ

Differentiated service

DoD

Department of Defense

ECMP

Equal cost multipath

IETF

Internet Engineering Task Force

IntServ

Integrated service

IP

Internet Protocol

IPv4

Internet Protocol version 4

IPv6

Internet Protocol version 6

ISO

International Organization for Standardization (not an acronym, but derived from Greek *isos*)

ISP

Internet service provider

LSA

Link state advertisment

LSP

Label-switched protocol

MATE

Multi-adaptive traffic engineering

MPLS

Multiprotocol Label Switching

NASA

National Aeronautics and Space Administration

NGI

Next Generation Internet

OSI

Open Systems Interconnect

OSPF

Open Shortest Path First

QoS

Quality of service

RIP

Routing Information Protocol

SAAM

Server and Agent-based Active Network Management

SLA

Service level agreement

TCP

Transmission Control Protocol

TEWG

Traffic Engineering Working Group

ToS

Type of service

XML

Extensible Markup Language

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C: BEST EFFORT MANAGER SOURCE CODE

```
//25Feb02[Wofford] - Sudafed renamed Congestion Advisory.
//15Feb2002[Wu] Repackaged
//04Feb 2002 [Wofford] Created.

package org.saamnet.saam.server;

import org.saamnet.saam.gui.MAGMAAdminGui;
import org.saamnet.saam.net.*;
import org.saamnet.saam.message.EdgeNotification;
import org.saamnet.saam.message.CongestionAdvisory;
import org.saamnet.saam.message.FlowRoutingTableEntry;
import org.saamnet.saam.agent.router.FlowRoutingTable;

import java.util.Enumeration;
import java.util.Vector;
import java.net.*;

/**
 * BestEffortManager (BEM) is the intelligence within the SAAM server that manages
 * best effort traffic.  It continuously monitors LSA's through one of two modes:
 * proactive or reactive.  Absent global congestion, BEM communicates with BE
 * router agents to handle local congestion.  During global congestion, BEM takes
 * measures to enforce fairness.
 */
public class BestEffortManager
{
  //when a BE path is expired, the period it will remain inactive
  public final static long PATH_EXPIRATION_TIME = 1800000;//30 minutes

  //references required for operation
  private BasePIB myBasePIB;
  private Server myServer;

  private MAGMAAdminGui gui;

  private boolean globalCongestion;//is it occurring?

  private long localResolutionTimeout;//allow local resolution to take place
  private boolean lrtInitialized;//tracks initialization of localResolutionTimeout
  private long timeLastActionTaken;//the last time an active measure was taken
  private long timeLastCongestion;//the last time congestion was noted
        private long timeLastSwitchback;//the last time a switchback was performed

  //statistics used for fairness measures
  private double meanLossRate;
  private double stdLossRateDev;

  //Vectors that store routerID's and interface addresses that
  //are registered for best effort traffic BY THEIR STRING REPRESENTATION.
  Vector vBestEffortRouters = new Vector();
  Vector vBestEffortDestAdds = new Vector();

        //used in bePathAdmin()
        private static final byte UNEXPIRE_PATHS = 0;
  private static final byte GET_PATHS = 1;
        private static final byte UPDATE_LOSS_RATE = 2;
        private static final byte RECLAIM_PATHS = 7;

        //used in beNodePairAdmin()
        private static final byte DEPLOY_INITIAL_PATHS = 3;
        private static final byte GET_LOSS_RATES = 4;
        private static final byte ROB_IF_RICH = 5;
        private static final byte GIVE_IF_POOR = 6;
```

```
  /**
   * CONSTRUCTOR
   * @param   basepib  required reference
   * @param   server   required reference
   */
  BestEffortManager(BasePIB basepib, Server server)
  {
    myBasePIB = basepib;
    myServer = server;

    // Create Gui for PIB display during generation.
    gui = new MAGMAAdminGui("Best Effort Manager", server);
    server.getControlExec().addMagmaGui(gui);

                gui.sendText("Initializing...");

    globalCongestion = false;
    lrtInitialized = false;
                timeLastSwitchback = 0;

                gui.sendText("initialized.");
  }


  /**
   * Processes EdgeNotification messages.
   * @param   edgeNotif  the message
   */
  protected void processEdgeNotification (EdgeNotification edgeNotif)
  {
    //initialize localResolutionTimeout variable
    if (!lrtInitialized)
    {
      localResolutionTimeout = 10 * myServer.getAC_cyclePeriod();
      lrtInitialized = true;
                      gui.sendText("\nLocal resolution timeout is " +
localResolutionTimeout + "ms.");
    }

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

    gui.sendText("\nProcessing edge notification.");

    IPv6Address interfaceAddress = edgeNotif.getEdgeInterfaceAddress();

    //Xie-darpa
    BasePIB.InterfaceInfo edgeInterfaceInfo = (BasePIB.InterfaceInfo)
myBasePIB.htInterfaces.get(interfaceAddress.toString());
    if (edgeInterfaceInfo == null)
    {
      gui.sendText("\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())))
    {
      gui.sendText("Adding " + routerID.toString() + " to edge routers vector.");
      vBestEffortRouters.add(routerID.toString());
      count++;
    }
    //is this a newly discovered destinaton interface?
    if (!(vBestEffortDestAdds.contains(interfaceAddress.toString())))
    {
```

92

```
      gui.sendText("Adding " + interfaceAddress.toString() + " to edge interfaces
vector.");
      vBestEffortDestAdds.add(interfaceAddress.toString());
      count++;
    }

             if (count > 0)
             {
         gui.sendText("Updating best effort topology...");
         updateBEtopology();
         gui.sendText("Completed updating best effort topology.");
             }
             else
             {
                 gui.sendText("No new information; best effort topology still
accurate.");
             }
  }//end processEdgeNotification()


  /**
   * When global congestion is absent, reactive monitoring takes place.
   * @param   path      the path being observed
   * @param   lossRate  the best effort loss rate on that path
   */
  protected void reactiveMonitor(BasePIB.Path path, short lossRate)
  {
    if (lossRate > myBasePIB.thresholdLossRate)
    {
      unexpireBEpaths();//see if any expired paths are due for reuse

      switch (path.bestEffortTrafficCondition)
      {
      case BasePIB.Path.GRAY:
        break;

      //this is a case of new congestion
      case BasePIB.Path.GREEN:
        int firstNodeID = path.getSrcNodeID();
                           int lastNodeID = path.getDestNodeID();
                           if (path == alternatePathForThisNodePair(firstNodeID,
lastNodeID))
                           {
                                 BasePIB.Path primaryPath =
primaryPathForThisNodePair(firstNodeID, lastNodeID);
                                 if (primaryPath.bestEffortTrafficCondition ==
BasePIB.Path.GREEN)
                                 {
                                     gui.sendText("\nNew congestion on an
alternate path while primary path lossless.");
                                     switchback(firstNodeID, lastNodeID);
                                 }
                                 else
                                 {
                                     path.newCongestion();
                                 }
                           }
                           else
                           {
                 IPv6Address routerID = (IPv6Address) myBasePIB.htNodeIDtoRouterID.get(new
Integer(firstNodeID));
                 myServer.sendCongestionAdvisory(routerID, path.getPathID().intValue(),
CongestionAdvisory.YELLOW);
                 path.newCongestion();
                                     gui.sendText("\nNew congestion on primary path " +
path.getPathID().intValue() + ".");
                                     gui.sendText("Congestion Advisory YELLOW sent to
node " + firstNodeID + ".");
                           }
```

93

```
      break;

    //if local resolution has failed, deploy new paths or initiate global congestion
procedures
    case BasePIB.Path.YELLOW:
      boolean noLocalResolutionPossible = false;
      if ((System.currentTimeMillis() - path.timeLastAdvisorySent) >
(localResolutionTimeout * myBasePIB.timeScale))
      {
        noLocalResolutionPossible = true;
        firstNodeID = path.getSrcNodeID();
        IPv6Address srcRouterID = (IPv6Address) myBasePIB.htNodeIDtoRouterID.get(new
Integer(firstNodeID));
        lastNodeID = ((Integer) (path.getNodeSequence().firstElement())).intValue();
        IPv6Address destRouterID = (IPv6Address) myBasePIB.htNodeIDtoRouterID.get(new
Integer(lastNodeID));
        BasePIB.Path bePath1 = myBasePIB.routingAlgorithm.findPath(srcRouterID,
                                                          destRouterID,
                                                          null,

myBasePIB.routingAlgorithm.SHORTEST_WIDEST_LEAST_CONGESTED_PATH);
        if (bePath1 != null)
        {
                                            gui.sendText("\nCongestion bypass initiated
for nodes " + firstNodeID + " to " + lastNodeID + ".");
          noLocalResolutionPossible = false;
          Integer bePathID1 = bePath1.getPathID();
                                            Integer bePathID2 = null;
                                            gui.sendText("Deploying path " + bePathID1 +
" as the new primary path.");
          if (!bePath1.bCreated)
          {
            myBasePIB.setupPath(bePath1, bePathID1.intValue(),
FlowRoutingTableEntry.INSTALLED_FOR_BE);
            bePath1.bCreated = true;
          }

          expireBEpaths(firstNodeID, lastNodeID);
          myServer.sendCongestionAdvisory(srcRouterID, path.getPathID().intValue(),
CongestionAdvisory.GREEN);
                                            gui.sendText("Congestion Advisory GREEN sent
to node " + firstNodeID + ".");
          BasePIB.Path bePath2 = myBasePIB.routingAlgorithm.findPath(srcRouterID,
                                                            destRouterID,
                                                            bePath1,

myBasePIB.routingAlgorithm.SHORTEST_WIDEST_MOST_DISJOINT_PATH);
          if (bePath2 != null)
          {
                                            bePathID2 = bePath2.getPathID();
                                            gui.sendText("Deploying path " +
bePathID2 + " as the new alternate path.");
            if (!bePath2.bCreated)
            {
              myBasePIB.setupPath(bePath2, bePathID2.intValue(),
FlowRoutingTableEntry.INSTALLED_FOR_BE);
              bePath2.bCreated = true;
            }
          }
          else
          {
                                            bePathID2 = bePathID1;
          }//end if
                                            sendTableEntries(srcRouterID, destRouterID,
bePathID1.intValue(), bePathID2.intValue());
        }//end if
      }//end if
      if ((noLocalResolutionPossible) && (!globalCongestion))
      {
```

```java
                                            gui.sendText("\nWARNING!\nWARNING!\nWARNING!");
                                            gui.sendText("G L O B A L   C O N G E S T I O N");
                                            gui.sendText("Initiating global congestion
resolution procedures.");
                                            initiateGlobalCongestionResolution();
          }
          break;

        //no action necessary; the path's been retired
        case BasePIB.Path.RED:
          break;

        default:
          break;
      }
    }
    //has the previous congestion cleared?
    else if (path.bestEffortTrafficCondition == BasePIB.Path.YELLOW)
    {
      int firstNodeID = path.getSrcNodeID();
                        int lastNodeID = path.getDestNodeID();
                        BasePIB.Path primaryPath = primaryPathForThisNodePair(firstNodeID,
lastNodeID);
                        BasePIB.Path alternatePath =
alternatePathForThisNodePair(firstNodeID, lastNodeID);
                        if (path == primaryPath)
                        {
                                if (alternatePath.bestEffortTrafficCondition ==
BasePIB.Path.GREEN)
                                {
                        IPv6Address routerID = (IPv6Address)
myBasePIB.htNodeIDtoRouterID.get(new Integer(firstNodeID));
                        myServer.sendCongestionAdvisory(routerID,
path.getPathID().intValue(), CongestionAdvisory.GREEN);
                        path.congestionCleared();
                                        gui.sendText("\nCongestion cleared on path " +
path.getPathID().intValue() + ".");
                                        gui.sendText("No more congestion for node pair (" +
firstNodeID + "," + lastNodeID + ").");
                                        gui.sendText("Congestion Advisory GREEN sent to node
" + firstNodeID + ".");
                                }
                                else
                                {
                                        gui.sendText("Congestion has cleared on a primary
path " + path.getPathID() + " while alternate path lossy.");
                                        path.congestionCleared();
                                        switchback(firstNodeID, lastNodeID);
                                }
                        }
                        else
                        {
                                path.congestionCleared();
                        }
    }
  }//end reactiveMonitor()


  /**
   * Proactive monitoring takes place during global congestion.
   * @param   path      the path being observed
   * @param   lossRate  the best effort loss rate on that path
   */
  protected void proactiveMonitor(BasePIB.Path path, short lossRate)
  {
    path.bestEffortLossRate = lossRate;//only recorded during active monitoring

    long currentTime = System.currentTimeMillis();
    if (lossRate > myBasePIB.thresholdLossRate)
```

```
      {
        timeLastCongestion = currentTime;
      }
      else if ((currentTime - timeLastCongestion) > (localResolutionTimeout *
myBasePIB.timeScale))
      {
        terminateGlobalCongestionResolution();
        reactiveMonitor(path, lossRate);
                      gui.sendText("\nGlobal congestion resolved!");
                      gui.sendText("Terminating global coneston resolution procedures.");
        return;
      }

      if ((currentTime - timeLastActionTaken) > (localResolutionTimeout *
myBasePIB.timeScale))
      {
                      gui.sendText("\nAttempting to rob from the rich and give to the
poor.");
                      gui.sendText("Calculating fairness variables...");
                      calculateFairnessVariables();
                      boolean robbed = robFromTheRich();
                      boolean gave = giveToThePoor();
        if (robbed || gave)
        {
          timeLastActionTaken = currentTime;
        }
                      else
                      {
                              timeLastActionTaken = currentTime + (10 *
localResolutionTimeout * myBasePIB.timeScale);
                              gui.sendText("No action taken.");
                      }
      }
      else
      {
        reactiveMonitor(path, lossRate);
      }
    }
  //end proactiveMonitor()

  /**
   * Every time a new edge router is discovered, the BE topology is updated
   * and new paths are deployed as necessary.
   */
  protected void updateBEtopology()
  {

    //first, reset the topology
              gui.sendText("Resetting old paths...");
    Enumeration allpaths = myBasePIB.htPaths.elements();
    while (allpaths.hasMoreElements())
    {
      BasePIB.Path thispath = (BasePIB.Path) (allpaths.nextElement());
      if (thispath.bBestEffortTraffic)
      {
        thispath.terminateBestEffortTraffic();
      }
    }
              gui.sendText("reset.");

              beNodePairAdmin(DEPLOY_INITIAL_PATHS);

  }//end updateBEtopology()

  /**
   * When one of a BET agent's path fails, the BEM restores redundancy by
   * deploying a new path.
   * @param   pathID  the remaining path
   */
```

96

```
  private void restoreRedundancy(int pathID)
  {
    BasePIB.Path deadPath = (BasePIB.Path) (myBasePIB.htPaths.get(new Integer(pathID)));
    BasePIB.Path livePath = null;
    int srcNodeID = ((BasePIB.Path) (myBasePIB.htPaths.get(new
Integer(pathID)))).getSrcNodeID();
    int destNodeID = ((BasePIB.Path) (myBasePIB.htPaths.get(new
Integer(pathID)))).getDestNodeID();
                gui.sendText("Affected node pair is (" + srcNodeID + "," + destNodeID +
").");
    IPv6Address srcRouterID = (IPv6Address) (myBasePIB.htNodeIDtoRouterID.get(new
Integer(srcNodeID)));
    IPv6Address destRouterID = (IPv6Address) (myBasePIB.htNodeIDtoRouterID.get(new
Integer(destNodeID)));

    //first, determine the identity of the live path
    Vector bePaths = getThisNodePairsBEpaths(srcNodeID, destNodeID);
    Enumeration thesePaths = bePaths.elements();
    while (thesePaths.hasMoreElements())
    {
      BasePIB.Path thisPath = (BasePIB.Path) (thesePaths.nextElement());
      if (deadPath != thisPath)
      {
        livePath = thisPath;
      }
    }

    //if this was the only path, then nothing can be done
    if (livePath == null)
    {
                    gui.sendText("NO SURVIVING PATH!!!");
      return;
    }

                gui.sendText("Resent surviving path " + livePath.getPathID() + " to reset
the destination.");

    BasePIB.Path newRedundantPath = myBasePIB.routingAlgorithm.findPath(srcRouterID,
                                                       destRouterID,
                                                       livePath,
myBasePIB.routingAlgorithm.SHORTEST_WIDEST_MOST_DISJOINT_PATH);
    //now, attempt to find and send a new alternate path
    if (newRedundantPath != null)
    {
      Integer newRedundantPathID = newRedundantPath.getPathID();
      if (!newRedundantPath.bCreated)
      {
        myBasePIB.setupPath(newRedundantPath, newRedundantPathID.intValue(),
FlowRoutingTableEntry.INSTALLED_FOR_BE);
        newRedundantPath.bCreated = true;
      }
                    gui.sendText("Sending path " + newRedundantPathID + " as the new
alternate path.");
                    gui.sendText("Redundancy restored!");
    }
                else//resend the same path as alternate
                {
                    newRedundantPath = livePath;
                    gui.sendText("Unable to find a redundant path.  Resending the
primary path as alternate.");
                }//end if else

                sendTableEntries(srcRouterID, destRouterID,
livePath.getPathID().intValue(), newRedundantPath.getPathID().intValue());

  }//end restoreRedundancy()

  /**
```

97

```
   * Tests to see if there are two BE routes currently active for this pair.
   * @param    srcNodeID    the source node
   * @param    destNodeID  the destination node
   * @return   whether there are
   */
  private boolean twoBEroutesActive(int srcNodeID, int destNodeID)
  {
    if (getThisNodePairsBEpaths(srcNodeID, destNodeID).size() == 2)
    {
      return true;
    }
    else
    {
      return false;
    }
  }

  /**
   * Expires best effort paths for this node pair.
   * @param    srcNodeID
   * @param    destNodeID
   */
  private void expireBEpaths(int srcNodeID, int destNodeID)
  {
    BasePIB.Path thisPath;

    Enumeration thesePaths = getThisNodePairsBEpaths(srcNodeID, destNodeID).elements();
    while (thesePaths.hasMoreElements())
    {
      thisPath = (BasePIB.Path) (thesePaths.nextElement());
      if ((thisPath.getSrcNodeID() == srcNodeID) && (thisPath.getDestNodeID() ==
destNodeID))
      {
        thisPath.expireBEpath();
      }
    }
  }

  /**
   * Unexpire those BE paths that have been expired
   * past the required time.
   */
  private void unexpireBEpaths()
  {
    bePathAdmin(0, 0, UNEXPIRE_PATHS);
  }

  /**
   * Determines the best effort paths for this node pair.
   * @param    srcNodeID
   * @param    destNodeID
   * @return   best effort paths as a Vector
   */
  private Vector getThisNodePairsBEpaths(int srcNodeID, int destNodeID)
  {
    return bePathAdmin(srcNodeID, destNodeID, GET_PATHS);
  }

  /**
   * Determines if global congestion is occurring.
   * @return whether it's occurring
   */
  protected boolean globalCongestionIsOccurring()
  {
    return globalCongestion;
  }

  /**
   * Initiates global congestion resolution procedures.
```

```
 */
private void initiateGlobalCongestionResolution()
{
  globalCongestion = true;

  BasePIB.Path thisPath;
  BasePIB.PathQoS thisqos;//must be declared here for visibility purposes

            //update loss rate parameters for all BE paths
  bePathAdmin(0, 0, UPDATE_LOSS_RATE);

            gui.sendText("Calculating fairness variables...");
  calculateFairnessVariables();

  timeLastActionTaken = System.currentTimeMillis();
            timeLastCongestion = System.currentTimeMillis();
}

/**
 * Calculates fairness variables to base later
 * actions upon.
 */
private void calculateFairnessVariables()
{
  Vector bepaths = new Vector();
  Vector samples = new Vector();
  IPv6Address thisRouterID;
  int srcNodeID;
  int destNodeID;
  BasePIB.Path thisbepath;
  BasePIB.Path thisPath;
  int count = 0;

            samples = (Vector) (beNodePairAdmin(GET_LOSS_RATES));

  meanLossRate = computeMean(samples);
            gui.sendText("Mean loss rate is " + (meanLossRate/100) + "%.");
  stdLossRateDev = computeStdDev(samples);
            gui.sendText("Loss rate SD is " + (stdLossRateDev/100) + "%.");
}

/**
 * Computes loss rate from this node pair assuming all traffic is on alternate path.
 * @param    srcNodeID
 * @param    destNodeID
 * @return   best effort loss rate
 */
private short lossRateFromThisNodePair(int srcNodeID, int destNodeID)
{
  long timeDeployed = 0;
  short lossRate = 0;
  BasePIB.Path thisPath;
  Vector bepaths = getThisNodePairsBEpaths(srcNodeID, destNodeID);

            if (srcNodeID == destNodeID)
            {
                    return 0;
            }

  Enumeration thesepaths = bepaths.elements();
  while (thesepaths.hasMoreElements())
  {
    thisPath = (BasePIB.Path) thesepaths.nextElement();
    if (thisPath.bBestEffortTraffic && (thisPath.timeBEinitiated > timeDeployed))
    {
      timeDeployed = thisPath.timeBEinitiated;
      lossRate = thisPath.bestEffortLossRate;
    }
  }
```

99

```java
    return lossRate;
  }


  /**
   * Terminates global congestion and proactive monitoring.
   */
  private void terminateGlobalCongestionResolution()
  {
    globalCongestion = false;
  }



  /**
   * Fairness measure that will release resources from those pairs
   * not experiencing congestion.
   * @return   success of operation
   */
  private boolean robFromTheRich()
  {

    return ((Boolean) (beNodePairAdmin(ROB_IF_RICH))).booleanValue();

  }//end robFromTheRich()

  /**
   * Fairness measure that will give more resources to those pairs
   * experiencing undue congestion
   * @return   success of operation
   */
  private boolean giveToThePoor()
  {

    reclaimExpiredPaths();

              return ((Boolean) (beNodePairAdmin(GIVE_IF_POOR))).booleanValue();

  }//end giveToThePoor()

  /**
   * Reclaims expired paths that have no congestion for reuse.
   */
  private void reclaimExpiredPaths()
  {
    BasePIB.Path thisPath = null;
    BasePIB.PathQoS thisPathQoS;

    Enumeration allPaths = myBasePIB.htPaths.elements();
    while (allPaths.hasMoreElements())
    {
      thisPath = (BasePIB.Path) (allPaths.nextElement());
      if (thisPath.bestEffortTrafficCondition == BasePIB.Path.RED)
      {
        if (thisPath.bestEffortLossRate < myBasePIB.thresholdLossRate)
        {
          thisPath.unexpireBEpath();
        }
      }
    }
  }

  /**
   * Resets traffic for a node pair back to the primary path.
   * @param    srcNodeID
   * @param    destNodeID
   */
  private boolean switchback(int srcNodeID, int destNodeID)
  {
              if ((System.currentTimeMillis() - timeLastSwitchback) <
(myServer.getAC_cyclePeriod()))
```

100

```
                {
                        return false;
                }

    BasePIB.Path primaryPath = primaryPathForThisNodePair(srcNodeID, destNodeID);
    BasePIB.Path alternatePath = primaryPathForThisNodePair(srcNodeID, destNodeID);

    IPv6Address srcRouterID = (IPv6Address) (myBasePIB.htNodeIDtoRouterID.get(new
Integer(srcNodeID)));
    IPv6Address destRouterID = (IPv6Address) (myBasePIB.htNodeIDtoRouterID.get(new
Integer(destNodeID)));

                sendTableEntries(srcRouterID, destRouterID,
primaryPath.getPathID().intValue(), alternatePath.getPathID().intValue());

    myServer.sendCongestionAdvisory(srcRouterID, primaryPath.getPathID().intValue(),
CongestionAdvisory.GREEN);
                gui.sendText("Reset traffic split to 100/0 for node pair (" + srcNodeID +
"," + destNodeID + ").");
                gui.sendText("Congestion Advisory GREEN sent to node " + srcNodeID + ".");

                timeLastSwitchback = System.currentTimeMillis();
                return true;
  }

  /**
   * Determines the primary path for this node pair based on deployment time.
   * @param   srcNodeID
   * @param   destNodeID
   * @return  primary path
   */
  private BasePIB.Path primaryPathForThisNodePair(int srcNodeID, int destNodeID)
  {
    BasePIB.Path thisPath;
    BasePIB.Path primaryPath = null;
    long leastRecentTime = System.currentTimeMillis();

    Vector bepaths = getThisNodePairsBEpaths(srcNodeID, destNodeID);

    Enumeration enum = bepaths.elements();
    while (enum.hasMoreElements())
    {
      thisPath = (BasePIB.Path) (enum.nextElement());
      if (thisPath.timeBEinitiated < leastRecentTime)
      {
        leastRecentTime = thisPath.timeBEinitiated;
        primaryPath = thisPath;
      }
    }

    return primaryPath;
  }

  /**
   * Determines the alternate path for this node pair based on deployment time.
   * @param   srcNodeID
   * @param   destNodeID
   * @return  alternate path
   */
  private BasePIB.Path alternatePathForThisNodePair(int srcNodeID, int destNodeID)
  {
    BasePIB.Path thisPath;
    BasePIB.Path alternatePath = null;
    long mostRecentTime = 0;

    if (!twoBEroutesActive(srcNodeID, destNodeID))
    {
      return null;
    }
```

101

```java
    Vector bepaths = getThisNodePairsBEpaths(srcNodeID, destNodeID);

    Enumeration enum = bepaths.elements();
    while (enum.hasMoreElements())
    {
      thisPath = (BasePIB.Path) (enum.nextElement());
      if (thisPath.timeBEinitiated > mostRecentTime)
      {
        mostRecentTime = thisPath.timeBEinitiated;
        alternatePath = thisPath;
      }
    }

    return alternatePath;
}

/**
 * Computes the mean of a set of values.
 * @param    samples  the set of values (must cast to Integer)
 * @return  mean
 */
private double computeMean(Vector samples)
{
  int sum = 0;
  int count = 0;

  Enumeration enum = samples.elements();
  while (enum.hasMoreElements())
  {
    sum += ((Integer) (enum.nextElement())).intValue();
    count ++;
  }
  if (count == 0)
  {
    return 0;
  }
  else
  {
    return sum / count;
  }
}

/**
 * Computes the standard deviation of a set of values.
 * @param    samples  the set of values (must cast to Integer)
 * @return  standard deviation
 */
private double computeStdDev(Vector samples)
{
  int sum = 0;
  int thisElement = 0;
  int count = 0;
  double mean = computeMean(samples);

  Enumeration enum = samples.elements();
  while (enum.hasMoreElements())
  {
    thisElement = ((Integer) (enum.nextElement())).intValue();
    sum += (thisElement - mean) * (thisElement - mean);
    count++;
  }
  if (count == 0)
  {
    return 0;
  }
  else
  {
    return java.lang.Math.sqrt((double) (sum / count));
```

```
    }
  }

  /**
   * Method through which a BE path failure notification is made.
   * @param   failedPathID  ID of the failed path
   */
  protected void handleBEpathFailure(int failedPathID)
  {
                gui.sendText("\nHandling failure of path " + failedPathID + ".");
                BasePIB.Path thisPath = (BasePIB.Path) (myBasePIB.htPaths.get(new
Integer(failedPathID)));
                int srcNodeID = thisPath.getSrcNodeID();
                IPv6Address srcRouterID = (IPv6Address)
(myBasePIB.htNodeIDtoRouterID.get(new Integer(srcNodeID)));
                myServer.sendCongestionAdvisory(srcRouterID,
thisPath.getPathID().intValue(), CongestionAdvisory.RED);
                gui.sendText("Congestion Advisory RED sent to node " + srcNodeID + ".");
                gui.sendText("Attempting to restore redundancy...");
    restoreRedundancy(failedPathID);
  }


        /**
         * Whenever BEM generates new paths for a BE node pair, this method is called
         * to send the table entries and perform the bookkeeping.  Note that entries
         * are always sent in pairs.  This is to force a 100/0 reset on the BET agent
         * end and acceptance of these new entries as active.
         * @param   srcRouterID                         the source router ID
         * @param   destRouterID      the destination router ID
         * @param   primaryPathID     the primary path ID
         * @param   alternatePathID   the alternate path ID
         */
        private void sendTableEntries(IPv6Address srcRouterID, IPv6Address destRouterID,
int primaryPathID, int alternatePathID)
        {
                int destNodeID = ((Integer)
(myBasePIB.htRouterIDtoNodeID.get(destRouterID.toString()))).intValue();

                try
                {
            Enumeration interfaces = vBestEffortDestAdds.elements();
            while (interfaces.hasMoreElements())
            {
              IPv6Address thisInterfaceAdd = IPv6Address.getByName((String)
interfaces.nextElement());
                if (destNodeID == ((BasePIB.InterfaceInfo)
myBasePIB.htInterfaces.get(thisInterfaceAdd.toString())).getNodeID().intValue())
                {
                  myServer.sendBETUpdate(srcRouterID, thisInterfaceAdd, primaryPathID, 0,
0);
                                        BasePIB.Path primaryPath = (BasePIB.Path)
myBasePIB.htPaths.get(new Integer(primaryPathID));
                                        primaryPath.initiateBestEffortTraffic();
                                        primaryPath.timeBEinitiated -= 1;//other parts of
code require primary path to be older
                  myServer.sendBETUpdate(srcRouterID, thisInterfaceAdd, alternatePathID, 0,
0);
                  BasePIB.Path alternatePath = (BasePIB.Path) myBasePIB.htPaths.get(new
Integer(alternatePathID));
                                        alternatePath.initiateBestEffortTraffic();

                    }
            }
                }
                catch (UnknownHostException uhe)
                {
                        System.out.println("UHE thrown by sendTableEntries() in
BestEffortManager.");
```

103

```
                }
        }

        /**
         * All code requiring an all paths iterator is consolidate here.
         * @param   srcNodeID  source node ID
         * @param   destNodeID destination node ID
         * @param   action      byte code defined at beginning of class
         * @return
         */
        private Vector bePathAdmin(int srcNodeID, int destNodeID, byte action)
        {
                Vector bepaths = new Vector();

    BasePIB.PathQoS thisPathQoS;

                Enumeration allPaths = myBasePIB.htPaths.elements();

                while (allPaths.hasMoreElements())
                {
                    BasePIB.Path thisPath = (BasePIB.Path) allPaths.nextElement();

                    switch (action)
                    {
                            case UNEXPIRE_PATHS:
                                    if (thisPath.bestEffortTrafficCondition ==
BasePIB.Path.RED)
                    {
                      if ((System.currentTimeMillis() - thisPath.timeConditionRed) >
                          (PATH_EXPIRATION_TIME * myBasePIB.timeScale))
                      {
                        if (thisPath.unexpireBEpath())
                                        {
                                                        gui.sendText("\nPath " +
thisPath.getPathID() + " has been unexpired.");
                                        }
                      }
                    }
                            break;

                            case GET_PATHS:
                                    if ((thisPath.bBestEffortTraffic) &&
(thisPath.getSrcNodeID() == srcNodeID)
                        && (thisPath.getDestNodeID() == destNodeID))
                    {
                      bepaths.add(thisPath);
                    }
                            break;

                            case UPDATE_LOSS_RATE:
                                    if (thisPath.bBestEffortTraffic)
                    {
                      BasePIB.PathQoS thisqos =
thisPath.getPathQoSArray()[BasePIB.BEST_EFFORT];
                      thisPath.bestEffortLossRate = thisqos.getPacketLossRate();
                    }
                            break;

                            case RECLAIM_PATHS:
                    if (thisPath.bestEffortTrafficCondition == BasePIB.Path.RED)
                    {
                      if (thisPath.bestEffortLossRate < myBasePIB.thresholdLossRate)
                      {
                        thisPath.unexpireBEpath();
                      }
                    }
                            break;

                            default:
```

104

```
                                        break;
                    }//end switch
            }//end while

            return bepaths;

      }//end bePathAdmin()


      /**
       * All code requiring node pair iterator is consolidate here.
       * @param    action    byte code defined at beginning of class
       * @return
       */
      private Object beNodePairAdmin(byte action)
      {
            boolean bResult = false;
            Vector vResult = new Vector();

            BasePIB.Path thisPath = null;
            BasePIB.Path pathToExpire, primaryPath, alternatePath, reclaimPath;
            BasePIB.PathQoS thisPathQoS;

            try
            {
        Enumeration eSources = vBestEffortRouters.elements();
        while (eSources.hasMoreElements())
        {
                            IPv6Address srcRouterID = IPv6Address.getByName((String)
eSources.nextElement());
                            Integer srcNodeID = ((BasePIB.InterfaceInfo)
myBasePIB.htInterfaces.get(srcRouterID.toString())).getNodeID();
                            Enumeration eDestinations = vBestEffortRouters.elements();
          while (eDestinations.hasMoreElements())
          {
            IPv6Address interfaceAddress = IPv6Address.getByName((String)
eDestinations.nextElement());
            Integer destNodeID = ((BasePIB.InterfaceInfo)
myBasePIB.htInterfaces.get(interfaceAddress.toString())).getNodeID();
            IPv6Address destRouterID = (IPv6Address)
myBasePIB.htNodeIDtoRouterID.get(destNodeID);

                              switch (action)
                              {
                                      case DEPLOY_INITIAL_PATHS:
                            //SHORTEST WIDEST PATH is used for the primary path
                            BasePIB.Path bePath1 =
myBasePIB.routingAlgorithm.findPath(srcRouterID,

destRouterID,

null,

myBasePIB.routingAlgorithm.SHORTEST_WIDEST_PATH);
                              if (bePath1 != null)
                              {
                                Integer bePathID1 = bePath1.getPathID();
                                if (!bePath1.bCreated)
                                {
                                  myBasePIB.setupPath(bePath1, bePathID1.intValue(),
FlowRoutingTableEntry.INSTALLED_FOR_BE);
                                    bePath1.bCreated = true;
                                }
                                                gui.sendText("Path " +
bePathID1.intValue() + " deployed as primary for (" + srcNodeID.intValue() + "," +
destNodeID.intValue() + ").");
                                //SHORTEST WIDEST MOST DISJOINT PATH  is used for the
alternate path
```

```java
                                        BasePIB.Path bePath2 =
myBasePIB.routingAlgorithm.findPath(srcRouterID,

destRouterID,

bePath1,

myBasePIB.routingAlgorithm.SHORTEST_WIDEST_MOST_DISJOINT_PATH);
                                        if (bePath2 != null)
                                        {
                                          Integer bePathID2 = bePath2.getPathID();
                                          if (!bePath2.bCreated)
                                          {
                                            myBasePIB.setupPath(bePath2, bePathID2.intValue(),
FlowRoutingTableEntry.INSTALLED_FOR_BE);
                                            bePath2.bCreated = true;
                                          }
                                                                    gui.sendText("Path " +
bePathID2.intValue() + " deployed as alternate.");
                                        }
                                        else
                                        {
                                          bePath2 = bePath1;
                                                                    gui.sendText("No
alternate path available.");
                                        }
                                                            sendTableEntries(srcRouterID,
destRouterID, bePath1.getPathID().intValue(), bePath2.getPathID().intValue());
                                  }//end if
                                        break;

                                  case GET_LOSS_RATES:
                                        if (srcNodeID != destNodeID)
                                        {
                                                vResult.add(new Integer((int)
(lossRateFromThisNodePair(srcNodeID.intValue(), destNodeID.intValue())))));
                                        }
                                        break;

                                  case ROB_IF_RICH:
                                        int leastBandwidth, thisBandwidth;
                                        if
((lossRateFromThisNodePair(srcNodeID.intValue(), destNodeID.intValue()) < (meanLossRate -
stdLossRateDev)) &&
                                        twoBEroutesActive(srcNodeID.intValue(),
destNodeID.intValue()))
                                        {
                                        leastBandwidth = 2000000000;//a large number
                                        Vector bepaths =
getThisNodePairsBEpaths(srcNodeID.intValue(), destNodeID.intValue());
                                        Enumeration enum = bepaths.elements();
                                        while (enum.hasMoreElements())
                                        {
                                          thisPath = (BasePIB.Path) (enum.nextElement());
                                          thisPathQoS =
thisPath.getPathQoSArray()[BasePIB.BEST_EFFORT];
                                          thisBandwidth = thisPathQoS.getAvailableBandwidth();
                                          if (thisBandwidth < leastBandwidth)
                                          {
                                            leastBandwidth = thisBandwidth;
                                            pathToExpire = thisPath;
                                          }
                                        }
                                        srcRouterID = (IPv6Address)
(myBasePIB.htNodeIDtoRouterID.get(srcNodeID));
                                        myServer.sendCongestionAdvisory(srcRouterID,
thisPath.getPathID().intValue(), CongestionAdvisory.RED);
                                                            thisPath.expireBEpath();
                                        bResult = true;
```

106

```
                                                    gui.sendText("Deactivated
path " + thisPath.getPathID() + " for node pair (" + srcNodeID + "," + destNodeID +
").");
                                                    gui.sendText("Robbed from the
rich.");
                                }//end if
                                        break;

                                case GIVE_IF_POOR:
                                        int currentBandwidth,
switchbackBandwidth, reclaimableBandwidth;
                                        if
((lossRateFromThisNodePair(srcNodeID.intValue(), destNodeID.intValue()) > (meanLossRate +
stdLossRateDev)) &&
                                        twoBEroutesActive(srcNodeID.intValue(),
destNodeID.intValue()))
                                                {
                                primaryPath =
primaryPathForThisNodePair(srcNodeID.intValue(), destNodeID.intValue());
                                alternatePath =
alternatePathForThisNodePair(srcNodeID.intValue(), destNodeID.intValue());
                                srcRouterID = (IPv6Address)
(myBasePIB.htNodeIDtoRouterID.get(srcNodeID));
                                destRouterID = (IPv6Address)
(myBasePIB.htNodeIDtoRouterID.get(destNodeID));
                                reclaimPath =
myBasePIB.routingAlgorithm.findPath(srcRouterID,

destRouterID,

                                                                            null,

myBasePIB.routingAlgorithm.SHORTEST_WIDEST_PATH);
                                if (twoBEroutesActive(srcNodeID.intValue(),
destNodeID.intValue()))
                                {
                                  thisPathQoS =
primaryPath.getPathQoSArray()[BasePIB.BEST_EFFORT];
                                  switchbackBandwidth =
thisPathQoS.getAvailableBandwidth();
                                  thisPathQoS =
alternatePath.getPathQoSArray()[BasePIB.BEST_EFFORT];
                                  currentBandwidth = thisPathQoS.getAvailableBandwidth();
                                }
                                else
                                {
                                  thisPathQoS =
primaryPath.getPathQoSArray()[BasePIB.BEST_EFFORT];
                                  currentBandwidth = thisPathQoS.getAvailableBandwidth();
                                  switchbackBandwidth = 0;
                                }
                                if (reclaimPath != null)
                                {
                                  thisPathQoS =
reclaimPath.getPathQoSArray()[BasePIB.BEST_EFFORT];
                                  reclaimableBandwidth =
thisPathQoS.getAvailableBandwidth();
                                }
                                else
                                {
                                  reclaimableBandwidth = 0;
                                }
                                if ((switchbackBandwidth > currentBandwidth) &&
(switchbackBandwidth >= reclaimableBandwidth))
                                {
                                  bResult = switchback(srcNodeID.intValue(),
destNodeID.intValue());
                                                    gui.sendText("Gave to
the poor.");
                                }
```

107

```
                                else if (reclaimableBandwidth > currentBandwidth)
                                {
                                  Vector bepaths =
getThisNodePairsBEpaths(srcNodeID.intValue(), destNodeID.intValue());
                                  Enumeration enum = bepaths.elements();
                                  while (enum.hasMoreElements())
                                  {
                                    thisPath = (BasePIB.Path) (enum.nextElement());
                                    thisPath.terminateBestEffortTraffic();
                                  }

                                  myServer.sendCongestionAdvisory(srcRouterID,
reclaimPath.getPathID().intValue(), CongestionAdvisory.GREEN);
                                                               gui.sendText("Deployed
fatter path " + reclaimPath + " for node pair (" + srcNodeID + "," + destNodeID + ").");
                                  BasePIB.Path bePath2 =
myBasePIB.routingAlgorithm.findPath(srcRouterID,

destRouterID,

reclaimPath,

myBasePIB.routingAlgorithm.SHORTEST_WIDEST_MOST_DISJOINT_PATH);
                                  if (bePath2 != null)
                                  {
                                    Integer bePathID2 = bePath2.getPathID();
                                    if (!bePath2.bCreated)
                                    {
                                      myBasePIB.setupPath(bePath2, bePathID2.intValue(),
FlowRoutingTableEntry.INSTALLED_FOR_BE);
                                      bePath2.bCreated = true;
                                    }

      gui.sendText("Deployed new alternate path " + bePathID2 + " for node pair (" +
srcNodeID + "," + destNodeID + ").");
                                  }
                                  else
                                  {
                                                                    bePath2 =
reclaimPath;
                                  }//end if

      sendTableEntries(srcRouterID, destRouterID, reclaimPath.getPathID().intValue(),
bePath2.getPathID().intValue());
                                  bResult = true;
                                                               gui.sendText("Gave to
the poor.");
                                }//end if
                              }//end if
                                                break;

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

              switch (action)
              {
              case GET_LOSS_RATES:
                      return vResult;

              case ROB_IF_RICH:
```

```
                        return new Boolean(bResult);

            case GIVE_IF_POOR:
                        return new Boolean(bResult);

            default:
                        return null;
            }

    }//end beNodePairAdmin()

}
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX D: BEST EFFORT TABLE AGENT SOURCE CODE

```
//25Feb02[Wofford] - Sudafed renamed Congestion Advisory.
//31Jan02[Wu] - repackaged
//09Oct01[Wofford]  - created

//[cw]

package org.saamnet.saam.agent.router;

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

import org.saamnet.saam.control.*;
import org.saamnet.saam.agent.*;
import org.saamnet.saam.router.*;
import org.saamnet.saam.net.*;
import org.saamnet.saam.event.*;
import org.saamnet.saam.message.*;
import org.saamnet.saam.gui.*;
//import com.objectspace.jgl.HashMap;//[cw]may be overkill for this class

/**
 * The BestEffortTable 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.  It is "smarter" than a
 * FlowRoutingTable, though, in that it will actually make decisions
 * independent of examining a single entry.
 */
public class BestEffortTable extends Hashtable implements TableResidentAgent,
                MessageProcessor
{

  //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 = //-crcp generic message registration
  {
    Message.BEST_EFFORT_TBL_ENTRY,
    Message.CONGESTION_ADVISORY
  };

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

        //The BestEffortTable acts autonomously at various intervals depending
        //on most recent information in Congeston Advisory messages for a server.  If a
        //TrafficDestination is congested, it will "redirect" traffic
        //periodically to an alternate route.  If the congestion has cleared,
        //then it will gradually revert until all traffic is carried by the
        //primary route.  These constants allow for tuning performance.
        //Philosophically, when you have congestion, you want to act QUICKLY.
        //When the congestion clears, gradually revert to the primary route.
        //The primary route is often more desirable AND it's best to have
        //maximum room available on the alternate route to handles extra traffic.
        private int timeScale;
  private final static int REDIRECT_INTERVAL = 200;//should always be equal to AC_Cycle
time
  private final static int REVERT_INTERVAL = 1800000;//30 minutes
```

111

```java
  /**
   * Constructs a BestEffortTable.
   */
  public BestEffortTable()
  {
    //super(true);
  }


        /**
   * A TrafficDestination is a data structure used by the BestEffortTable to
         * track information on a per-destination basis.  Notably, it holds two
         * key arrays.  Array currentSplit holds the split in percentage over the
         * number of routes being used.  Array routeInstalled tracks whether those
         * routes have been received from the server and installed by BestEffortTable.
         * It is twice as big in order to hold a full complement of spare routes.
   */
        public static class TrafficDestination
        {
                public IPv6Address destination;
    public int[] currentSplit = new int[MAX_ROUTES];//the traffic split
        public boolean[] routeInstalled = new boolean[2 * MAX_ROUTES];
        public int primaryRoute;//array index pointer for primary route
        public int nextEntry;//array index pointer for where to install next route

                public byte trafficCondition;
                public boolean isUsingAlternateRoute;
        public long timeLastRedirect;
                public long timeLastRevert;

        public TrafficDestination(IPv6Address address)
        {
                destination = address;
        primaryRoute = 0;
        nextEntry = 0;
        currentSplit[primaryRoute] = 100;
        trafficCondition = CongestionAdvisory.GREEN;
                        isUsingAlternateRoute = false;
        timeLastRedirect = 0;
        }
        }

  /**
   * Required install method of the ResidentAgent interface.
   *
   * @param controlExec The ControlExecutive on the router this agent
   *        is being installed on.
   * @param String instanceName
   * @param String [] parameters - array of parameters 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");
    int[] columnWidths = {210, 120, 120};
    gui = new TableGui(controlExec.mainGui.getContentPanel(), instanceName, columnLabels,
columnWidths);
    controlExec.addTableGui(gui); //-crcy
    this.controlExec = controlExec;
    controlExec.registerMessageProcessor(myMessages, this);
                timeScale = controlExec.getTimeScale();
  }


  /**
   * Required uninstall method of the ResidentAgent interface.
```

```
   */
  public void uninstall(){
    clear();
  }

  /**
   * The communication method through which ResidentAgent 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;
  }

  /**
   * 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 BE traffic is congested to this destination, redirect traffic to alternate path
    if (trafDest != null) //may be no such entry yet; see RoutingAlogrithm
                {
                        if ((trafDest.trafficCondition == CongestionAdvisory.YELLOW) &&
                  ((System.currentTimeMillis() - trafDest.timeLastRedirect) >
                                          (REDIRECT_INTERVAL * timeScale)))
        {
                redirect(trafDest);
        }
                        if ((trafDest.isUsingAlternateRoute) && (trafDest.trafficCondition
== CongestionAdvisory.GREEN) &&
                          ((System.currentTimeMillis() - trafDest.timeLastRevert) >
                                  (REVERT_INTERVAL * timeScale)))
                        {
                                revert(trafDest);
                        }

        int serialNo = trafDest.primaryRoute;
        int split = 0;//0%
        int percentile = bucketMap * 10 + 10;//i.e. f(0)=10%...f(9)=100%

        for (int counter = 0; counter < MAX_ROUTES; counter++)
        {
                split += trafDest.currentSplit[counter];
        if (percentile <= split)
        {
                serialNo = (serialNo + counter) % (MAX_ROUTES * 2);
                break;
        }
        }

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

113

```
/**
* Returns true if the BET contains an entry indexed by destination
* address andfalse otherwise.
* @param destAddr, a particular destination IP address
* @return whether or not the BET contains an entry indexed by the destination address
*/
public boolean hasEntry(IPv6Address destAddr)
{
  if (destinationList.get(destAddr.toString()) != null)
        {
                return true;
        }
        else
        {
                return false;
        }
}

/**
* 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 getTable()

/**
 * Required method for ResidentAgents for state transfer
 * @param    replacement the ResidentAgent replacement
 */
public void transferState (ResidentAgent replacement)
{
  for (Enumeration e = elements(); e.hasMoreElements();)
  {
     replacement.receiveState((BestEffortTableEntry) e.nextElement());
  }
}


/**
 * Required method for ResidentAgents to receive state.
 * @param    message a BETE (one at a time from transferState())
 */
public void receiveState (Message message){
```

```
    add((BestEffortTableEntry) message);
  }


  /**
   * Required method for a TableResidentAgent.
   * Not used by BET.
   * @param   res FlowResponse
   */
  public void receiveFlowResponse (FlowResponse res) { }


        /**
         * BestEffortTable process two types of messages, BEST_EFFORT_TBL_ENTRY and
CONGESTION_ADVISORY.
         * For BEST_EFFORT_TBL_ENTRY, it adds the entry and makes a new TrafficDestination
   * if it does not have this destination on file.  For CONGESTION_ADVISORY, it updates
the
   * congestion condition for the TrafficDestination using that pathID.
   * @param   message  CongestionAdvisory from BestEffortManager on server
        */
  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)
                    {
                      System.out.println("BestEffortTable Error: can't create local BETE."
+ uhe);
                    }
        //check to see if this is a known destination
                    if (destinationList.containsKey(betentry.getDestAddr().toString()))
                    {
                       TrafficDestination trafDest = (TrafficDestination)
destinationList.get(betentry.getDestAddr().toString());
                       betentry.serialNo = trafDest.nextEntry;
            //check to see if a new complement of routes is being received
            //if so, reset previous splits to 0 and mark next route as primary
                                    boolean resetRoutes = (trafDest.nextEntry -
trafDest.primaryRoute == MAX_ROUTES)
                                                || (trafDest.primaryRoute -
trafDest.nextEntry == MAX_ROUTES);
                                    if (resetRoutes)
                                    {
                                            for (int i = 0; i < MAX_ROUTES; i++)
                                            {
                                                    int index = (trafDest.primaryRoute +
i) % (2 * MAX_ROUTES);
                                                    String key =
betentry.getDestAddr().toString() + index;
                                                    BestEffortTableEntry zeroedentry =
(BestEffortTableEntry) get(key);
                                                    zeroedentry.split = 0;
                                                    trafDest.currentSplit[i] = 0;
                                            }
                                            betentry.split = 100;
                                            trafDest.currentSplit[0] = 100;
                                            trafDest.primaryRoute =
(trafDest.primaryRoute + MAX_ROUTES) % (2 * MAX_ROUTES);
                                            trafDest.isUsingAlternateRoute = false;

                                    }
                                    else //this is not a new primary route
```

115

```
                                        {
                      betentry.split = 0;
                                        }
                  add(betentry);
                  trafDest.routeInstalled[trafDest.nextEntry] = true;
                  trafDest.nextEntry = (trafDest.nextEntry + 1) % (2 * MAX_ROUTES);
                              }
                else //need to start tracking this new destination
                {
                      TrafficDestination trafDest = new
TrafficDestination(betentry.getDestAddr());
                  destinationList.put(betentry.getDestAddr().toString(), trafDest);
                  betentry.serialNo = trafDest.nextEntry;
                  betentry.split = 100;
                  add(betentry);
                  trafDest.routeInstalled[trafDest.nextEntry] = true;
                  trafDest.nextEntry = (trafDest.nextEntry + 1) % (2 * MAX_ROUTES);
                }
        //this is the server's way of granting edge router permission
                  controlExec.acceptEdgeTraffic();
             break;

                case Message.CONGESTION_ADVISORY:
        CongestionAdvisory pill = new CongestionAdvisory(message.getBytes());
        //determine affected path
        int affectedPathID = pill.getPathID();
                  Enumeration e = elements();
                  while (e.hasMoreElements())
                  {
                      BestEffortTableEntry betentry1 = (BestEffortTableEntry)
e.nextElement();
          if (betentry1.getPathMap() == affectedPathID)
          {
              TrafficDestination trafDest = (TrafficDestination)

destinationList.get(betentry1.getDestAddr().toString());
            //update the traffic condition
            trafDest.trafficCondition = pill.pathCondition();
            //if RED, then route all traffic to unaffected path
                                        if (pill.pathCondition() ==
CongestionAdvisory.RED)
                                        {
                                            int unaffectedSerialNo;
                                            int serialNo =
betentry1.getSerialNo();

                                            if (serialNo ==
trafDest.primaryRoute)

                                            {
                                                    unaffectedSerialNo =
(serialNo + 1) % (2 * MAX_ROUTES);
                                            }
                                            else
                                            {
                                                    unaffectedSerialNo =
(serialNo - 1) % (2 * MAX_ROUTES);
                                            }
                                            String unaffectedKey =
trafDest.destination.toString() + unaffectedSerialNo;
                                            BestEffortTableEntry unaffectedEntry
= (BestEffortTableEntry) get(unaffectedKey);

        betentry1.setPathMap(unaffectedEntry.getPathMap());
                                            gui.fillTable(getTable());
                                        }
          }
                  }
        break;

                default:
```

116

```
                  break;
            }


  }//End processMessage()

  /**
   * Redirects one bucket of traffic from the primary to alternate path.
   * @param    trafDest the traffic destination
   * @return   success of operation
   */
  private boolean redirect(TrafficDestination trafDest)
  {
                int primaryRoute = trafDest.primaryRoute;
                int alternateRoute = (trafDest.primaryRoute + 1) % (2 * MAX_ROUTES);

                if ((trafDest.currentSplit[0] >= 10) &&
(trafDest.routeInstalled[alternateRoute]))
                {
                        String primaryKey = trafDest.destination.toString() + primaryRoute;
        BestEffortTableEntry primaryEntry = (BestEffortTableEntry) get(primaryKey);
                        primaryEntry.setSplit(primaryEntry.getSplit() - 10);
                        trafDest.currentSplit[0] -= 10;

                        String alternateKey = trafDest.destination.toString() +
alternateRoute;
                        BestEffortTableEntry alternateEntry = (BestEffortTableEntry)
get(alternateKey);
                        alternateEntry.setSplit(alternateEntry.getSplit() + 10);
                        trafDest.currentSplit[1] += 10;

                        gui.fillTable(getTable());

                        trafDest.isUsingAlternateRoute = true;

                        trafDest.timeLastRedirect = System.currentTimeMillis();

                return true;
                }
                else
                {
                        trafDest.timeLastRedirect = System.currentTimeMillis();

                        return false;
                }
  }

  /**
   * Reverts one bucket of traffic back to the primary path.
   * @param    trafDest  the traffic destination
   * @return   success of operation
   */
      private boolean revert(TrafficDestination trafDest)
      {
                int primaryRoute = trafDest.primaryRoute;
                int alternateRoute = (trafDest.primaryRoute + 1) % (2 * MAX_ROUTES);

                if (trafDest.currentSplit[0] <= 90)
                {
                        String primaryKey = trafDest.destination.toString() + primaryRoute;
        BestEffortTableEntry primaryEntry = (BestEffortTableEntry) get(primaryKey);
                        primaryEntry.setSplit(primaryEntry.getSplit() + 10);
                        trafDest.currentSplit[0] += 10;

                        String alternateKey = trafDest.destination.toString() +
alternateRoute;
                        BestEffortTableEntry alternateEntry = (BestEffortTableEntry)
get(alternateKey);
                        alternateEntry.setSplit(alternateEntry.getSplit() - 10);
```

117

```
                              trafDest.currentSplit[1] -= 10;

                              gui.fillTable(getTable());

                              if (trafDest.currentSplit[1] == 0)
                              {
                                      trafDest.isUsingAlternateRoute = false;
                              }

                              trafDest.timeLastRevert = System.currentTimeMillis();

                      return true;
                      }
                      else
                      {
                              trafDest.timeLastRedirect = System.currentTimeMillis();

                              return false;
                      }
      }

/**
 * Required method for MessageProcessors.
 * @return message types processed
 */
public byte[] getMessageTypes()
{
  return myMessages;
}


/**
 * Required method for SaamListeners.
 * @param    se   event received
 */
public void receiveEvent(SaamEvent se){ }

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

/**
 * 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()
```

118

```
}//end BestEffortTable
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX E: CONGESTION ADVISORY MESSAGE SOURCE CODE

```
//25Feb02[Wofford] - Sudafed renamed Congestion Advisory.
//31Jan02[Wu] - repackaged
//29Jan02[Wofford] - Rewrote to conform to standard format (like DCM)
//07Dec01[Wofford] - Created.

package org.saamnet.saam.message;

import java.net.UnknownHostException;
import java.util.*;
import org.saamnet.saam.net.*;
import org.saamnet.saam.util.*;

/**
 * CongestionAdvisory is how a server tells a router whether or not it is
 * experiencing congestion of its best effort traffic.  It is
 * also how it tells the router that congestion is relieved.
 */
public class CongestionAdvisory extends Message{

  public static final byte GREEN  = 0;
  public static final byte YELLOW = 1;
  public static final byte RED    = 2;

  //total length (in bytes) of fields below
  private final static short CADV_LENGTH = (short) (4 + 1);

  int pathID;
  byte pathCondition;

  public CongestionAdvisory(int pathID, byte pathCondition)
  {
    super(Message.CONGESTION_ADVISORY);
    this.pathID = pathID;
    this.pathCondition = pathCondition;

    bytes = Array.concat(type, PrimitiveConversions.getBytes(CADV_LENGTH));
              bytes = Array.concat(bytes, PrimitiveConversions.getBytes(pathID));
              bytes = Array.concat(bytes, pathCondition);
  }

  public CongestionAdvisory (byte[] bytes)
  {
    super(Message.CONGESTION_ADVISORY);
    this.bytes = bytes;

              int index = 3;//skip type and length fields

    pathID = PrimitiveConversions.getInt(Array.getSubArray(bytes, index, index +4));
    index += 4;

    pathCondition = bytes[index];

  }//end byte array based Constructor

  public int getPathID()
  {
    return pathID;
  }

  public byte pathCondition()
  {
        return pathCondition;
  }
```

```java
   public String toString()
   {
     String advisory = "Congestion Advisory Message:" +
                    "\n  Path ID = " + pathID +
                    "; Traffic condition = ";
               switch (pathCondition)
               {
               case GREEN:
                      advisory += "GREEN";
                      break;
               case YELLOW:
                      advisory += "YELLOW";
                      break;
               case RED:
                      advisory += "RED";
                      break;
               default:
                      break;
               }

     return advisory;
   }

}
```

APPENDIX F: MODIFICATIONS TO ROUTING ALGORITHM SOURCE CODE

```java
/**
 * Forwards the outbound packet to the appropriate Interface
 * @param packet A byte array representation of the outbound packet.
 */
 public void forwardPacket(IPv6Packet packet)
 {
   int MIN_APP_PATH_ID = 65; // mirroring definition in server/BasePIB.java

   byte INT_SERV = 0x01;      // last two bits of ToS field
   byte DIFF_SERV = 0x02; //[cw] used to say 0x00 which is not cons. w/ FlowGenerator
   byte BEST_EFF = 0x00; //[cw] BE packets will have default ToS setting

   IPv6Header v6Header = packet.getHeader();
   IPv6Address dest = v6Header.getDest();

   byte ToS = (byte) (v6Header.getToS() & 0x03); // obtain the last two bits

   int flowLabel = v6Header.getFlowLabel();
   int pathID = flowLabel & 0x00000FFF;  // obtain the last 12 bits.

   byte sl;
   if (ToS == INT_SERV)
   {
     if (pathID < MIN_APP_PATH_ID) // Path IDs 0-64 are reserved for signaling channels.
       sl = Interface.CTRL_TRAFFIC_SL;
     else
       sl = Interface.INT_SERV_SL;
   }
   else if (ToS == DIFF_SERV)
   {//[cw] changed from " =Interface.BEST_EFFORT_SL" below
     sl = Interface.DIFF_SERV_SL;  // [GX] Need to handle DS later; check PHB bits.
   }
   else if (ToS == BEST_EFF) //[cw] added branch for BE traffic
   {
       sl = Interface.BEST_EFFORT_SL;
   }
   else
   {
      gui.sendText("\nUnknown ToS!  Quit forwarding this packet.");
     return;
   }

                //[cw] This is the control structure that handles best effort traffic.
Basically, for an
     //[cw] inbound BE packet to be routed, this router must be an edge router AND have an
entry
     //[cw] in its BE table for that destination IP address.  If this is not the case,
than the
     //[cw] router makes an edge notification and drops packets to that address until it
gets an
     //[cw] entry.  It will try again to notify the server if 20 seconds elapse with no
table
     //[cw] entry.  If this router is a protected core router (forbidden to handle edge
traffic),
     //[cw] it will always drop BE packets.  Note that after being routed by this router,
they
     //[cw] are assigned to a path and essentially partially encapsulated.  That is, the
logic
     //[cw] below only applies to "naked packets", those without a pathID.  After they are
assigned
     //[cw] a pathID at the ingress edge router, they are routed solely on that.  Their
ToS bits
```

123

```
    //[cw] still indicate BE, but this now only matters for bandwidth provision and
queueing.
    if (ToS == BEST_EFF && (pathID == 0))
    {
        if (controlExec.isEdgeRouter())
        {
        int bucketMap = java.lang.Math.abs(v6Header.getSource().toString().hashCode()) %
10;
        Message message = (Message) (new BestEffortTableEntry(dest, 0, 0, bucketMap));
        BestEffortTableEntry betentry = (BestEffortTableEntry)
bestEffortTable.query(message);
        if (betentry != null)
        {
                                    pathID = betentry.getPathMap();
            IPv6Header newHdr = new IPv6Header(ToS, pathID, v6Header.getSource(), dest);
                IPv6Packet newPkt = new IPv6Packet(newHdr, packet.getUDPHeader(),
packet.getPayload());
                packet = newPkt;
        }
        else
        {
                //if it hasn't been at least 20s, give the server some time to
            //update the BE topology and deploy routes; drop packets until then
            if ((System.currentTimeMillis() - timeLastEdgeNotifSent) > (20000 *
controlExec.getTimeScale())))
            {
                controlExec.sendEdgeNotification(dest);
              timeLastEdgeNotifSent = System.currentTimeMillis();
            }
            requeueBestEffPkt(packet);
        }
        }
        else
        {
          //if it hasn't been at least 20s, give the server some to
          //update the BE topology and deploy routes; drop packets until then
          if (!controlExec.isProtectedCore() &&
              ((System.currentTimeMillis() - timeLastEdgeNotifSent) > (20000 *
controlExec.getTimeScale()))))
          {
                controlExec.sendEdgeNotification(controlExec.getRouterId());
            controlExec.sendEdgeNotification(dest);
            timeLastEdgeNotifSent = System.currentTimeMillis();
            requeueBestEffPkt(packet);
          }
          else if (!controlExec.isProtectedCore())
          {
            requeueBestEffPkt(packet);
          }
        }
    }//[cw] end control structure for best effort traffic

    IPv6Address nextHop = null;
    Interface outboundInterface = null;
    Message message = (Message) (new FlowRoutingTableEntry(pathID));

    if (pathID >= MIN_APP_PATH_ID)
    { // This is an application flow
      //[cw] changed "ent" to "frtentry" inside these brackets
      FlowRoutingTableEntry frtentry = (FlowRoutingTableEntry)
flowRoutingTable.query(message);
      if (frtentry != null)
      {
        nextHop = frtentry.getNextHop();
        outboundInterface = (Interface) interfaces.elementAt(frtentry.getInterfaceNum());
      }
      else
      {
        gui.sendText("\nNo routing entry for this application packet! (pathID = " +
```

```
                    pathID + "). Quit forwarding this packet.");
        return;
      }
    }
    else if (packet.queryPossibleMessageType() == Message.DCM)
    { // For DCMs, use destination in IPv6 header; Why not just broadcast?
      nextHop = dest;
      outboundInterface = Interface.getMatchInterface(interfaces, nextHop);

      gui.sendText("Routing a DCM packet (pathID = " + pathID +
                   ")  nextHop = " + nextHop.toString());
    }
    else if (pathID % 2 == 1)
    { //[cw] changed "entry" to "stentry" in this scope
      // Need to check RBCCTs for router-bound signaling packets
      // Router-bound signaling packets carry server root path IDs

      // First find the right server entry
      ServerTable table =  controlExec.getServerTable();
      //[cw] changed "entry" to "stentry" in this scope
      ServerTableEntry stentry = table.getEntryByRootPathId(pathID);

      if (stentry == null)
      {
        gui.sendText("\nNo server entry exists with matching root path id! (pathId = " +
                     pathID + "). Quit forwarding this packet.");
        return;

      }

      // Then look up the RBCCT for that server to find next hop
      RouterBoundCtrlChTable rbccTable = stentry.getRouterBoundCtrlChTable();
      RouterBoundCtrlChTableEntry rbccEntry = rbccTable.get(dest);

      if (rbccEntry != null)
      {
        nextHop = rbccEntry.getNextHop();
        outboundInterface = Interface.getMatchInterface(interfaces, nextHop);
      }
      else
      {
        gui.sendText("\nNo routing entry for this router-bound signaling packet! (pathID
= " +
                     pathID + "). Quit forwarding this packet.");
        return;
      }
    }
    else
    { //[cw] changed "ent" to "frtentry" in this scope
      FlowRoutingTableEntry frtentry = (FlowRoutingTableEntry)
flowRoutingTable.query(message);
      if (frtentry != null)
      {
        nextHop = frtentry.getNextHop();
        outboundInterface = (Interface) interfaces.elementAt(frtentry.getInterfaceNum());
      }
      else
      {
        gui.sendText("\nNo routing entry for this server-bound signaling packet! (pathID
= " +
                     pathID + "). Quit forwarding this packet.");
        return;
      }
    }

    //Now use ARP to determine the MAC address of the next hop.
    //It would be better if this ARP cache lookup is done by the
    //Interface layer. This is a temporary solution.
```

125

```
  message = (Message) (new ARPCacheEntry(nextHop));
  //[cw] changed "entry" to "arpcentry" in this scope
  ARPCacheEntry arpcentry = (ARPCacheEntry) arpCache.query(message);

  try
  {
    byte nextMAC = arpcentry.getNextMAC();
    gui.sendText("  nextMAC: " + nextMAC);

    if(v6Header.getSource().toString().equals(IPv6Address.DEFAULT_HOST))
    {
      v6Header.setSource(outboundInterface.getInfo().getIPv6());
      packet.setHeader(v6Header);
    }

    gui.sendText("  Source: " + v6Header.getSource());
    gui.sendText("  Dest: " + v6Header.getDest());
    gui.sendText("  Forwarding packet to: " + outboundInterface);

    byte[] outboundPacket = Array.concat(nextMAC,packet.getBytes());

    //send a SaamEvent to the appropriate outbound interface.
    //This SaamEvent contains the service level among other things.
    ProtocolStackEvent event = new ProtocolStackEvent(
        toString(),
        this,
        ProtocolStackEvent.getFromRoutingAlgorithmToInterfaceChannel(
                           interfaces.indexOf(outboundInterface)),
        outboundPacket,
        sl,
        nextHop);

    try
    {
      controlExec.talk(event);
    }
    catch (ChannelException tde)
    {
      gui.sendText(tde.toString());
    }
  }
  catch (NullPointerException npe)
  {
    gui.sendText("Next Hop is not in the ARPCache");
    gui.sendText("Packet Dropped\n");
  }

}//end forwardPacket()

      //[cw]
/**
 * Requeues packets not immediately handled while edge router
 * promotion or route deployment takes place.
 * @param   packet  the packet that would have been dropped
 * @return   success of operation
 */
    private boolean requeueBestEffPkt(IPv6Packet packet)
    {
  //requeueing is only allowed one second out of every six
            long checkTime = System.currentTimeMillis() - timeBestEffAmnesty;
            if (checkTime < 0)
            {
                    return false;
            }
            else if (checkTime < (1000 * controlExec.getTimeScale()))
            {
                    ProtocolStackEvent event = new ProtocolStackEvent(
        toString(),
        this,
```

126

```
      ProtocolStackEvent.getFromNICToInterfaceChannel(0),
      packet.getBytes());
try
                {
  controlExec.talk(event);
}
                catch(ChannelException tde)
                {
  System.out.println(tde.toString());
}
                return true;
        }
        else
        {
                timeBestEffAmnesty += 5000 * controlExec.getTimeScale();
                return false;
        }
  }
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX G: MODIFICATIONS TO BASE PATH INFORMATION BASE SOURCE CODE

```
  //[cw]
  public void processEdgeNotification (EdgeNotification edgeNotif)
  {
    myBestEffortManager.processEdgeNotification(edgeNotif);
  }

  // [PS] - created to support inter-service borrowing
  /**
   * Refreshes the QoS information of a given path.
   * @param path the path to update.
   * @param deltaDelay[] an array with the delay variations per Service Level.
   * @param deltaLossRate[] an array with the loss rate variations per SLevel.
   * @return void.
   */
  protected void refreshPathQoS(
    Path path,
    short [] deltaDelay,
    short [] deltaLossRate)
  {
    testMsg("refreshPathQoS(" + path.getPathID().toString() + ")");

    PathQoS pathQoS[] = path.getPathQoSArray();

    int [][] pathAvailableBW = pathBandwidth(path);

    //For every Service Level, check the QoS attributes of this Path
    for (byte sl = 0; sl < NUM_OF_SERVICE_LEVELS - 1; sl++)
    {
      //Set path QoS parameters
                     if (sl == INT_SERV || sl == DIFF_SERV) //Xie-corey: remove CTRL/BE
from the bandwidth update
                         {
        pathQoS[sl].setAvailableBandwidth(pathAvailableBW[0][sl]);
        pathQoS[sl].setAvailableBandwidthIncludingBorrowing(pathAvailableBW[1][sl]);
                         }
      pathQoS[sl].updatePathDelay(deltaDelay[sl]);
      pathQoS[sl].updatePathLossRate(deltaLossRate[sl]);
    }//end for-loop through all service levels

              //[cw]
    if (myBestEffortManager.globalCongestionIsOccurring())
    {
      myBestEffortManager.proactiveMonitor(path,
pathQoS[BEST_EFFORT].getPacketLossRate());
    }
    else
    {
      myBestEffortManager.reactiveMonitor(path,
pathQoS[BEST_EFFORT].getPacketLossRate());
    }
  } // end of refreshPathQoS(Path, dDelay, dLR)

  //[PS] - redesigned to support of inter-service borrowing and to improve
  //efficiency
  /**
   * Receives and processes a LSA - extracts the vector of ISAs, determines the
   * type of each ISA (Add, Remove or Update) and processes each of them in
   * sequence according to their type.
   * @param LSA a LinkStateAdvertisement object.
   * @return void.
   */
  public void processLSA (LinkStateAdvertisement LSA)
  {
```

```java
//[cw] constructor is TOO EARLY a place to call for this value
if ((!iKnowWhatTimeItIs) || (timeScale > 999))
{
  timeScale = controlExec.getTimeScale();
  iKnowWhatTimeItIs = true;
}

// Increase the LSA counter
iLsaCounter++;
Vector interfaceSAs = LSA.getISAvector();
IPv6Address routerID = LSA.getSenderRouterID();

testMsg("processLSA()");

boolean isNewRouter = false;
Integer thisNodeID = (Integer) htRouterIDtoNodeID.get(routerID.toString());

String strNodeID =
  (thisNodeID == null) ? "new router" : thisNodeID.toString();
gui.sendText(
  "\nProcess LSA number " + iLsaCounter + "\n" +
  "\tRouter / Node ID:  \t" + routerID + "  /  " + strNodeID);

if (thisNodeID == null) // LSA is from a new NODE
{
  // If it is a New Node, assign it a new NodeID
  // then place the router in the router/node and node/router look-up tables
  thisNodeID = new Integer(iNextNodeID++);

  htRouterIDtoNodeID.put(routerID.toString(), thisNodeID);
  htNodeIDtoRouterID.put(thisNodeID, routerID);
  isNewRouter = true;
}// End if for new router detection

// Step through the list of ISA's and process each according to its type
Enumeration enumISAs = interfaceSAs.elements();
while (enumISAs.hasMoreElements())  // Step through each ISA in turn
{
  testMsg("Start processing new ISA");
  InterfaceSA thisISA  = (InterfaceSA) enumISAs.nextElement();
  byte type = thisISA.getInterfaceSAType();

  // Use simple if structure to determine the type of ISA
  switch (type)
  {
    case InterfaceSA.UPDATE: //Update Interface type

      try
      {
        testMsg("ISA of type UPDATE. Going to update interface...");
        gui.sendText("ISA of type UPDATE. Going to update interface...");
        updateInterface(thisISA, thisNodeID, routerID, isNewRouter);
        isNewRouter = false;
      }
      catch (Exception e)
      {
        gui.sendText("!!!An exception occurred in updating interface!");
        e.printStackTrace();
      }
      break;

    case InterfaceSA.REMOVE: // Remove Interface type
      try
      {
        testMsg("ISA of type REMOVE. Going to remove interface");
        gui.sendText("ISA of type REMOVE. Going to remove interface...");
        removeInterface(thisISA, thisNodeID, routerID);
        displayPIB();
      }
```

130

```
        catch (Exception e)
        {
          gui.sendText("!!!An exception occurred in removing interface!");
          e.printStackTrace();
        }
        break;

      case InterfaceSA.SILENT: //[TW] Silent Interface type
        try
        {
          gui.sendText("ISA of type SILENT.");
          //[cw] The solution developed below is conservative pending
          // further investigation of how to administer and rebuild the
          // PIB in this scenario.  For BE, the path's BE boolean is never
          // reset so this path will never be assigned BE traffic again.
          // When the BET receives code RED, it erases that path and replaces
          // it with the alternate for that destination.  So...if the SILENT
          // message was due to a temporary network hiccup, resources will be
          // wasted.  Somehow, the PIB needs to "heal" itself here.  For now,
          // the working solution is to create a new boolean member of inner
          // class Path and reference that to determine if the path is usable.
          //[cw] advise affected edge routers of broken BE paths
          IPv6Address sInterfaceAdd = thisISA.getInterfaceIP();
          InterfaceInfo sInterfaceInfo = (InterfaceInfo)
(htInterfaces.get(sInterfaceAdd.toString()));
          Enumeration allAffectedPaths = sInterfaceInfo.getPathIDs().elements();
          while (allAffectedPaths.hasMoreElements())
          {
            Integer thisPathID = (Integer) allAffectedPaths.nextElement();
            Path thisPath = (Path) htPaths.get(thisPathID);
            if ((thisPath.bBestEffortTraffic) && thisPath.bConnected)
            {
              myBestEffortManager.handleBEpathFailure(thisPathID.intValue());
            }
                                            thisPath.bConnected = false;//[cw]
the path is no longer connected
          }

          //actions to take left for further research
        }
        catch (Exception e)
        {
          gui.sendText("!!!An exception occurred in processing interface slient
message!");
          e.printStackTrace();
        }
        break;

      default:  // Undefined type of ISA
        gui.sendText("Undefined type of ISA.");
    }

  } // End while statement processing vector of ISAs

} // End processLSA()

//[cw]
/**
 * Implementation of the First Shortest Path algorithm.
 * @param srcRterID the IPv6 address of the source router.
 * @param destRterID the IPv6 address of the destination router.
 * @return the required path or null if no path was found.
 */
private Path findPathFSP(IPv6Address srcRterID, IPv6Address destRterID)
{
  testMsg("findPathFSP()");

  InterfaceInfo interfaceInformation =
      (InterfaceInfo) htInterfaces.get(srcRterID.toString());
```

```java
    int sourceNodeID = ((InterfaceInfo) htInterfaces
                        .get(srcRterID.toString())).getNodeID().intValue();

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

    Path bestPath = null;

    Hashtable table = new Hashtable();

    //labeled compound statement
    stop:
    {
      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
            bestPath = (Path) htPaths.get(currentPathID);

            if ((!bestPath.bBestEffortTraffic) && (bestPath.bestEffortTrafficCondition
!= Path.RED))
            {

              int availableBandwidth =
                bestPath.getPathServiceLevelQoS(BEST_EFFORT)
                  .getAvailableBandwidth();

              //
              // If available BW is greater than a minimum, the flow is admited
              if (availableBandwidth >=  Best_Effort_Minimum_Bandwidth) //Xie-jan02
              {

                gui.sendText(
                  "\t    The selected path is:\t" + bestPath.toString() + "\n" +
                  "\t    Available bandwidth: \t" + availableBandwidth + "kpbs");

                break stop;
              }//End of if structure
            }
            else
            {
              bestPath = null;
            }

          }//End of while-loop

        }//End of if structure

      }//End of for-loop

    }//End of labeled stop structure

    return bestPath;

  }//End of findPathFSP() for Best Effort Service

  /**
```

132

```
     * Implementation of the Shortest Widest Path algorithm for Best Effort.
     * The admission procedure for BE requires revision. Currently, a BE flow is
     * admitted if there is more than a minumum amount of path available BW in the BE svc
level.
     * @param srcRterID the IPv6 address of the source router.
     * @param destRterID the IPv6 address of the destination router.
     * @return the required path or null if no path was found.
     */
    private Path findPathSWP(IPv6Address srcRterID, IPv6Address destRterID)
    {
      testMsg("findPathSWP()");

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

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

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

      Path thisPath = null;
      Path bestPath = null;

      int thisAvailableBandwidth = 0;
      int bestAvailableBandwidth = 0;

      Hashtable table1 = new Hashtable();

      for (int i = 1; i < MAX_HOP_COUNT; i++)
      {
        testMsg("Hop count = " + i);
        table1 = aPI[sourceNodeID][destNodeID][i];
        Enumeration enum1 = table1.elements();

        if (enum1.hasMoreElements())
        {
          //Cycle through each of the paths of the current hop count, between
          //source and destination nodes
          while (enum1.hasMoreElements())
          {
            Integer currentPathID = (Integer) enum1.nextElement();

            // Extract current path information
            thisPath = (Path) htPaths.get(currentPathID);

            thisAvailableBandwidth =
              thisPath.getPathServiceLevelQoS(BEST_EFFORT)
                .getAvailableBandwidth();

            if ((!thisPath.bBestEffortTraffic) && (thisPath.bestEffortTrafficCondition !=
Path.RED))
            {
              if (thisAvailableBandwidth > bestAvailableBandwidth)
              {
                bestPath = thisPath;
                bestAvailableBandwidth = thisAvailableBandwidth;
              }
            }


          }//End of while-loop

        }//End of if structure

      }//End of for-loop
```

```
        if (bestPath != null)
        {
        gui.sendText("\t     The selected path is:\t" + bestPath.toString() + "\n" +
                     "\t     Available bandwidth: \t" + bestAvailableBandwidth + " kbps");
        }

        return bestPath;

    }//End of findPathSWP for Best Effort Service

    //[cw]
    /**
     * Implementation of the Shortest Widest Most Disjoint Path algorithm for Best Effort.
     * The admission procedure for BE requires revision. Currently, a BE flow is
     * admitted if there is more than a minumum amount of path available BW in the BE svc
level.
     * @param srcRterID the IPv6 address of the source router.
     * @param destRterID the IPv6 address of the destination router.
     * @return the required path or null if no path was found.
     */
    private Path findPathSWMDP(IPv6Address srcRterID, IPv6Address destRterID, Path
disjointPath)
    {
        testMsg("findPathSWMDP()");

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

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

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

        Path thisPath = null;
        Path bestPath = null;

        int thisAvailableBandwidth = 0;
        int bestAvailableBandwidth = 0;

        int thisIntersection = 999;
        int bestIntersection = 999;

        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);
              thisAvailableBandwidth =
                thisPath.getPathServiceLevelQoS(BEST_EFFORT)
                  .getAvailableBandwidth();

              thisIntersection = 0;
              Enumeration myLinks = thisPath.getInterfaceSequence().elements();
              while (myLinks.hasMoreElements())
              {
```

134

```java
                                                IPv6Address thisLink = (IPv6Address)
(myLinks.nextElement());
              Enumeration linksToAvoid = disjointPath.getInterfaceSequence().elements();
              while (linksToAvoid.hasMoreElements())
              {
                if (thisLink == ((IPv6Address) (linksToAvoid.nextElement())))
                {
                  thisIntersection++;
                }
              }
            }

            if ((!thisPath.bBestEffortTraffic) && (thisPath.bestEffortTrafficCondition !=
Path.RED))
            {
              if (thisIntersection < bestIntersection)
              {
                bestPath = thisPath;
                bestAvailableBandwidth = thisAvailableBandwidth;
                bestIntersection = thisIntersection;
              }
              else if (thisIntersection == bestIntersection)
              {
                if (thisAvailableBandwidth > bestAvailableBandwidth)
                {
                  bestPath = thisPath;
                  bestAvailableBandwidth = thisAvailableBandwidth;
                  bestIntersection = thisIntersection;
                }
              }
            }

          }//End of while-loop

        }//End of if structure

      }//End of for-loop

      if (bestPath != null)
      {
      gui.sendText("\t     The selected path is:\t" + bestPath.toString() + "\n" +
                   "\t     Available bandwidth: \t" + bestAvailableBandwidth + " kbps");
      }

      return bestPath;

    }//End of findPathSWMDP for Best Effort Service

    //[cw]
    /**
     * Implementation of the Shortest Widest LeastCongested Path algorithm for Best
Effort.
     * The admission procedure for BE requires revision. Currently, a BE flow is
     * admitted if there is more than a minumum amount of path available BW in the BE svc
level.
     * @param srcRterID the IPv6 address of the source router.
     * @param destRterID the IPv6 address of the destination router.
     * @return the required path or null if no path was found.
     */
    private Path findPathSWLCP(IPv6Address srcRterID, IPv6Address destRterID)
    {
      testMsg("findPathSWLCP()");

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

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

```java
        int destNodeID = ((InterfaceInfo) htInterfaces
                            .get(destRterID.toString())).getNodeID().intValue();

        Path thisPath = null;
        Path bestPath = null;

        int thisAvailableBandwidth = 0;
        int bestAvailableBandwidth = 0;

        short thisCongestion = 10000; //100%
        short bestCongestion = 10000;

        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);
              thisAvailableBandwidth =
                thisPath.getPathServiceLevelQoS(BEST_EFFORT)
                  .getAvailableBandwidth();

              thisCongestion = thisPath.getPathQoSArray()[BEST_EFFORT].getPacketLossRate();

              if ((!thisPath.bBestEffortTraffic) && (thisPath.bestEffortTrafficCondition !=
Path.RED))
              {
                if (thisCongestion < bestCongestion)
                {
                  bestPath = thisPath;
                  bestAvailableBandwidth = thisAvailableBandwidth;
                  bestCongestion = thisCongestion;
                }
                else if (thisCongestion == bestCongestion)
                {
                  if (thisAvailableBandwidth > bestAvailableBandwidth)
                  {
                    bestPath = thisPath;
                    bestAvailableBandwidth = thisAvailableBandwidth;
                    bestCongestion = thisCongestion;
                  }
                }
              }
            }

          }//End of while-loop

        }//End of if structure

      }//End of for-loop

      if (bestPath != null)
      {
      gui.sendText("\t      The selected path is:\t" + bestPath.toString() + "\n" +
                  "\t      Available bandwidth: \t" + bestAvailableBandwidth + " kbps");
      }

      return bestPath;
```

136

```
    }//End of findPathSWLCP for Best Effort Service

    //[cw]
    /**
     * Updates attributes to reflect new BE traffic.
     * @return  success of operation
     */
    protected boolean initiateBestEffortTraffic()
    {
      bBestEffortTraffic = true;
      bestEffortTrafficCondition = GREEN;
      timeBEinitiated = System.currentTimeMillis();
      timeLastAdvisorySent = 0;
      timeConditionRed = 0;
      return true;
    }

    //[cw]
    /**
     * Updates attributes to reflect termination of BE traffic.
     * @return  success of operation
     */
    protected boolean terminateBestEffortTraffic()
    {
      bBestEffortTraffic = false;
      bestEffortTrafficCondition = GRAY;
      return true;
    }

    //[cw]
    /**
     * Updates attributes to reflect new congestion.
     * @return  success of operation
     */
    protected boolean newCongestion()
    {
      bestEffortTrafficCondition = YELLOW;
      timeLastAdvisorySent = System.currentTimeMillis();
      return true;
    }

    //[cw]
    /**
     * Updates attributes to reflect congestion cleared.
     * @return  success of operation
     */
    protected boolean congestionCleared()
    {
      bestEffortTrafficCondition = GREEN;
      return true;
    }

    //[cw]
    /**
     * Updates attributes to reflect being expired for BE traffic.
     * @return  success of operation
     */
    protected boolean expireBEpath()
    {
                      if ((bestEffortTrafficCondition == GREEN) ||
(bestEffortTrafficCondition == YELLOW))
                      {
              bestEffortTrafficCondition = RED;
              bBestEffortTraffic = false;
              timeConditionRed = System.currentTimeMillis();
              return true;
                      }
                      else
```

137

```
                        {
                                return false;
                        }
}

//[cw]
/**
 * Updates attributes to reflect being unexpired for BE traffic.
 * @return   success of operation
 */
protected boolean unexpireBEpath()
{
  if ((bestEffortTrafficCondition == RED) && bConnected)
  {
    bestEffortTrafficCondition = GREEN;
    return true;
  }
  else
  {
    return false;
  }
}
```

APPENDIX H: MODIFICATIONS TO OTHER SAAM SOURCE CODE

```
        // create IPV6 packet
                                        //[cw]
        // For BE traffic, assign random source addresses to simulate internet traffic.
        // Note: when a packet leaves Flow Generator with an all-zero source address,
        // it will be assigned the sending interface's address.  The code below allows
        // the Flow Generator to simulate traffic coming in from outside since it gives
        // it a non-zero source address.  For BE testing, THIS IS THE INTENT.
                    if (typeOfService.equals("BestEffort"))
                    {
            int randNum = (int) ((9 - 0) * (Math.random()) + 0);
                    try
                    {

        ipv6Header.setSource(IPv6Address.getByName("1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1." +

                                                            (new
Integer(randNum)).toString())));
                    }
                    catch (UnknownHostException uhe)
                    {
            gui.sendText("Error setting random Best Effort header.");
                    }
                    }

  /**
   * Forwards the packet that was just dequeued from a service level
   * queue to the outbound NetworkInterfaceCard.
   * @param sl The service level this packet was dequeued from.
   * @param packet the byte array representation of this packet.
   */
  private void forwardPacket(int sl, byte [] packet)
  {
    //Since the nextHop was added to the packet before
    //the packet was enqueued into the Service Level Queue,
    //we strip it off here.

    //Xie-dec01:
    // Need to scale packet transmission time based on time scale, packet length (bits),
    // and link speed (bits/second); insert a 200 ms constant delay temporarily
    // May require a timer to determine the end of transmission for this packet.
                          //[cw] 1000 is for s-to-ms, 8 is for B-to-b

                // bits/Kpbs = milliseconds
                timeElapsed = System.currentTimeMillis() +
                          ((packet.length * 8) / linkSpeed) * timeScale;

    IPv6Address nextHop = null;
    try
    {
      nextHop = new IPv6Address(Array.getSubArray(packet, 0, IPv6Address.length));
    }
    catch (UnknownHostException uhe)
    {
      gui.sendText(toString() + ": " + uhe.toString());
    }
    gui.sendText("  Next Hop: " + nextHop.toString());
    IPv6Packet v6Packet = null;
    try
    {
      v6Packet = new IPv6Packet(packet);
    }
    catch (UnknownHostException uhe)
    {
```

```
    gui.sendText("Scheduler: " + uhe.toString());
  }
  gui.sendText("  Forwarding packet to my NIC; Payload length = " +
    v6Packet.getPayload().length);

  ProtocolStackEvent event = new ProtocolStackEvent(
      toString(),
      this,
      outBoundChannel,
      packet,
      sl,
      nextHop);

  try
  {
    controlExec.talk(event);
  }
  catch (ChannelException tde)
  {
    gui.sendText(tde.toString());
  }


            //[cw] corey's proposed solution
            long now = System.currentTimeMillis();
            if (now < timeElapsed)
            {
                    try
                    {
                            Thread.sleep(timeElapsed - now);
                    }
                    catch (InterruptedException ie)
                    {
                            gui.sendText("Thread sleep problem in module " +
this.toString());
                    }
            }

  }//end forwardPacket()
```

LIST OF REFERENCES

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

[2] Elwalid, A., C. Jin, S. Low, and I. Widjaja, "MATE: MPLS Adaptive Traffic Engineering", IEEE INFOCOM 2001.

[3] Feit, Sidnie, TCP/IP: Architecture, Protocols, and Implementation with IPv6 and IP Security, McGraw-Hill, 1999.

[4] Floyd, S., and K. Fall, "Promoting the Use of End-to-end Congestion Control in the Internet", IEEE/ACM Transactions on Networking, August 1999.

[5] Gibson, John H. and Dao-Cheng, Kuo, "Design of a dynamic management capability for the Server and Agent Based Active Network Management (SAAM) system to support requests for guaranteed Quality of Service traffic routing and recovery", Computer Science Department, Naval Postgraduate School, Monterey, September 2000.

[6] Internet2, www.internet2.edu.

[7] Isenberg, David, "Rise of the Stupid Network", www.isen.com.

[8] Quek, Henry C., "QoS management with adaptive routing for next generation Internet", Computer Science Department, Naval Postgraduate School, Monterey, March 2000.

[9] Silva, Paulo, "Advanced Quality of Service Management for Next Generation Internet", Computer Science Department, Naval Postgraduate School, Monterey, Septemeber 2001.

[10] Thompson, K., G.J. Miller, and R. Wilder, "Wide-area Internet Traffic Patterns and Characteristics", IEEE Network, November 1997.

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

[12] Vrable, Dean J. and Yarger, John W., "The SAAM architecture: enabling integrated services", Computer Science Department, Naval Postgraduate School, Monterey, September 1999.

[13] Wang, Z., and J. Crowcroft, "QoS Routing for Supporting Resource Reservation", IEEE Journal on Selected Areas in Communication, September 1996.

[14] Widmer, J., R. Denda, and M. Mauve, "A Survey on TCP-friendly Congestion Control", IEEE Network, May 2001.

[15] Wright, Troy, "Fault Tolerance in the Server and Agent Based Network Management (SAAM) System", Computer Science Department, Naval Postgraduate School, Monterey, September 2001.

[16] Xiao, Xipeng, A. Hannan, B. Bailey, and M. Li, "Traffic Engineering with MPLS in the Internet", IEEE Network, March 2000.

[17] Yang, Y., K. Muppala, and S.T. Chanson, "Quality of Service Routing Algorithms for Bandwidth-Delay Constrained Applications", ICNP 2001.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Fort Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. LCDR Chris Eagle
   Naval Postgraduate School
   Monterey, California

4. Professor Geoffrey Xie
   Naval Postgraduate School
   Monterey, California

5. Professor James Bret Michael
   Naval Postgraduate School
   Monterey, California

6. Cary Colwell
   Naval Postgraduate School
   Monterey, California

7. LT Corey Wofford
   Naval Postgraduate School
   Monterey, California