FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Shafiq Pirbhai
AUTEUR DE LA THÈSE / AUTHOR OF THESIS

Master of Computer Science
GRADE / DEGREE

School of Information Technology and Engineering
FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

Using Mobile Agents and Maximum Path Flow Algorithms to Maximize Network Utilization

TITRE DE LA THÈSE / TITLE OF THESIS

George White
DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

Ahmed Karmouch

Michael Weiss

Gary W. Slater
LE DOYEN DE LA FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES /
DEAN OF THE FACULTY OF GRADUATE AND POSTDOCORAL STUDIES

Using Mobile Agents and Maximum Path Flow Algorithms to Maximize Network Utilization

Supervised by Dr. George White

Shafiq Pirbhai

Student Number: 1434255

Candidate for Masters Degree in Computer Science

School of Information Technology and Engineering
University of Ottawa

February 24th, 2006

Your file Votre référence
ISBN: 0-494-14938-8
Our file Notre référence
ISBN: 0-494-14938-8

To my wife Zahra

Acknowledgement

I would like to express my appreciation to Dr. George White, without his supervision, encouragement and patience I could not complete this research and thesis writing.

I would also like to thank the examiners who have offered valuable feedback before and after the final presentation of this thesis.

I would like to thank my wife for her support and encouragement. She has been patient with me while I worked evenings and weekends completing this thesis.

Finally, I would like to thank my Managers and Directors at Alcatel for giving me the flexibility and support to complete this thesis.

ABSTRACT

The purpose of this thesis is to show how the use of one or more of the shortest augmenting path (polynomial time) algorithms are used to maximize the bandwidth usage and minimize the cost and latency in Multiprotocol Label Switched (MPLS) networks. The thesis will also show how providers that offer services such as Virtual Private Networks (VPNs) based on Border Gateway Protocol (BGP) and MPLS, or Virtual Private LAN Services (VPLS) can use such an algorithm to offer premium services to higher paying customers.

This thesis extends the problem of adaptive routing in communications networks:

It focuses on maximizing the bandwidth usage in a MPLS network by using a modified augmenting path algorithm. The algorithm is influenced by *Resource Reservation Protocol (RSVP)* **[20]** to reserve traffic flows on routers. The algorithm proposed is inspired by previous research in the area of indirect communication among agents through modifications induced in their environment or simply known as *stigmergy* **[15]**. The agents implement the Edmonds Karp Algorithm rather than the pheromone tables in *stigmergy*. By using a mobile agents approach, the Augmenting Path Algorithm is adapted to find the maximum flow through a network.

GLOSSARY

AA	*Active Applications.* Code that can be dynamically loaded and run on top of an execution environment (EE).
ATM	*Asynchronous Transfer Mode*: A network technology based on transferring data in cells or packets of a fixed size.
BGP	*Border Gateway Protocol.* An inter-Autonomous System routing protocol defined by the IETF.
EE	*Execution Environment.* Exports a programming interface or virtual machine that can be programmed or controlled by an active application (AA).
IETF	*Internet Engineering Task Force.* The IETF is a large, open international community of network designers, operators, vendors and researchers whose purpose is to coordinate the operation, management and evolution of the Internet and to resolve short- and mid-range protocol and architectural issues.
IS-IS	*Intermediate System to Intermediate System Protocol.* The ISO standard intra-domain routing protocol documented in ISO 10589.
IP	*Internet Protocol.* The Internet Protocol is the network layer for the TCP/IP Protocol Suite. It is a connectionless, best-effort packet switching protocol defined by the IETF.
LER	*Label Edge Router.* A term used to indicate an LSR that is able to provide ingress to and egress from an LSP. In individual implementations, this tends to be a function of the capabilities of device interfaces more than of the overall device. In theory, it is possible for a device to be an LER and not an LSR (if it is not able to swap labels, for instance).
LSR	*Label Switched Router.* A device that participates in one or more routing protocols and uses the route information derived from routing protocol exchanges to drive LSP setup and maintenance. Such a device typically distributes labels to peers and uses these labels (when provided as part of data presented for forwarding) to forward label-encapsulated packets by swapping labels.
LSP	*Label Switched Path.* The specific path through a network that a datagram follows, based on its MPLS labels

MPLS *Multi-Protocol Label Switching.* MPLS is a method for speeding up the network traffic flow of IP packets. It involves setting up a specific path for a given sequence of packets, identified by a label put in each packet, thus saving the time needed for a router to look up the address to the next node to forward the packet to

OSPF *Open Shortest Path First.* A link-state routing protocol defined by the IETF, that is designed to be run internal to a single Autonomous System.

PE *Provider Edge Node.* A provider edge node is the demarcation point between the customer and the service provider networks. The PE node may be located at the customer premise (for example in the basement or wiring closet) or at the provider premise (for example at the Central Office or the Point of Presence) and is owned and operated by the service provider.

PNNI *Private Network-Network Interface.* A dynamic source routing protocol for ATM networks that provides routing between ATM switches. It is also known as Private Node-Network Interface.

QoS *Quality of Service.* QoS provides the ability to distinguish between different types of traffic for the purpose of resource allocation. It uses bandwidth, latency, jitter, and packet loss as its metrics.

RFC *Request For Comments.* Internet informational documents and standards widely followed by commercial software and freeware in the Internet and Unix communities. Few RFCs are standards but all Internet standards are recorded in RFCs.

RSVP The *Resource Reservation Protocol.* A network-control protocol that enables Internet applications to obtain differing qualities of service (QoS) for their data flows.

ToS *Type of Service.* Type of Service bits is a set of bits in the IP header that enables the application transmitting the data to tell the network what type of service it requires.

VPLS *Virtual Private LAN Service.* A VPN service based on Ethernet protocol used to seamlessly integrate carrier and customer networks.

VPN *Virtual Private Network.* A service that allows connectivity of various private customer networks through a carrier's network.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1 Resource utilization problems in IP and MPLS networks.

A provider that offers IP connectivity using a common MPLS backbone network sells services to hundreds of customers. Each customer at any given time has a known bandwidth requirement. Each physical link in the provider's backbone has a fixed bandwidth. Providing bandwidth over some links may be more expensive than over others (in the case where a provider leases bandwidth from another provider), or uses a more expensive media to provide connectivity. There are certain links that charge based on the minute (even if there is no traffic on this link at a given time) or by the mile, as opposed to other links that charge by the packet. As well, each link may have a different latency and reliability.

For each customer that requires connectivity between a pair of sites *(i, j)* a unit cost c_{ij} is incurred. This cost depends on the medium used to provide the service and the distance between the sites. As well, traffic that travels between these two sites incurs a latency l_{ij} and has a certain reliability r_{ij}. The latency is determined by the media used to provide the service, the number of hops between the sites, and the amount of time it takes for the data to be processed at each node along the path. The problem is to arrange a minimum cost bandwidth usage pattern; that is, to provide service by maximizing the usage of the resources in the network while trying to keep costs as low as possible with no latency and reliability guarantees for lower paying customers. A higher paying customer is given a service with smaller latency and higher reliability. Maximum flow and minimum cost

problem algorithms are applied to MPLS networks to allow providers to offer higher paying customers guaranteed bandwidth, low latency and high reliability services.

Conventional IP systems use Interior Gateway Protocols (IGPs) such as Intermediate System–Intermediate System (IS-IS) and Open Shortest Path First (OSPF). Exterior Gateway Protocols (EGPs) such as Border Gateway Protocol (BGP), and Asynchronous Transfer Mode (ATM) systems use Private Network-Network Interface (PNNI) to make independent routing decisions using a local instantiation of a synchronized routing area link state database. Route selection is based on shortest path computations (Dijkstra's shortest path algorithm) using simple additive link metrics. The following statement extracted from the Internet Engineering Task Force (IETF) specification for Routing Information Protocol (RIP) [48] discusses the weakness of the use of a 'metric':

"This protocol uses fixed *metrics* to compare alternative routes. It is not appropriate for situations where routes need to be chosen based on real-time parameters such a measured delay, reliability, or load. The obvious extensions to allow metrics of this type are likely to introduce instabilities of a sort that the protocol is not designed to handle".

This is reinforced by [47] which states that common IP routing protocols that use such *metrics* are "highly distributed and scalable, but flawed". The flaw is that these routing protocols only consider the shortest path but do not consider the characteristics of the data traffic and network link latency and capacity constraints when making routing decisions. This results in some links of network resources becoming congested, while other links along alternate paths remain underutilized. This also leads to the underutilization of network resources. As well, data originating from sites of higher

paying customers is not treated any differently from data originating from other customer sites. After a thorough search through existing literature I was only able to find a message based implementation for rerouting of MPLS paths [53]. I was unable to find literature that describes methods to maximize bandwidth usage in an MPLS network using mobile agents.

1.1.1 MPLS and RSVP

The growing number of computer users on the Internet and intranets, as well as new bandwidth intensive applications such as those incorporating voice and video, are driving the need for guaranteed bandwidth and increased network reliability. The typical frame and packet-based networks lack the quality of service (QoS) and traffic shaping sophistication of the powerful yet expensive ATM networks. Furthermore, the proliferation of network protocols increases the complexity and reduces network capability and performance. In an effort to increase throughput, reduce network complexity in ATM networks, and bring advanced bandwidth shaping and QoS capabilities to non-ATM networks, the Internet Engineering Task Force (IETF) created Multi Protocol Label Switching (MPLS). MPLS combines the power of layer 2 switching with the flexibility and intelligence of layer 3 protocols; it operates independently of other network technologies but is fully capable of interoperating with them. MPLS brings non-ATM networks powerful QoS capabilities, the ability to route multiple network technologies (Ethernet, frame relay, ATM) over one infrastructure and the capability of interoperating with modern routing protocols such as IS-IS, OSPF and BGP, while increasing efficiency and simplifying network infrastructure.

Although MPLS is a relatively simple technology (based on the classical label swapping paradigm), it enables the introduction of sophisticated control capabilities that advance the traffic engineering function in IP networks. A particularly interesting aspect of MPLS is that it efficiently supports origination connection control through explicit label switched paths.

Resource Reservation Protocol (RSVP) **[20]** was developed in the mid-1990s to combat network congestion by allowing routers to decide in advance whether they could meet the requirements of an application flow. RSVP is a network-control protocol that enables Internet applications to obtain differing qualities of service (QoS) for their data flows. Such a capability recognizes that different applications have different network performance requirements. One implementation of MPLS is RSVP with a signaling protocol extension. The extension supports label distribution, resource reservation and quality of service to create MPLS RSVP Label Switched Paths (LSPs).

1.1.2 Mobile Agents

The management of MPLS backbone networks is becoming an increasingly complicated task, as the number of customers and their demand for bandwidth increases. It has become evident that centralized management solutions cannot cope with the scalability issues [33]. Mobile agent technology is a new distributed system and network paradigm that can be used to manage the utilization of network resources.

Mobile Agents describes the concept of mobile computing or mobile code [1],[2]. The mobile agent paradigm has attracted attention from many fields of Computer Science. The use of mobile agents is quite appealing – mobile agents roam networks searching for

information, and interacting with other agents that roam the network or are bound to a particular machine. Agents are being used or proposed for a large number of applications that range from small systems to large networks, from simple non real time applications to complicated real time ones.

Danny Lange, who led the IBM Aglets development team lists the following seven reasons for using mobile agents [50]:

1. They can reduce network load, because agents move to a system and do their work rather than take up network bandwidth sending messages back and forth.
2. They can overcome network latency because they are resident on the machine rather than remote.
3. They can encapsulate protocols as they move around the network talking to other mobile agents.
4. They can operate autonomously so they keep working even when network connections go down.
5. They can dynamically adapt to changes in system loading.
6. They are heterogeneous.
7. They are fault-tolerant because they can move from a system that is having difficulty or about to fail.

As well, mobile agents allow service providers to introduce new versions of the agent without affecting the network. This allows quicker and easier introduction of new

services when customers demand it. Agents may need to be replaced due to 'bug' fixes or because a new algorithm is needed (for example, to use a different search criteria). Mobile Agents reduce the need to upgrade every node in the network when a new version is implemented.

When networking protocols are upgraded, parts of the network need to be taken offline. A recent article in Network World states "A conservative estimate from Gartner pegs the hourly cost of downtime for computer networks at $42,000, so a company that suffers from worse than average downtime of 175 hours a year can lose more than $7 million per year."

1.2 Goals

This thesis shows how the Edmonds Karp shortest augmenting path algorithm and the use of mobile agents improve the bandwidth usage in MPLS networks. Unlike existing protocols that rely on a cost metric, mobile agents are used to select network links based on one or a combination of criteria such as

i. Minimum link cost,

ii. Maximum available link bandwidth,

iii. Minimum link latency

iv. Maximum Reliability

Mobile agents allow the flexibility to set up best effort services for lower paying customers by selecting links with minimum available bandwidth, maximum link latency and minimum reliability.

1.3 Contributions

The contributions of this thesis are the improvement in the selection of MPLS network links. The Agent based Edmonds Karp RSVP (A-EK-RSVP) algorithm maximizes bandwidth usage in an MPLS network by using the Edmonds Karp augmenting path algorithm.

1.3.1 Flexible Link Selection Criteria

Simulations (Section 8) show the flexibility of A-EK-RSVP algorithm since it allows links to be selected based on different criteria (Section 5.1). This is an improvement over the RSVP algorithm that uses a single routing protocol metric to select network links.

1.3.2 Bumping of lower priority LSPs

The A-EK-RSVP algorithm allows bumping of LSPs in the Network based on LSP priorities (Section 5.2). The result is a more improved use of network resources based on the amount paid by the customer for the service.

1.3.3 Maximizing Network Resources

The A-EK-RSVP algorithm implements the Edmonds Karp augmenting path algorithm. Maximum augmenting path algorithms are used to maximize flows that can be sent through a network. Simulations (Section 8) show that number of LSPs created using the A-RK-RSVP algorithm is greater than when using the metric based RSVP algorithm.

1.4 Thesis Overview

The thesis is divided into the following broad areas: introduction (Section 1), background and previous work (Section 2, 3 and 4), proposed algorithm (Section 5), and examples and results (Sections 7 and 8). This section concludes with the introductory material. The following sections discuss existing research in the field of Maximum Flow algorithms, and existing MPLS and RSVP standards.

The following sections discuss the proposed algorithms, examples and simulation results. A summary of conclusions and suggestions for future areas of research concludes the thesis.

2. BACKGROUND ON MAXIMUM FLOW PROBLEMS

2.1 Flow Terminology

Before proceeding into the details of the algorithm, it would be helpful to define a number of concepts in flow terminology and constraints. This section explains the various terms used when describing flow problems.

2.1.1 Flow Network

A **flow network** $G = (V, E)$ is a directed connected graph where each edge $(u,v) \in E$ (between node $u \in V$ and node $v \in V$) has capacity $c(u,v) \geq 0$. Then $|E| \geq |V| - 1$.

The source node $(s \in V)$ and sink node $(t \in V)$ are the vertices between which the flow is to be maximized.

2.1.2 Flow

A **flow** f is a real-valued function representing the rate of flow between u and v, on arc (u,v), with capacity $c(u,v)$, is denoted by $f(u,v)$. The flow is constrained by the following:

$$\text{Capacity Constraint}: \quad 0 \leq f(u,v) \leq c(u,v) \tag{2.4}$$

$$\text{Skew Symmetry}: \quad f(u,v) = -f(v,u) \tag{2.5}$$

$$\text{Flow Conservation}: \quad \forall u \in V \setminus \{s,t\}, \sum_{v} f(u,v) = 0 \tag{2.6}$$

2.1.3 Residual Capacity

Given flow network $G = (V, E)$ with flow f between any two vertices $u, v \in V$, define the **residual capacity** $c_f(u,v)$ as the additional net flow possible from u to v, not exceeding $c(u,v)$. Thus

$$c_f(u,v) = c(u,v) - f(u,v)$$

For example, if capacity of a link $(u,v) = 10$ units, and the flow on that link $(u,v) = 6$ units, then the residual capacity of the link $(u,v) = 4$ units.

2.1.4 Residual Network

Given flow network $G = (V, E)$ and flow f, the **residual network** of G induced by f is $G_f = (V, E_f)$ where

$$E_f = \{(u,v) \in VxV \mid c_f(u,v) > 0\}$$

$(u,v) \in E_f$ is a residual edge; i.e., any edge that can admit more flow. If there is such a path from s to t in the residual network, then it is said to be an augmenting path and indicates where flow can increase.

2.2 Maximum Flow Problem

We consider a network flow problem called the *Generalized Maximum Flow problem.* First, we describe the *maximum flow problem.* This problem was first studied by Dantzig **[6]** and Ford and Fulkerson **[9]** in the 1950's. The problem statement is defined as follows:

Given capacity limits on the arcs, the problem is to send as much flow as possible from one distinguished node called the source to another called the sink.

2.2.1 Maximum Integral flow problem

Suppose that we want to send data from a source node s of the network to a destination node t. The restriction is that, for each link (u,v) in the network, there is an upper bound $c(u,v)$ in the capacity (also known as bandwidth).

If we formulate this problem on a corresponding digraph G, it is one of finding a family $(P_1, \ldots\ldots, P_k)$ of (not necessarily distinct) *(s,t)-dipaths* such that each link (u,v) is a link of at most $c(u,v)$ of the dipaths and such that k is maximized. The reason why they are not necessarily distinct is that a dipath P_i that originates from s and terminates at t, may have an identical route through the network as another dipath P_j. As well, from our definition that $c(u,v)$ is the upper bound of the capacity of each link (u,v), it follows that at most $c(u,v)$ of dipaths that consume 1 unit of capacity can traverse the link (u,v).

Notice for each node $u \neq s,t$, any P_i must enter and leave v the same number of times. Let f_{uv} denote a flow on edge e which is part of dipath P_i. Therefore, flow f satisfies:

$$\sum(f_{uv} : u \in V, uv \in E) - \sum(f_{vu} : v \in V, vu \in E) = 0, \text{ for all } u \in V \setminus \{s,t\} \qquad (2.1)$$

$$0 \leq f_{uv} \leq c_{uv}, \text{ for all } uv \in E \qquad (2.2)$$

$$f_{uv} \text{ integer, for all } uv \in E \qquad (2.3)$$

Let us call a vector f an *(s,t) flow*, or just a flow if it satisfies (2.1), and a feasible flow if it satisfies (2.2). The left-hand side of (2.1) is the net flow of u, and we **abbreviate** it to $f(u)$. The condition $f(u) = 0$ requires conservation of flow at u. That is, the total flow into the node minus the total flow out of the node is equal to zero. Conservation of flow is not required at the source and sink of the flow (s and t respectively).

Figure 1 shows an example of flows that satisfy the limitations given by (2.1), (2.2) and (2.3) above. The figure shows 3 flows, *{s,a,b,t}* of 1 unit, *{s,c,d,t}* of 1 unit, and *{s,c,b,a,d,t}* of 1 unit.

At node a there are 2 units of flow entering and 2 units of flow leaving. At node b there are 2 units of flow entering and there are 2 units of flow leaving. At node c there are 2 units of flow entering and 2 units of flow leaving. And finally at node d there are 2 units of flow entering and 2 units of flow leaving. Therefore for every node $u \neq s,t$ the total value of flows entering and leaving u are equal. Each of the edges (u,v) in the diagram has an integer flow $f(u,v)$ that is less than or equal to the capacity $c(u,v)$. Therefore this is a feasible flow.

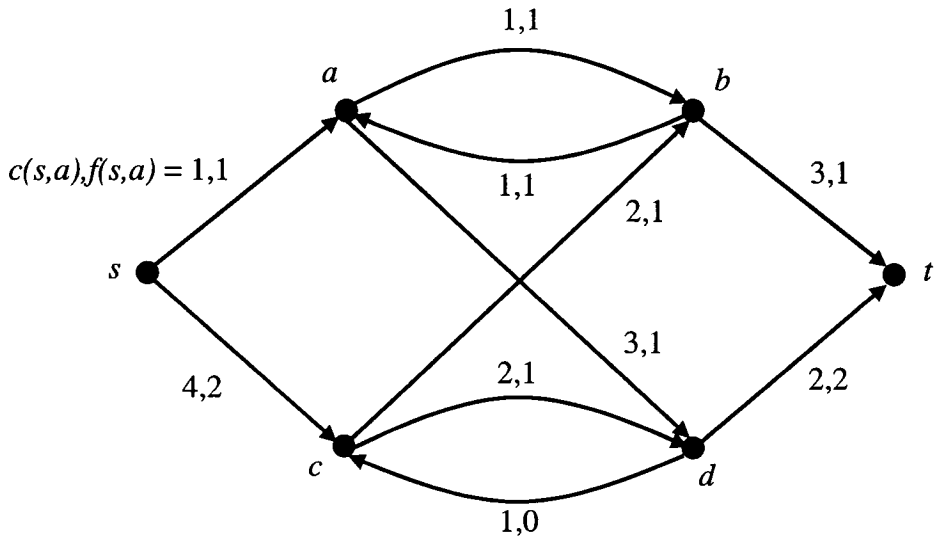

Figure 1. Feasible flow digraph

The Maximum Integral Flow Problem is formulated as follows:

$$\text{Maximize } f(s,t) \text{ subject to}$$

$$f(u) = 0, \text{ for all } u \in V \setminus \{s,t\}$$

$$0 \leq f(u,v) \leq c(u,v), \text{ for all } (u,v) \in E$$

$$f(u,v), \text{ integral, for all } (u,v) \in E$$

The maximum flow problem is the same problem without the restriction of integrality. We shall see that the integrality restriction causes no additional difficulties. The quantities *c(u,v)* are called capacities. We allow them to be non-negative real numbers or ∞. The latter means that there is no explicit upper bound on *f(u,v)*.

2.2.2 Maximum Flows and Minimum Cuts

A concept from graph theory that is useful for modeling the carrying capacity of a network is the *cut*. A *u-v cut* is a set of arcs whose removal disconnects node *u* from node *v*. A minimal cut is one such that the replacement of any of its member arcs reconnects the graph. In other words, in a minimal cut all arcs are essential. In a weighted graph, each cut has a capacity. The capacity of a cut is the sum of the weights of the arcs in the cut. A cut with the minimum capacity is called a *Minimum Cut*.

The *Maximum Flow* between any two arbitrary nodes in any graph cannot exceed the capacity of the *Minimum Cut* separating those two nodes [9].

We call a set $\delta(R) = \{u,v : uv \in E, u \in R, v \notin R\}$ for some $R \subseteq V$, a *cut*. An *(s,t)-cut* is a cut for which $s \in R, t \notin R$. For any $A \subseteq V$ we use $\overline{A}$ to denote $V \setminus A$. Finally, for $u \in V$, we use $\delta(u)$ as an abbreviation for $\delta(\{u\})$, and $\delta(\overline{u})$ as an abbreviation for $\delta(\overline{\{u\}})$.

For any *(s,t)-cut* $\delta(R)$ and any *(s,t)-flow f*, we have:

$$f(\delta(R)) - f(\delta(\overline{R})) = f(s)$$

For example in Figure 2 if R = *{s, a, c)* then we have:

$f(\delta(R)) - f(\delta(\overline{R}))$ = *(f(a,b)+f(a,d)+f(c,b)+f(c,d))* - *(f(b,a))* = 4 – 1; and

$f(s)$ = *(f(s,a)+f(a,c))* = 3.

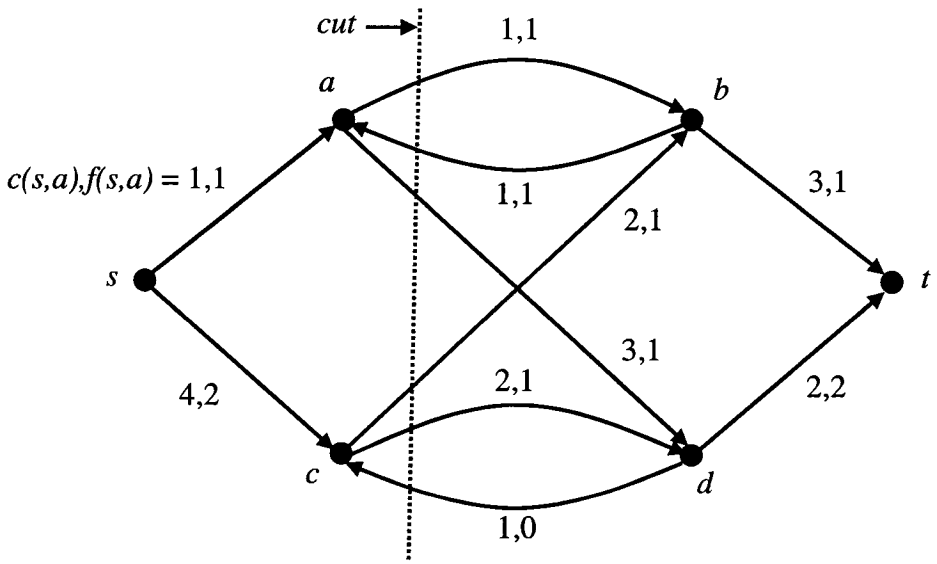

Figure 2. Cut

For any feasible *(s,t)-flow f* and any *(s,t)-cut* $\delta(R)$, we have:

$$f(s) \leq c(\delta(R))$$

Max-Flow Min-Cut Theorem: If there is a maximum *(s,t)-flow*, then:

$$\max\{ f(s) : f \text{ a feasible } (s,t)\text{-flow}\} = \min\{ c(\delta(R)) : \delta(R) \text{ an } (s,t)\text{-cut}\}$$

2.2.3 The Augmenting Path Algorithm

An example of an Augmenting Path algorithm is the maximum flow algorithm of Ford and Fulkerson. Beginning with any feasible *flow f*, repeatedly find an *f*-augmenting path *P* and augment *x* by the maximum amount permitted. Of course, this amount is $min(\varepsilon_1, \varepsilon_2)$, where $\varepsilon_1 = min\ (c_e - f_e : e \text{ forward in } P)$ and where $\varepsilon_2 = min\ (f_e : e \text{ reverse in } P)$. We call ε the *f*-width (or flow-width) of *P*.

2.2.4 Shortest Augmenting Path Algorithm

We call an x-augmenting path shortest if it has the minimum possible number of arcs. By restricting attention to the shortest augmenting paths we can improve the running time of the maximum flow algorithm dramatically [10]. It is no harder to find the shortest augmenting path than to find an augmenting path. For this reason Edmonds and Karp [10] called their modification "so simple that it is likely to be incorporated innocently into a computer implementation".

One such algorithm is Busacker-Gowen **[11]** (there are better algorithms but we'll pick this one as it is easily illustrated).

1. Start with a zero flow
2. Repeat the following (until no more flow augmenting paths can be found)
    a. Using the feasible flow digraph, add *forward arcs* for arcs where the capacity is still not maximized, and *reverse arcs* for arcs that have positive flow (also known as *backward arcs*). This new digraph is known as an *auxiliary* digraph. Figure 3 shows the auxiliary digraph for the feasible flow digraph in Figure 1. The *forward arcs* are shown as solid lines while the *reverse arcs* are shown as dotted lines.
    b. Find the cheapest *s-t* dipath (Use Belman Ford)
    c. Use the corresponding flow augmenting path to augment the flow.

Result: A minimum cost maximum flow when algorithm is run to completion.

Several such shortest augmenting path algorithms with different complexities have been published (refer to Section 2.5).

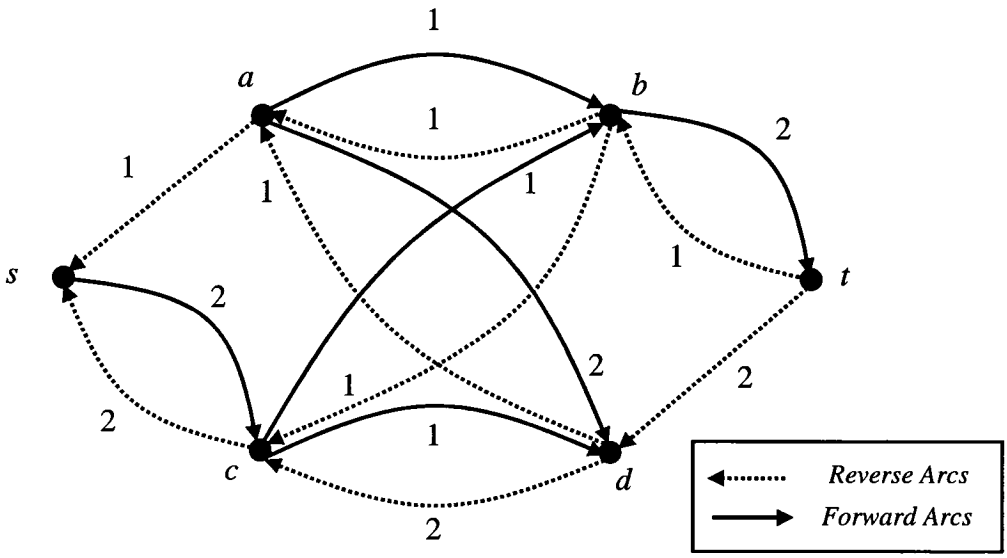

Figure 3. Auxiliary digraph

2.3 Shortest Path Problem

The *Shortest Path Problem* (also known as minimum cost path) can be described as:

Find a simple path between two nodes, so as to minimize the total length (or cost).

An instance of the shortest path problem is a network $G = (V, E, s, l)$, where $s \in V$ is a distinguished node called the *source*, and l is a length (or cost) function. The problem is NP-hard if negative length cycles are allowed [44]. In networks with no negative length cycles, there are a number of polynomial time algorithms for the problem, e.g. Bellman-Ford [54]. There are faster specialized algorithms for networks with nonnegative arc lengths, e.g., Dijkstra [43].

We will discuss these algorithms and show how they are used with max flow problem algorithms to maximize network utilization.

2.4 Ford-Fulkerson Max flow Labeling Algorithm

The Ford-Fulkerson max flow labeling algorithm [7],[8],[9] was introduced in the mid-1950's. Given flow network G and flow f, an augmenting path p from s to t in the residual network G_f, then the minimum net flow along path p in G_f indicates the amount flow can increase along this path in G.

Thus, the **residual capacity of path *p*** is

$$c_f(p) = \min\{c_f(u,v) \mid (u,v) \text{ is along } p\}$$

Define flow f_p in G_f as

$$f_p(u,v) = \begin{cases} c_f(p) & \text{if } (u,v) \text{ on } p \\ -c_f(p) & \text{if } (v,u) \text{ on } p \\ 0 & \text{otherwise} \end{cases}$$

$$|f_p| = c_f(p) > 0$$

Define flow sum $f_1 + f_2$ as $(f_1 + f_2)(u,v) = f_1(u,v) + f_2(u,v)$

If flow network G has flow f, augmenting path p in G_f, and $f'=f+f_p$, *then* f *is a flow in* G *with value* $|f'|=|f|+|f_p|>|f|$

The above steps are repeated until there are no more augmenting paths.

f is a maximum flow in G if and only if residual network G_f has no augmenting paths.

2.4.1 Ford-Fulkerson Routines

We are given a network with two specified nodes: the source s and sink t. An arc can be identified by its endpoints: (u,v) is the arc from node u to node v. Recall (from Section 2.2.1) that a flow across arc (u,v) is denoted by $f(u,v)$, and the arc's capacity is $c(u,v)$. The flow must satisfy:

$$0 \leq f(u,v) \leq c(u,v)$$

The algorithm has two parts, which Ford and Fulkerson called *Routine* A and *Routine* B. The first is a labeling process that searches for a *flow augmenting path* - i.e., a path from s to t for which $f < c$ along all *forward arcs* (u,v) and $f > 0$ along all *backward arcs* (v,u).

If *Routine* A finds a flow augmenting path, *Routine* B changes the flow accordingly. Otherwise, no augmenting path exists, and optimality of the current flow is ensured by the theorem [7]:

A flow f has maximum value if, and only if, there is no flow augmenting path with respect to f.

We begin with any feasible flow (e.g., f=0). In general, a node is in one of three states: ***"unlabeled"***, or ***"labeled and scanned"***, or ***"labeled and unscanned"***. Upon entering *Routine* A, all nodes are unlabeled. The first step renders the source ***labeled and unscanned***.

Routine A (labeling process). Initially, label the source $(-, \varepsilon(s) = \infty)$

General step: Select any node, u, that is ***labeled and unscanned***, and let $(u^{\pm}, \varepsilon(u))$ be its label. To all unlabeled successor nodes, v, such that $f(u,v) < c(u,v)$, assign the label $\left(u^{+}, \varepsilon(v)\right)$, where

$$\varepsilon(v) = \min\{\varepsilon(u), c(u,v) - f(u,v)\}$$

(Such v are now ***labeled and unscanned***). To all predecessor nodes, v, that are ***unlabeled***, such that $f(v,u) > 0$, assign the label $\left(u^{-}, \varepsilon(v)\right)$, where

$$\varepsilon(v) = \min\{\varepsilon(u), f(v,u)\}$$

(Such v are now ***labeled and unscanned***). Now define u to be ***labeled and scanned***. Repeat the general step until the sink is ***labeled and unscanned***, or until no more labels can be assigned. In the former case, go to *Routine* B; in the latter case, terminate (f is a maximum flow).

Routine B (flow change). The sink has been labeled $(v^{\pm}, \varepsilon(t))$.

If the first part of the label is v^{+}, replace $f(v,t)$ with $f(v,t)+\varepsilon(t)$; otherwise, replace $f(t,v)$ with $f(v,t)-\varepsilon(t)$. Go to node v and treat it the same way: if its label is $(u^{+}, \varepsilon(t))$, replace $f(u,v)$ with $f(u,t)+\varepsilon(t)$; if its label is $(u^{-}, \varepsilon(v))$, replace $f(v,u)$ with $f(v,u)-\varepsilon(t)$. In either case, go to node u and repeat until the source node s is reached. Then, discard all labels and return to *Routine* A.

2.4.2 Example of Ford-Fulkerson

The figures below illustrate how the Ford-Fulkerson algorithm is used to solve a maximum flow problem. Each edge of the network is marked with a capacity (the first value), and the flow (second value). The source is indicated as s and the sink as t.

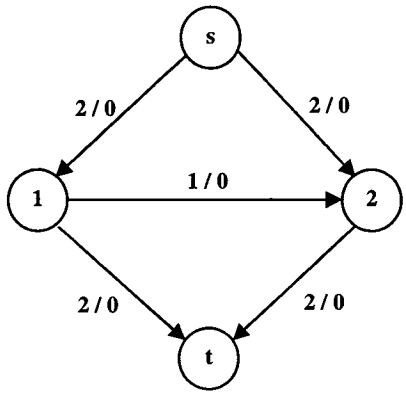

Figure 4. Initial graph

A flow (traced with a dashed line below) of 1 unit is sent from node *s*, through nodes 1 and 2, to node *t*. The flow along the edges in incremented by 1 unit.

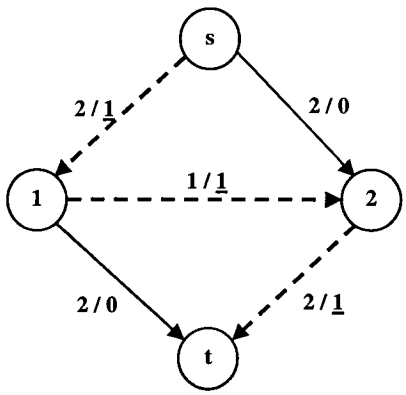

Figure 5. Flow of 1 unit

Another flow (traced with a dashed line) of 1 unit is sent from node *s*, through node 1, to node *t*. The flow along the edges in incremented by 1 unit.

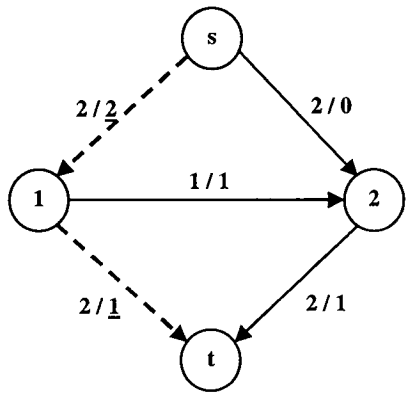

Figure 6. Another flow of 1 unit

Another flow (traced with a dashed line) of 1 unit is sent from node *s*, through node 2, to node *t*. The flow along the edges in incremented by 1 unit.

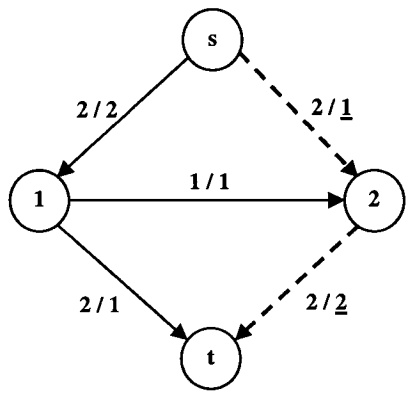

Figure 7. Another flow of 1 unit

At this point it would seem that no more flows can pass through the graph above. However, there still exists an augmenting path from node *s*, through nodes 2 and 1, to node *t*. The flow is augmented by 1 unit along this path.

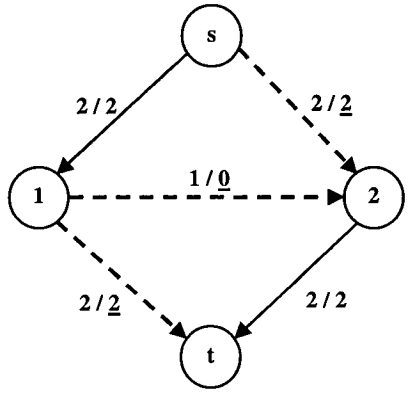

Figure 8. Augmenting flow of 1 unit

With the flow augmenting path algorithm such as Ford-Fulkerson algorithm we achieved a better maximum flow (as shown in Figure 9) than without using such an augmenting path algorithm (as shown in Figure 7).

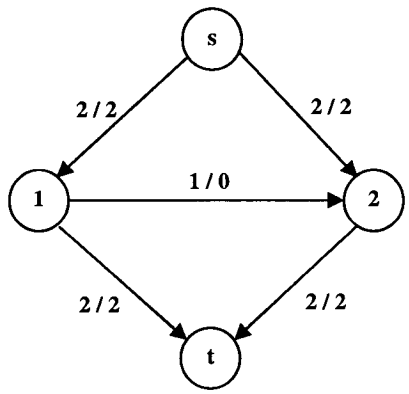

Figure 9. Maximum flow

2.5 Complexity of Maximum Flow Algorithms

In order to pick a good maximum flow algorithm, we must measure the efficiency of the algorithm based on its ability to perform well on the practical problem. We measure the algorithm based on its *worst-case complexity*. This is defined as the maximum number of

operations that the algorithm requires on a given problem instance of a given size. A network flow algorithm is called a *polynomial time* algorithm if its worst-case complexity is bounded by a polynomial function of the problem itself (number of nodes, edges and bandwidth capacity of the edges). For a given problem, the goal is to design a polynomial-time algorithm with the smallest worst-case complexity. There are many reasons to justify such a goal. First, this provides a framework in which we can compare different algorithms. Second, there is strong computational evidence suggesting that there is a high correlation between an algorithm's worst-case complexity and its performance in a practical application [5]. Finally, the study of polynomial-time algorithms has led to advances in the design of new algorithms that can be used in practical applications. Refer to Section 2.5.3 for a list of polynomial time algorithms.

2.5.1 Complexity of Ford-Fulkerson

As shown in the example in section 2.4.2, the Ford-Fulkerson algorithm can be used to better maximize the usage of network capacity. However, the complexity of the algorithm may hinder its use in real time network problems. The time complexity of the algorithm is:

$$O(m \times |F|)$$

where $|F|$ is the maximum flow value and m is the number of edges. The problem with this algorithm is that it is dependant on the maximum flow value F. For example if n is the number of nodes, and $F = 2^n$, then this algorithm would take exponential time [7].

2.5.2 Edmonds and Karp Algorithm

The Ford-Fulkerson algorithm correctly finds the maximum flow but does not use any heuristics to to improve the running time. Edmonds and Karp **[10]** used the idea that the shortest augmenting path could be chosen using breadth first search to improve the running time of the algorithm.

Edmonds and Karp **[10]** improved the complexity of the maximum flow algorithm to:

$$O(nm^2)$$

With this algorithm, the time complexity is not dependent on the maximum flow.

The Edmonds and Karp Algorithm solves any minimum-cost flow problem in a finite number of steps even when neither the capacities nor the costs are measurable in integer values [10]. However, the algorithm scales much better if capacities and demands have a coarser scale. This is described as the "Scaling Method for the Hitchcock Transportation Problem" in [10]. The number of computation steps required by the scaling method is proportional not to the capacities but to the number of digits in the binary representation of the capacities. In other words the number of computation steps is proportional to "logarithm base 2" of the capacities. Edmonds and Karp have proven that any Hitchcock Transportation minimum cost flow problem having $|A|$ arcs with capacities c and n nodes would require at most $((L + 2)|A|)$ flow augmentations, where $L = \log_2\left(\sum_i c_i/|A|\right)$ [8].

2.5.3 Complexity of other algorithms

Table 1 summarizes improvements on the algorithms since Ford-Fulkerson *(where n is the number of nodes, and m is the number of edges, the maximum flow is $|F|$ and capacities of arcs range between 1 and U).*

Year	Discoverer	Method	Running time	Reference
1956	Ford-Fulkerson	Augmenting Path	$O(m\|F\|)$	[7]
1970	Edmonds-Karp	Shortest Path	$O(nm^2)$	**[22]**
1972	Dinitz	Shortest Path	$O(mn^2)$	[24]
1972	Edmonds-Karp, Dinitz	Capacity Scaling	$O(m^2 logU)$	**[22]**
1973	Dinitz-Gabow	Capacity Scaling	$O(mnlogU)$	[25]
1974	Karzanov	Preflow-push	$O(n^3)$	[26]
1983	Sleator-Tarjan	Dynamic Trees	$O(mnlogn)$	[27]
1986	Goldberg-Tarjan	FIFO preflow-push	$mnlog(n^2/m)$	[28]
...	...	...	...	
1997	Goldberg-Rao	Length function	$O(m^{3/2}log(n^2/m)logU)$ $O(mn^{2/3}log(n^2/m)logU)$	[29]

Table 1: Maximum Flow Algorithms running time

2.6 Approximation Algorithms

For most practical applications, highly efficient algorithms are desired. These algorithms may not find the optimal solution to the problem, but are guaranteed to be "close" to optimal [5]. There are two reasons why these algorithms are highly desired. First, most of these algorithms are used in real-time applications. For such applications, it is preferable to tradeoff solution quality for computational speed. Most algorithms that are used in such applications provide high quality solutions in substantially less time than it would take to find the optimal solution. Second, for certain practical applications, it is acceptable to provide a solution with a certain level of precision since these practical applications have input and output with a certain level of precision [5]. For example, packets that flow into the network may not be equal to the packets that flow out because of packet loss due to congestion, resource failures, quality of media, reroute of data flow, and policing of flows based on quality of service or service level agreements.

A *ξ-approximation algorithm* is a polynomial-time algorithm for an optimization problem that is guaranteed to produce a solution that is within the factor of $(1-\xi)$ of the optimum solution. For example if ξ is equal to *0.001* for a given algorithm, then the *ξ-approximation algorithm* produces a solution that is 99.9% of the optimum solution, or within 0.1% of the optimum.

Radzik **[31]** observed that an algorithm known as a Fat-Path algorithm computes an ξ-optimal flow in $O(mn^2 \log B)\log 1/\xi$ time, where the capacities of the links of the network

is are integers between 1 and B. Radzik's variant of the *Fat-Path* algorithm **[32]** computes an ξ-optimal flow in $O(m^2+mn\log(\log B))\log 1/\xi$ time.

2.7 Combinatorial Algorithms

The problem can also be solved by combinatorial algorithms. Combinatorial algorithms exploit the discrete structure of the underlying network, often using graph search, shortest path, maximum flow, and minimum cost flow computations as sub-routines. These methods have led to superior algorithms for many traditional network flow problems including the shortest path, maximum flow, and minimum cost flow.

Despite a rich history dating back to Kantorovich, Fulkerson and Dantzig, until now, the only known way to solve the problem in polynomial-time was via general purpose linear programming techniques. Wayne **[31]** proposed the first combinatorial algorithm that solves the generalized maximum flow problem (flows with gains and losses, and links with costs). The algorithm uses an approximation scheme to find a good solution faster than an optimal one **[31]**.

3. MPLS OVERVIEW

MPLS stands for "Multiprotocol" Label Switching. It is "multiprotocol" because its techniques are applicable to any network layer protocol. In this thesis, however, we focus on the use of IP as the network layer protocol. A node, switch or router which supports MPLS is generally known as a "Label Switching Router" (LSR). The "Label Edge Router" (LER) is used to describe an LSR at the (ingress or egress) edge of the MPLS network.

As a data frame of a connectionless network layer protocol travels from the source to the destination it travels from one node to the next. Each node makes an independent forwarding decision for that packet. That is, each node analyzes the data frame's header to determine where to forward the packet next. The forwarding decision is determined by a forwarding table that is present on each node and that is built by network layer routing algorithms running on that node. Therefore each router independently chooses a next hop for the data frame, based on its analysis of the packet's header and the results of running the routing algorithm.

Frame headers contain considerably more information than is needed simply to choose the next hop along the path. Choosing the next hop can therefore be thought of as the composition of two functions. The first function partitions the entire set of possible packets into a set of "Forwarding Equivalence Classes (FECs)". In conventional IP forwarding the FEC is a subnet IP address prefix. Therefore a particular node will typically consider two packets to be in the same FEC if there is some address prefix X in

that router's routing tables such that X is the "longest match" for each packet's destination address. The second maps each FEC to a next hop. Insofar as the forwarding decision is concerned, different packets which get mapped into the same FEC are indistinguishable. All data frames which belong to a particular FEC and which travel from a particular node will follow the same path (or if certain kinds of multi-path routing are in use, they will all follow one of a set of paths associated with the FEC).

As the data frame traverses the network, each hop in turn re-examines the packet and matches it to a FEC in order to determine the next-hop.

In MPLS, the assignment of a particular data frame to a particular FEC is done just once, as the data frame enters the network. The FEC to which the packet is assigned is encoded as a short fixed length value known as a "label". When a packet is forwarded to its next hop, the label is sent along with it; that is, the packets are "labeled" before they are forwarded.

At subsequent hops, there is no further analysis of the data frame's network layer header. Rather, the label in the frame header is used as an index into a table on the node. The table entry specifies the next hop, and a new label. The old label in the frame header is replaced with the new label, and the data frame is forwarded to its next hop.

In the MPLS forwarding paradigm, once a packet is assigned to a FEC, no further network layer header analysis is done by subsequent routers; all forwarding is driven by the labels. This has a number of advantages over conventional network layer forwarding.

- Since forwarding packets are based on a label lookup rather than by analyzing the network layer headers, the speed at which packets are forwarded is greatly increased [36].

- Conventional forwarding, on the other hand, can only consider information which travels with the packet in the data frame header. With MPLS however, a data frame that enters the network can be assigned a FEC based on the port the frame arrives on. This allows the use of MPLS to offer VPN services.

- A data frame that enters the network at a particular node can be labeled differently than the same data frame entering the network at a different node or even a different port on the same node, and as a result forwarding decisions that depend on the ingress point can be easily made. This cannot be done with conventional forwarding.

- Sometimes it is desirable to force a data frame to follow a particular route through the network which is explicitly chosen at or before the time the packet enters the network, rather than being chosen by the normal dynamic routing algorithm as the packet travels through the network. This may be done as a matter of policy, or to support traffic engineering (the ability to apply QoS to data frames, the ability to determine the latency through the network, and the ability to determine usage of network resources). In *conventional* forwarding, since there is no explicit route through the network there is no way to determine latency or to guarantee the amount of resources available for packets flowing between a source and destination. For details on MPLS QoS refer to Section 3.4.

3.1 MPLS Label stack

The MPLS header is made up of a stack of 32 bit labels. Figure 10 shows a label stack entry. The MPLS Label is 20 bits long and is the identifier that is locally significant to the LSR. The experimental (EXP) bits field is 3 bits long and is used to determine the QoS that is to be applied to the data frame. The stack field takes one bit and is used to determine whether there is another label stack entry in the header. The Time-to-Live (TTL) field is 8 bits long and is similar to the TTL field carried in the IP header and is used to determine how many hops the frame can traverse before it is dropped.

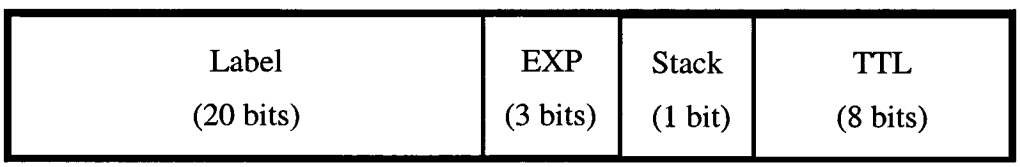

Figure 10. MPLS Label Stack Entry

The IP frame is encapsulated in with an MPLS header at the ingress edge of the MPLS network. At the egress edge, the IP frame restored by removing the MPLS header. Figure 11 shows an IP frame encapsulated with an MPLS header.

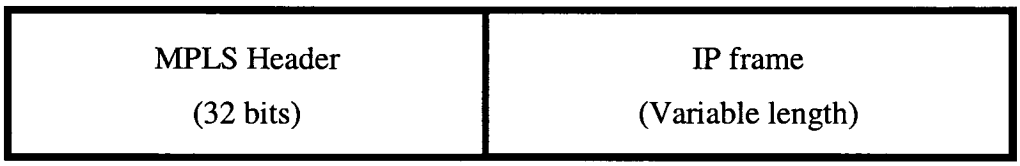

Figure 11. MPLS Header prepended to an IP Frame

3.2 Label Switched Paths (LSPs)

Label Switched Paths (LSPs) are specific traffic paths though an MPLS network. They are similar to circuit switched paths such as those found in ATM or Frame Relay networks and their use can guarantee a specific level of performance. Packets are forwarded along a *label switched path (LSP)* where each *label switched router (LSR)* makes forwarding decisions based solely on the contents of the label. MPLS enables routers to make forwarding decisions based on the contents of this shortened label, rather than by performing a complex route lookup based on the destination network layer address.

3.3 MPLS and RSVP

The first goal of adding RSVP (Resource Reservation Protocol) support to MPLS is to enable a label edge router (LER) or a label switched router (LSR) to classify packets by examining labels instead of Internet Protocol (IP) headers to recognize packets that

belong to a *flow* for which reservations have been made. RSVP is used to bind labels to reserved *flows*.

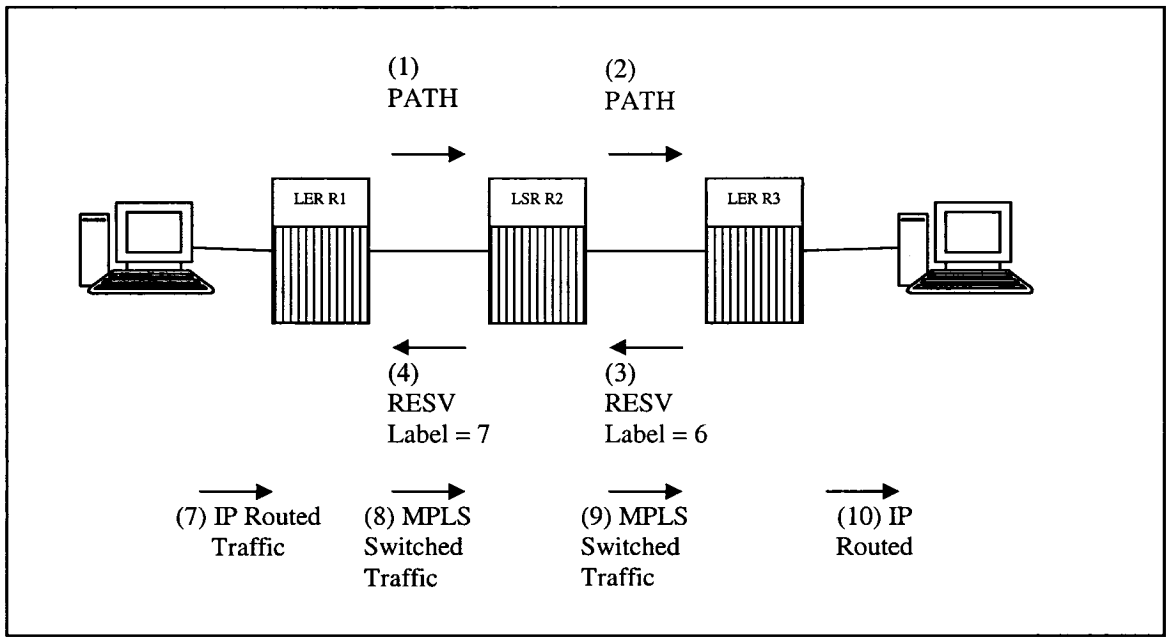

Figure 12. RSVP resource and label allocation

Figure 12 illustrates the process by which RSVP is used to reserve resources and allocate labels for a flow. The RSVP PATH message is used by LER R1 to tell LSR R2 (step 1), and LSR R2 to tell LER R3 (step 2) to set up an LSP. When an LER R3 reserves resources for the new flow, it allocates a label from its pool of free labels, and sends out a RESV message containing the label (step 3). Upon receipt of a RESV message with a label, the MPLS Agent on LSR R2 populates its forwarding table with this label as the outgoing label. LSR R2 then reserves resources and allocates a new label and inserts it into a RESV message before sending it upstream to LER R1 (step 4). If there are more LSRs on the path to the source, then this process repeats until the RESV message reaches the initiating LER. In the example above, LER R3 allocates a label "6" for this

reservation and advertises it upstream to its neighbour LSR R2. LSR R2 allocates a label "7" for the same reservation and advertises it to LER R1. At this point a label switched path (LSP) is setup for a *flow* from R1 to R3.

When an IP packet is received from the customer port at LER R1 (step 7), label "7" is "pushed" in front of the IP packet, and the resulting MPLS packet is sent to LSR R2 (step 8). At LSR2, the label "7" on the MPLS packet is "swapped" with the outgoing label "6", and the MPLS packet is forwarded to LER R3 (step 9). At LER R3, the label "6" is "popped" from the MPLS frame, and the IP packet is forwarded out the customer port (step 10).

3.4 MPLS Quality of Service (QoS)

The facilities provided by MPLS traffic engineering and Virtual Private Network (VPN) support are strong reasons for a Network Provider to deploy MPLS in a network [37]. The goal of MPLS QoS is to establish parity between the QoS features of IP and MPLS. One of the main reasons that MPLS supports, rather than extends, the IP QoS model is that MPLS, unlike IP, is not an end-to-end protocol. MPLS does not currently run in hosts, but runs on the router and switches of a provider's network.

Service providers do not sell MPLS services. They sell IP VPN services that offer IP QoS. MPLS helps providers offer IP QoS services more efficiently and on a wider range of platforms (IP routers and ATM switches). The two models of QoS architectures introduced by the Internet Engineering Task Force (IETF) are Integrated Services (also known as int-serv) and Differentiated Services (also known as diff-serv).

3.4.1 Integrated Services

Integrated Services provides a guarantee of minimum amount of bandwidth or some upper bound on the end-to-end delay for individual application flows (also known as microflows). RSVP was originally designed to support resource reservation for these microflows. Making reservations for individual microflows leads to scalability problems since the number of reservations that might be made across a network is likely to grow as fast as the numbers of users in the network. Therefore this type of QoS is not further discussed in this thesis.

3.4.2 Differentiated Services

Whereas resources are allocated on individual application flows on the int-serv architecture, the diff-serv model divides traffic into a small number of classes and allocates resources on a per-class basis. A simple diff-serv network can have just two classes: best-effort and premium QoS. Because diff-serv has only a few classes of traffic, the flow's QoS can be marked directly in the packet of the flow. This contrasts to the int-serv model, in which a signaling protocol is required to tell the LSR which flows of packets require special QoS. The field in the IP packet header is called the Differentiated Services Code Point (DSCP) [55]. DSCP can be marked on the packet by the host that sends the packets or by the LSR connected to the host according to its configured policy.

As mentioned in Section 3.4, Service providers sell IP VPN services. The packets that are sent from one customer host to another are IP packets. Figure 13 shows the format of

the header of an IP packet. Although Type of Service (ToS) [56] was defined in the early 1980s, it was largely unused until much later when IP traffic congestion in networks required the prioritization of the packets for better service levels. The 1 byte ToS field in the IP header includes 3 bits defining seven different priority levels with the highest priority reserved for network control packets. The remaining 5 bits were not used.

The DSCP byte replaces the ToS byte in the IP header. Currently, only the first 6 bits are used and the last 2 bits are reserved for future use. This allows up to 64 different classifications.

The DSCP is unstructured to facilitate the definition of future per-hop behaviors [55] , but it does reserve some values to maintain limited backward compatibility with the precedence bits in the ToS byte, because some systems do use these bits for controlling traffic.

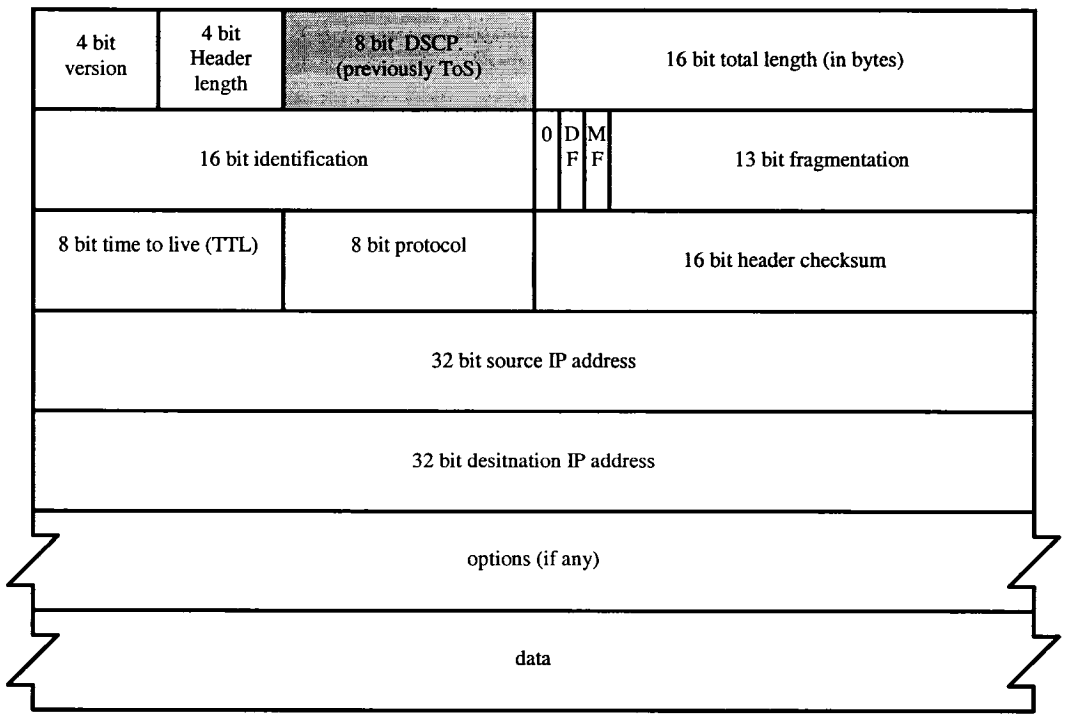

Figure 13. IP Header

Differentiated services (DS) is analogous to consumer-based differentiated service industries, such as transportation services. A customer can travel by bus, train, or airplane. Based on the mode of transportation, the customer may choose to travel first class, business class, economy class, or on standby. The class of service is characterized by how fast you reach your destination, how many stops you make, and what kind of service you receive. Some services may have limitations, such as when you can travel, and others, such as standby, include the risk of not reaching your destination in the time frame expected. In all cases, you pay more for higher quality services.

The Differentiated Services framework offers the same kind of classification system. Based on network policies, different kinds of traffic can be marked for different kinds of forwarding. Resources can then be allocated according to the policies and the DSCP markings on the packet.

For example, packets of a critical service such as video streaming can be marked with a DSCP that indicates low latency and a certain amount of sustained bandwidth.

Another simple example of such a policy would be to mark packet flows from higher paying customers as "premium", while those of lower paying customers as "best-effort".

3.4.3 MPLS Support of Differentiated Services

MPLS provides two ways of ensuring that packets marked with various DSCP bits receive the appropriate QoS treatment at each LSR in the network. One way to do this is by using E-LSPs (Exp-Label Switched Paths) and the other is by using L-LSPs (Label-Label Switched Paths).

3.4.3.1 E-LSPs

Since DSCP is carried in a 6 bit field in the IP packet header it allows 64 different values. MPLS has a 3-bit field in packet header defined for experimental use (also known as the Exp field) which allows only 8 different values. Therefore each LSP can be used to carry packets with up to 8 QoS classes. If a provider sells no more than eight different QoS

classes to its customers then this would be sufficient. But this may not be sufficient when providers want to offer many levels of services for customers paying at different levels. Even if the provider wants to offer no more than eight different service levels at first, E-LSPs do not allow the flexibility to expand in the future. As well, providers that want to only provide eight levels of services may prefer not using the same LSP to transmit packets of all eight classes. Therefore E-LSPs are not further discussed in this thesis.

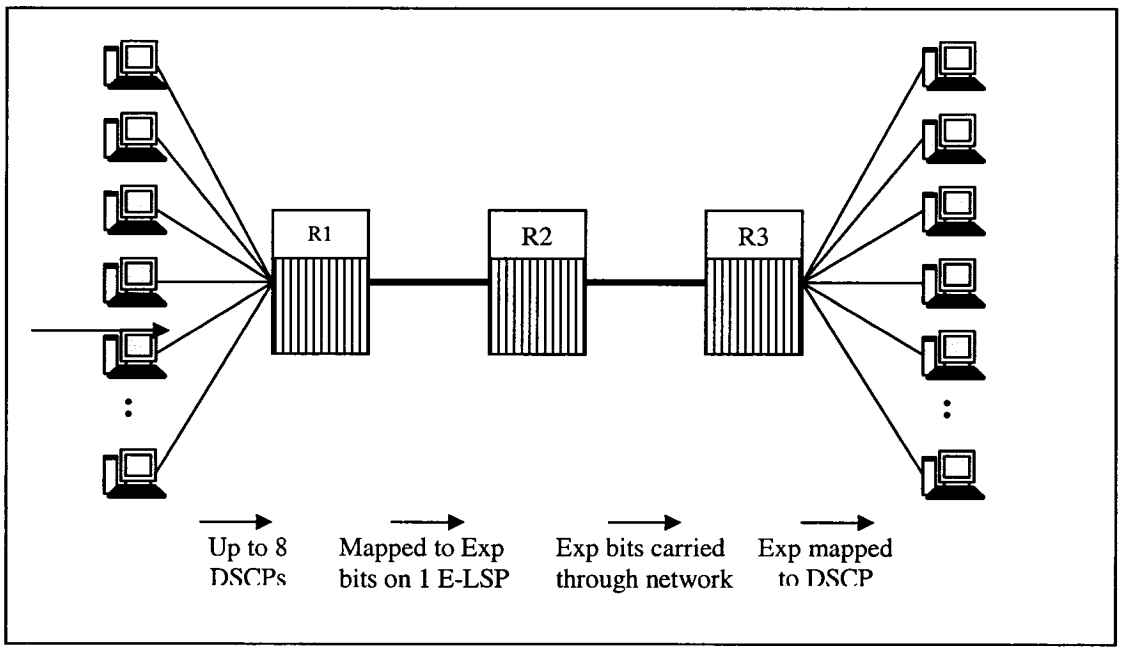

Figure 14. E-LSP

3.4.3.2 L-LSPs

If a provider wants to sell more than eight QoS classes to its customers then E-LSPs would not be sufficient. As well, if a provider wants to provide different QoS using an

MPLS over an IP and ATM network E-LSPs cannot be used since ATM does not have the Exp field. L-LSPs extend Label Distribution Protocols by allowing the QoS to be advertised with the label. Therefore each L-LSP will carry packets for one QoS (and the Exp bits are unused).

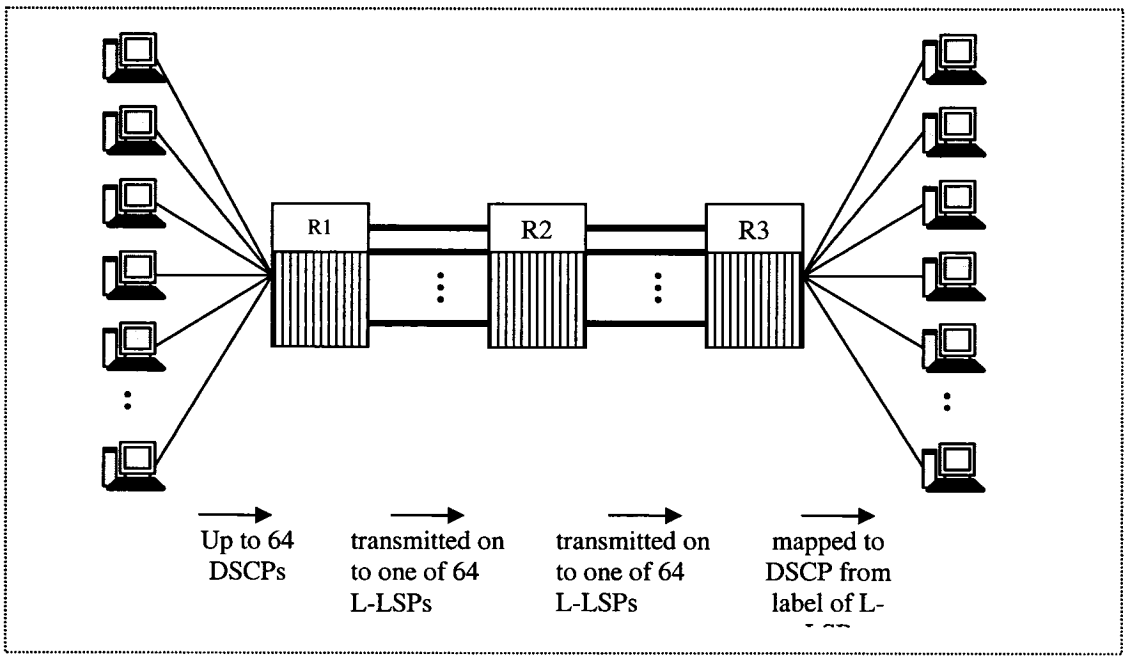

Figure 15. L-LSP

The RAA shown in Figure 29 can carry the QoS requested, while the RBA shown in Figure 30 can be used to advertise the label and the QoS for a LSP during the set up of an L-LSP.

4. PREVIOUS WORK ON MOBILE AGENTS

Mobile agents describe the concept of mobile computing or mobile code. Mobile agents roam networks, search for information, meet and interact with other agents that roam the network or are bound to a particular host. A mobile agent has four distinct characteristics, namely Intelligence, Communication, Autonomy, and Mobility **[12],[13]**. A mobile agent is able to adapt itself to its environment based on information available to it, hence its intelligence. A mobile agent is able to coordinate with other agents residing on the same host by exchanging data in order to plan future strategies, and hence its communication ability. A mobile agent has the authority to control its actions and strategies without the necessity of human control, and hence its autonomy. A mobile agent has the ability to migrate through a network, from host to host, performing specific tasks at each one, and hence the mobility.

The Agent based paradigm is applied to many parallel network applications. The mobile agent based approach benefits a large number of applications in communications networks. Some examples include load balancing, network management and network routing **[13],[14]**.

In this thesis, the use of mobile agents is incorporated into flow augmenting path algorithms to allow maximum utilization of network resources.

4.1 Stigmergy

The term *stigmergy* was first introduced by Grassé **[34]** to describe the communication taking place between agents through modification induced in their environment. The network optimization problem using the augmenting path algorithm is well suited for a multi-agent approach that uses the *stigmergy* paradigm used by ants. Ants indirectly communicate with each other through their environment. Ants returning from a food source to the nest lay down pheromones (a chemical substance) behind them. Other ants use the pheromone trail to find their way from the nest back to the food source. Over a period of time, as ants travel between the food and the nest, the shortest path will emerge.

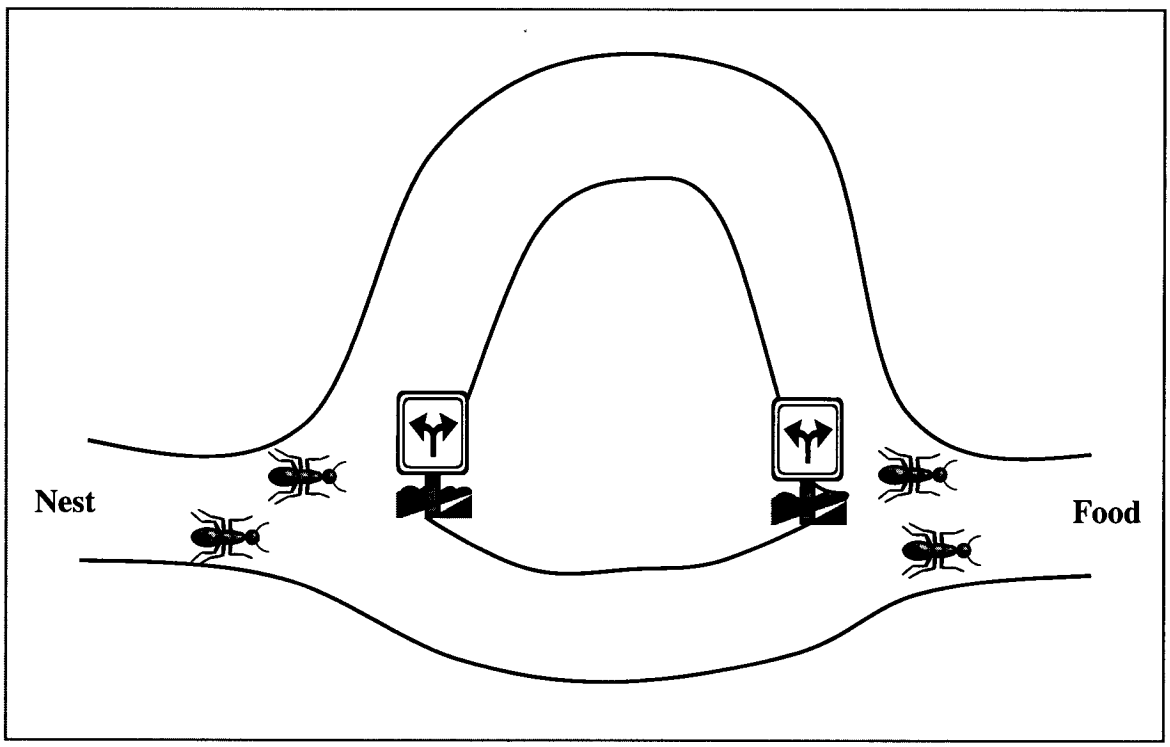

Figure 16. Ants finding best path

Figure 16 shows an example where ants have two paths between the nest and food. On arriving at the fork in the path, the ant makes a random decision on which path to take. The decision to take either path is of equal probability. The ants that choose the shorter path will arrive at their destination faster than the ants that choose the longer path. In a given amount of time an ant will travel over the shorter path more times than an ant that travels over the longer path. This will result in a higher level of pheromone over the shorter path than the longer path. Other ants will prefer to travel over the path with the higher concentration of pheromones than that of the weaker concentration. This will lead to the concentration of the pheromone on the shorter path to increase even further. Over time fewer ants will choose the longer path, resulting in the slow evaporation of the pheromone. The higher concentration of pheromone will result in the shortest path being selected.

4.2 Mobile Agents used in Congestion Control Mechanisms

The behavior of ants has inspired a congestion control mechanism that is highly adaptive to changes in network and traffic patterns **[39]**. The network model is populated by artificial agents that leave a "pheromone" on each node. The pheromone left at each node is a function of the congestion of a node and the distance the ant has traveled from the source node. The ant uses the pheromone to choose the next node in the network. Routing calls in the network are based on the distribution of the pheromone in the network.

Routing tables on node are replaced with probability tables (also known as pheromone tables). Every network node has a pheromone table for every destination node in the network. Each pheromone table on the node has an entry of every neighbor. The entries in the table are the probabilities that influence the ants' selection of the next node towards the destination node. At regular time intervals on each node, ants are launched to travel to a random destination. Ants move from one node to another, selecting the next node based on the probabilities in the pheromone tables. On arriving at a node, the ant updates the source node's probabilities in the node's pheromone table. When the ant reaches the destination node it dies.

The entry that is updated in the node's pheromone table is increased according to the following formula **[39]**:

$$p = \frac{p_{old} + \Delta p}{1 + \Delta p}$$

Where p is the new probability (for reaching a specific node through a neighboring node) and Δp is the probability or pheromone increase. The other entries in this table are decreased as follows:

$$p = \frac{p_{old}}{1 + \Delta p}$$

To enable ants to prefer shorter routes and avoid heavily congested nodes, the value of Δp must decrease with the age of the ant. To further influence path selection, ants must be delayed at congested nodes. Delaying ants at congested nodes slows them from updating of pheromone tables on other nodes thereby allowing the probabilities of

alternate nodes to increase. As well, the aged ants will have less of an effect on the pheromone tables of other nodes (since the value of Δp decreases with the age of the ant). When routing a call through the network to a particular destination, the node selects the next node based on the highest amount of pheromone (that is, the largest probability) in the pheromone table for the destination node. Newly provisioned calls change the level of congestion in different parts of the network. This influences how the ants update pheromone tables, which in turn influences how calls are routed through the network. This simple system shows how the implementation of a completely decentralized adaptive control system can be modeled on the behavior of ants.

4.3 Mobile Agents based routing algorithm

AntNet **[12]** is a routing algorithm for communication networks that is inspired by the behavior of ants. Each artificial ant (mobile agent) builds a path from the source to the destination node. While building the path it collects information about the time length of the path components and the load status of the network. The information is propagated back by another ant moving in the opposite direction and is used to modify the routing tables of the visited nodes.

The Antnet system consists of two sets of mobile agents called the forward and backward ants. The Antnet algorithm is described as follows:

1. At regular intervals a forward ant is launched towards a randomly selected destination node. At every node visited, the node identifier and the time elapsed since the ant was launched is pushed on to the ant's stack **[12]**.

2. The ant selects the next-hop node using information stored in the routing table. If the next-hop node has already been visited it randomly selects a node among the neighbors.
3. If a cycle is detected, the cycle's nodes are popped from the ant's stack and destroyed.
4. When the ant reaches the destination node it generates a backwards ant and transfers all the data to it.
5. The backward ant takes the path in the opposite direction of the forward ant. At each node it pops the stack in order to identify the next node in the reverse path.
6. At every node the ant updates the node with the elapsed time from the node to every node towards the destination node. As well, it updates the routing table of the node by incrementing the probability of getting to the destination through the node that the ant arrived from.

Forward ants share the same queues as data packets in order to accurately measure data congestion points in the network. Backwards ants use queues that are of a higher priority than data queues in order to update accumulated information as fast as possible.

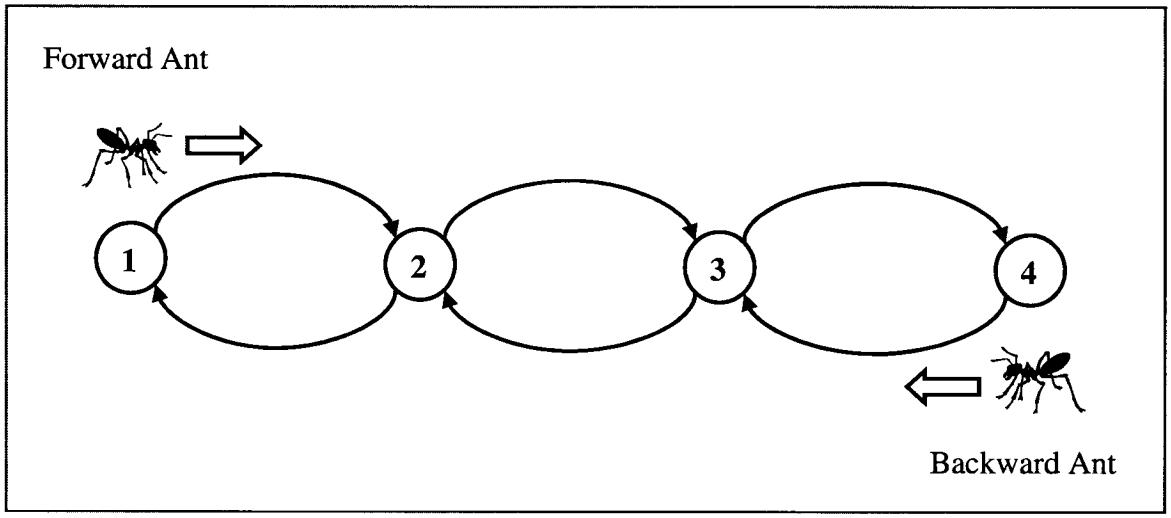

Figure 17. Example of AntNet Behavior

Figure 17 above shows an example of the AntNet behavior. The forward ant moves along the path from node 1 to node 2, then to node 3, and finally to node 4. The forward ant puts node 1 through 4 onto its stack along with the time elapsed. At node 4 it launches a backward ant and transfers the data to it. The backward ant uses the information from the stack to travel in the opposite direction, from node 4 to node 3, then to node 2 and finally to node 1. At each node, the backward ant updates the node's tables.

Classical routing protocols do not perform as well as Antnet **[12]**. Antnet is a robust, adaptive, and distributed routing system that implements an alternative scheme to classical shortest path routing protocols.

4.4 Mobile Agents in Ad-hoc Networks

Mobile agents have also been used as a routing scheme in Mobile Ad-hoc Networks (MANETs) [41],[42]. Routing schemes such as Destination Sequenced Distance Vector (DSDV) that continuously update the routing tables of mobile nodes by transmitting control messages between the nodes. This consumes a lot of network resources that could otherwise be used for data traffic. Routing protocols such as Ad-hoc On-Demand Distance Vector (AODV) and Dynamic Source Routing require communication between end users be delayed until a route can be determined. Combining the AODV and ant-based routing produces an Ant-ADHOV hybrid routing scheme that does not overload network capacity and is able to reduce route discovery latency and the end-to-end delay.

In Ant-ADOV, ant agents work independently to provide routes to nodes. The nodes also have the capability of initiating on-demand route discovery to find destinations for which it does not have a fresh route entry. The use of ants with AODV increases the number of un-expired routes a node would have at any given time. This reduces the number of route discoveries needed. Even if a node initiates a route discovery the probability of it receiving a successful reply from nodes nearby is increased resulting in reduced route discovery latency. As ant agents update the routes continuously, a source node can switch from a longer stale route to a shorter newer route. This leads to a decrease in the end-to-end delay. This makes Ant-AODV suitable for real-time data and multimedia communication.

4.5 Mobile Agents used in Network Management

Rather than managing a large network with a traditional centralized Network Management System, mobile agents can be used to manage networks in a distributed but cooperative manner. In comparison, traditional solutions appear to be less efficient, difficult to deploy or awkward [40].

Network faults must be diagnosed and fixed quickly, either automatically or by the network operator. In large networks, the network operator has to deal with a range or network components that are geographically dispersed. The components incorporate different network technologies, have different operator interfaces and tools, and have varying degrees of manageability.

For example a delegation agent (or a deglet) [40] can be injected into a network to automatically discover new network devices. Providing the deglet with constraints allows it to discover devices of a certain type. The termination of the deglet can be fixed in different ways, for example, after a certain number of hops, or after visiting a particular node a certain number of times.

A netlet is a permanent network agent. A netlet allows the network model to be maintained dynamically by discovering changes to the network configuration. A larger number of netlets in the network leads to a faster detection of changes in the network. There are certain upper bound constraints on the number of netlets in the network. Netlets and deglets can also be used to detect network faults and address the problem autonomously. The netlets inform the network manager of the event or if a recovery is not possible. Agents can also provide remote maintenance of devices that are located on

customer premises. The agent can be sent to the device to perform a suite of tests and to fix the problem. A network technician is only sent to the site if the problem cannot be fixed by the agent.

Heterogeneous networks are usually managed by several network managers. Mobile agents can be used to provision services through such networks. A network call or a permanent virtual circuit (PVC) can be assigned to a deglet that coordinates the provisioning. The coordinating deglet uses other deglets to perform partial tasks of provisioning the PVC in the heterogeneous networks.

Instead of remotely polling network elements to measure performance, a mobile agent can be launched to perform analysis on the device. Since there are fewer delays involved the measurements are accurate. A different version of the agent can be launched at any time to take difference measurements. Introducing a newer version of the agent to the network is easier than updating a local static monitoring agent on the device.

Mobile agents bring us closer to a plug-and-play network [40]. This allows networks to automatically adjust network devices when network problems are detected, or when a service is to be provided, or when a change is made to the network. This leads to reduced down time and reduced costs when managing networks.

4.6 Mobile Agents and Active Networks

Active Networks (AN) **[16]** allow network managers or users to program the network nodes according to their needs, offering a greater amount of flexibility. The nodes of an

active network are capable of loading and executing mobile agents. The mobile agent is transferred from one active network node to another within specialized signaling channels or within packets called *capsules*. A capsule may contain a reference to the location of the code such that the code can be downloaded from the reference when the first capsule containing the reference arrives at the node. Another type of capsule is one that contains the code itself, which is downloaded and run upon arrival at the node. Both types of capsules can be used to maximize bandwidth usage in the network.

4.7 NodeOS

The framework for the architecture of a router or switch in an active network is proposed in **[17]**. It included a supporting operating system, the NodeOS, one or more execution environments (EE), and active applications. The NodeOS is responsible for allocating and scheduling the node's local resources such as link bandwidth, CPU cycles, and storage. One or more EEs can run on top of the NodeOS. The EE defines a virtual machine. The EE is responsible for providing Active Applications (AAs) access to local resources. AAs can be dynamically loaded and executed on top of the EEs. AAs program the virtual machine provided by an EE to provide a specific service (such as reservation or freeing of bandwidth on a particular link). The general organization of these components is shown in Figure 18.

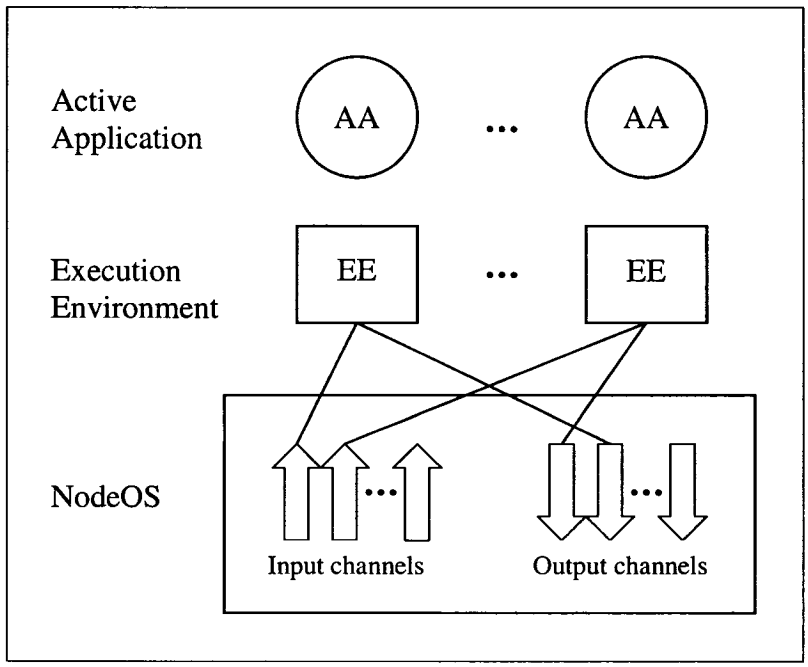

Figure 18. NodeOS, EE and AA

The NodeOS provides the basic functionality from which execution environments build the abstractions presented to the active applications. The NodeOS provides access to local node resources, as well as information about resource availability. The interface to the NodeOS is specified in **[19]**.

The NodeOS provides EEs with access to node resources through five primary resource abstractions: thread pools, memory pools, channels, files, and domains. The first four encapsulate a system's four types of resources: computation, memory, communication, and persistent storage. The fifth abstraction, the domain, is used to aggregate control and scheduling of the other four abstractions. The abstractions of interest in this thesis are domains and channels.

The domain is the primary abstraction for accounting, admission control, and scheduling in the system. Each domain contains the resources needed to carry a particular packet flow. Therefore a domain can be mapped to a flow as defined in Section 2.1.1. A domain typically contains the following resources: a set of channels on which messages are received and sent, a thread pool, and is associated with a particular memory pool. Active packets arrive on an input channel (inChan), are processed by the EE using threads and memory allocated to the domain, and are then transmitted on an output channel (outChan). Figure 19 shows a representative domain hierarchy.

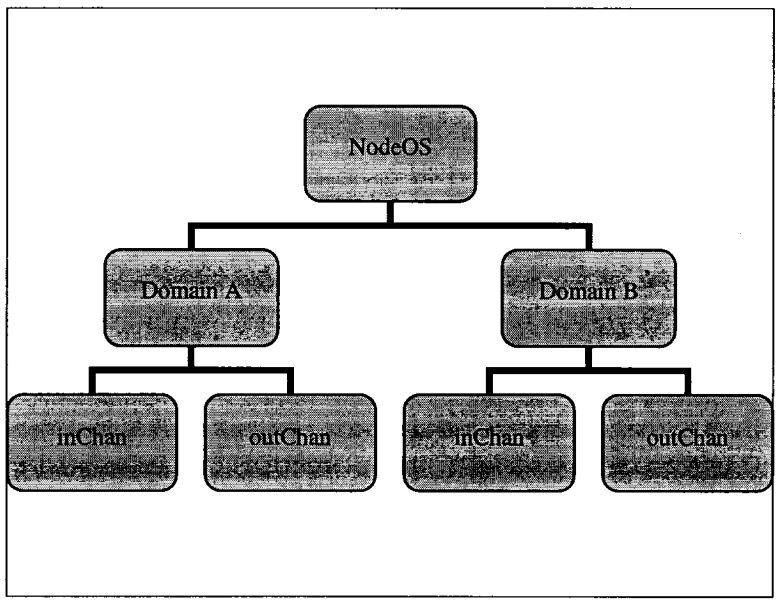

Figure 19. Domain Hierarchy

Channels are the primary abstraction for communication flows. Domains create channels to send, receive, and forward packets. The allocation of link bandwidth through the channel abstraction of the NodeOS is described with an RSVP-like QoS specification **[20]**. The channel abstraction is of particular interest to this thesis as it influences the design of a system that is used to set up MPLS RSVP LSPs on a node. Section 5.3.3 discusses the architecture that is used to implement an MPLS Agent EE on a network device such as a router.

4.8 A Mobile Agent System

The 2 fundamental concepts of the mobile agent model are the ***agent*** and its ***execution environment (also known as place)*** **[52]**.

4.8.1 Mobile Agent Attributes

A mobile agent is an entity that has 5 attributes:

1. State

   When an agent moves from one host to another it carries its state with it. It must do so in order to resume its execution at the destination host. The state of the agent is a snapshot of its execution at a given time.

2. Implementation

   Like any process or task, a mobile agent needs an implementation in the form of compiled source code in order to execute. The agent may move from one host to another with its implementation. It may also travel to a destination host and execute some code available on it, and retrieve any other code needed over the network. The agent implementation must be executable at the hosts it intends to travel to in the network. As well, the implementation must be safe for the hosts to execute.

3. Interface

   A mobile agent provides an interface that allows other agents and hosts to interact with it. The interface can either be a set of method signatures that allow access to

other agents and applications, or a messaging interface that allow communication through a well known language.

4. Identifier

   Every agent in the network must have a unique identifier that is immutable. Its uniqueness allows it to be used as the agent's key and a way to refer to a particular agent instance in the network.

5. Principals

   The principle of an agent is an entity whose identity can be authenticated by any host that the agent may try to access. The identity consists of a name and possibly other attributes such as the manufacturer (or author) and its owner (creator of the agent instance).

4.8.2 Concept of Place

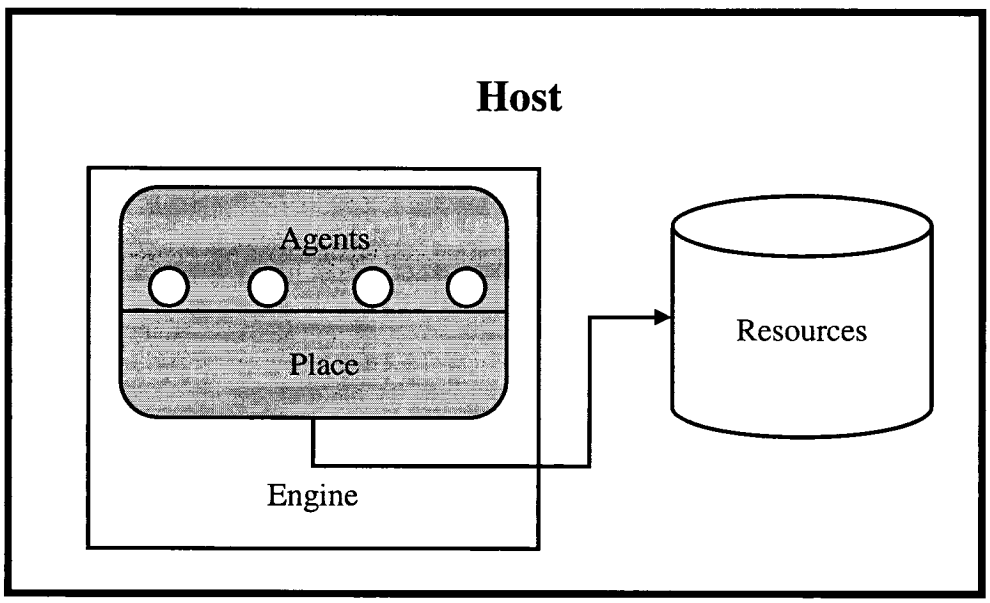

Figure 20. Place and Engine

Mobile agents travel between different *places* in the network. The place is the context in which the agent can execute (see Figure 20). The place is the entry point for a visiting agent that wishes to execute on a host. It provides a unique set of services that the agent can rely on irrespective of its location. There are 4 concepts that play an important role in *places*:

1. The Engine

   Places on their own cannot execute agents. The *engine* on a host provides the virtual machine for places and their agents. The *engine* provides *places* and agents with access to the network and links to the host resources. This concept has a hierarchical structure and is not limited to a one-to-one relationship. A

network entity can host multiple *engines*; each *engine* can contain multiple *places*, and each *place* can contain multiple agents. Each *place* requires a unique name in order to distinguish it from other *places* in the *engine*.

2. Resources

   The *engine* and *place* provide controlled access to host resources and services such as databases, file systems, processors, memory, hardware devices, software services and the network.

3. Location

   The location is a combination of the name of the *place* in which it executes and the network address of the engine in which the place resides. The location is typically written as the IP (Internet Protocol) address of the host and a port of the engine with a place name attribute.

4. Principals

   Like an agent, a place has two principals. It is associated with the manufacturer (the author) and the place master (the principal responsible for the operation of the place).

4.8.3 Mobile Agent Behavior

4.8.3.1 Mobile Agent Creation

An agent is created in a *place*. The creation can be initiated by another agent residing in the same place or by another agent or non-agent system outside the *place*. Before the creation of the agent is allowed, the creator must authenticate itself to the place. The creator can also pass initialization arguments for the agent. The class definition (implementation) of the agent can be present on the local or remote host, or can be provided by the creator. Creation involves the following three steps:

1. Instantiation and identifier assignment:

   The class definition of the agent is loaded and made executable, and instantiated to create the agent. The agent class specifies the implementation and the interface of the agent. The *place* assigns a unique identifier to the agent instance.

2. Initialization:

   Once instantiated, the agent starts initializing itself by using any arguments provided by the creator. Once the initialization is complete, the agent can assume that it is fully and correctly installed in the *place*.

3. Autonomous execution.

   After being fully and correctly installed in place, the agent starts execution. The agent is now capable of executing independently of other agents in the same *place*.

4.8.3.2 Mobile Agent Disposal

The disposal of an agent can be initiated by the agent itself, by another agent in the same *place*, or by an agent or non-agent outside the *place*. An agent can also be disposed of by the host if the lifetime of the agent has expired, if no one refers to or uses the agent, if there is a security violation, or if the host system is shutting down.

Disposing of an agent requires the following two steps:

1. Preparing for disposal:

   The agent is given a chance to complete its current task.

2. Suspension of execution:

   The *place* suspends the execution of the agent (i.e. it halts the execution thread of the agent).

4.8.3.3 Mobile Agent Transfer

The transfer of the mobile agent from one *place* to another can be initiated by the agent itself, by another agent residing in the same *place*, or by another agent or non-agent system outside the *place*. The originating host's *place* and the destination host's *place* are responsible for managing the agent's dispatch process. When the destination *place* is contacted by the originating *place*, it can accept the transfer request or reject it. The rejection by the destination *place* is communicated by a failure indication back to the origin *place*. If the origin place is unable to locate or communicate with the destination *place*, it returns a failure indication back to the agent.

4.8.3.3.1 Dispatching an Agent

In order to transfer an agent to a destination the agent must provide the destination's identifier. If the specific destination *place* is not provided, the destination host may choose to run the agent in a default destination *place*. Once the destination has been determined, the agent informs the local host to transfer it to the destination host. When the host receives the transfer request from the agent it informs the agent to prepare itself for departure and then suspends the agent. The host engine *serializes* the agent state and class. Serialization is the process of creating a persistent representation of the agent's object. This allows agent to be moved from one *place* to another. It then *encodes* the serialized agent so that it can be transported over the network using a chosen transport protocol. The originating host establishes a network connection with the destination host and transfers the encoded serialized agent.

4.8.3.3.2 Receiving an Agent

Before the destination host's engine can receive an agent it must determine whether it can accept an agent from the dispatching agent. Only after the dispatcher has been successfully authenticated will the encoded serialized agent be accepted. The engine decodes and deserializes the encoded serialized agent received. The agent is instantiated and its state is restored. The recreated agent is notified of its successful arrival to the destination so that it can initialize itself and resume execution.

4.8.3.3.3 Mobile Agent Communication

Agents communicate with other agents that reside at the same place, or with agents that reside in other places on the same engine, or with agents that reside in other places on other engines. To communicate with another agent, an agent can invoke a method of another agent or send a message to it. Agent messaging can be either peer-to-peer (between a pair of agents) or broadcast (to multiple agents). Communication between agents can be described as one of the three following schemes:

1) No-type messaging:

   This type of messaging is the most popular and commonly used scheme. A No-type message is synchronous and blocks further execution of the agent until the message receiver has completed processing the message and replies back to the agent.

2) Future-type messaging.

This type of messaging is asynchronous and does not block the agent's execution. The sender retains a handle which can be used to obtain the result. Because the sender does not have to wait until the receiver sends the reply, this messaging scheme is flexible and useful when multiple agents communicate with each other.

3) One-way messaging.

This type of messaging is asynchronous and does not block the agent's execution. The sender will not retain a handle for this message since it does not expect a reply for the message from the receiver. This messaging scheme is convenient when two agents are only needed to communicate in one direction. This scheme is also known as 'fire-and-forget'.

4.8.3.3.4 Agent Processing Strategies

Based on agent processing strategies, agents can be grouped into three different types [49]:

1. Reflex or Reactive agents.

These agents respond soley to external stimuli and information available from sensing the environment. These agents act on an event (or stimulus), and determine what action to take based on predetermined conditions. Applications of these agents have been limited to field of robotics [49].

2. Goal-directed or Deliberative agents.

   These agents try to achieve a specific goal by having domain knowledge and planning capabilities that allow it to take a sequence of actions.

3. Collaborative Agents.

   These agents work with each other to reach a specific goal. While each agent is autonomous, communication between agents is a key factor in solving problems. Colaborative agents are used to solve large problems and they allow a modular approach based on specialization of agent function or domain knowledge.

In a collaborative agent system, the agents exchange information about their beliefs, desires and intentions. The beliefs represent that knowledge about the state of the environment. All the agent's planning and subsequent actions are based on its beliefs. The desires turn into goals when the agent is reasoning on how it wants to change its environment. The courses of action are called intentions.

5. PROPOSED ALGORITHM

The algorithm proposed in this thesis is an extended version of Edmond-Karp Algorithm, (see section 2.5.2). There are several reasons this algorithm is best suited for the application proposed in this thesis. The first and main reason is the way in which MPLS paths are setup. The two phases of the Ford Fulkerson Algorithm are suited to MPLS. However, since the algorithm's complexity (Section 2.5.1) hinders its use in real time network applications the two phases of the Edmonds Karp Algorithm better suite the applications. MPLS paths are set-up using the PATH and RESV messages as discussed in Section 3.3. The Edmonds-Karp algorithm is used for the purpose of both setting up an MPLS path and ensuring that the network bandwidth is optimally used, and for QoS (as discussed in Sections 3.4 and 5.3.6). A second reason is that the algorithm is well suited to *forward* and *reverse* mobile agents. The forward agent starts at a source node *s* and follows a certain path collecting information along the way. When the agent reaches the destination node *t*, it spawns one or more backward agents, transfers its information to the new (backward) agents, and dies. The backward agents follow reverse paths back towards the source node. Along its path, each backward agent uses the information it received from the forward agent to update the nodes. This suits the two parts of the Edmond-Karp Algorithm.

The proposal is to extend the Edmond-Karp algorithm to allow it to select links in a network by taking several *cost* factors into account as well as to allow the ability to "bump" paths of a lower *priority*.

5.1 Cost Factors

The cost would include one or a combination of capacities available on a link (for load balancing), the latency over the link, and the monetary cost of using the link (a service provider may lease links from one or more service providers), the time of day the service is provided, and more. The combination used to determine the weight of each cost does not need to be fixed and can be based on how much a customer is paying for a given service. As well, the cost of the link can be dynamic. For example, the cost can be based on the time of day the service is offered or the amount of link capacity that is available at a given time.

The "cost" m on a given link (u,v) can be defined as:

$$m(u,v) = \sum \omega_i \; m_i(u,v)$$

where ω is known as the weight for a given cost factor, and has the following properties:

$$\forall i,\ 0 \leq \omega_i \leq 1, \text{ and}$$

$$\sum \omega_i = 1$$

Based on what the customer is paying, the network provider can assign different weights to different cost factors when provisioning the MPLS path.

Table 2 shows an example of how a provider may factor a link's fixed costs and the dynamic costs in the network. The fixed costs include such things as the latency of the link, jitter of the link, the reliability of the link, and the providers it is leased from.

Cost description	Cost Factor	Weight	Cost on link	Total link cost
Latency	20	0.4	8	44
Jitter	60	0.2	12	
Reliability	100	0.2	20	
Leased from provider 1	20	0.2	4	
Leased from provider 2	20	0.0	0	
Leased from provider 3	20	0.0	0	

Table 2: Cost Factors

There are also other dynamic factors that influence whether a link is chosen for a particular path. The dynamic costs include the available link capacity and the time of day the service is provisioned. The dynamic cost factor (for the link) may be valid for a given snapshot in time. This value may be different at another snapshot in time. Dynamic costs make up a significant factor in determining whether the link should be used. As well, the weight of each cost should be determined by the how much the customer is willing to pay for the given service. These combinations determine whether a link in the network would be optimal for a given type of service for a given customer at a given time. The use of mobile agents allows the flexibility of the combination to be selected at the time of the MPLS path setup.

5.2 Priorities

Comparing the "priority" of LSPs determines whether the new LSP can "bump" an LSP that is already provisioned on a node. The priority can be based on how much the customer is paying or the importance of the service. An LSP with a higher priority is allowed to bump an LSP of a lower priority. Bumping results in a higher priority LSP taking a more optimal path through the MPLS network (based on the costs described above), and the bumped (lower priority) LSP taking a less optimal path through the network.

The way this is achieved is by modifying the auxiliary graph based on priorities of the existing flows. Let's take a simple example with 3 priorities, "High", "Medium" and "Low". For example, Figure 21 shows 4 feasible flows *{s,a,b,t}*, *{s,c.d,t}*, *{s,c,b,a,d,t}*, and *{s,c,b,t}* of priority "Medium" and 1 unit each.

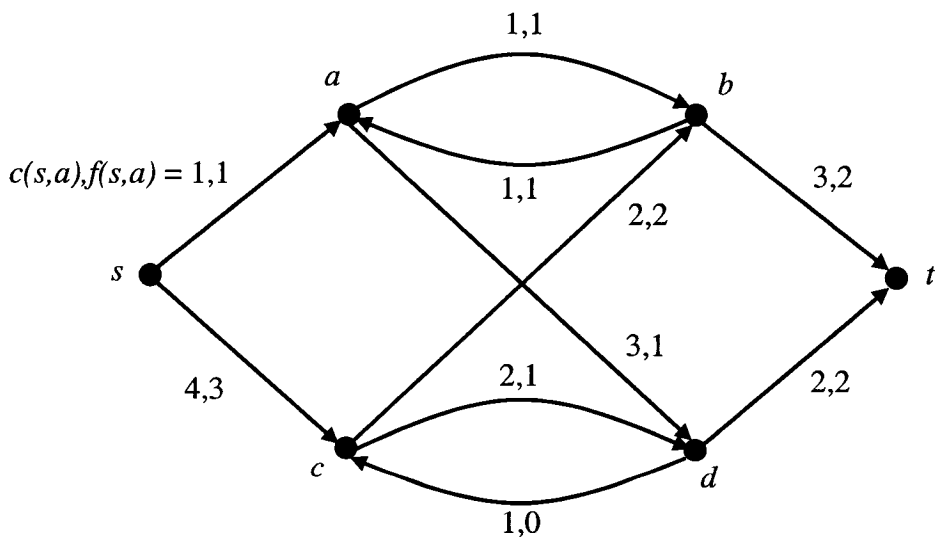

Figure 21. Feasible flow digraph

In the absence of any priorities, Figure 22 shows the auxiliary graph for Figure 21.

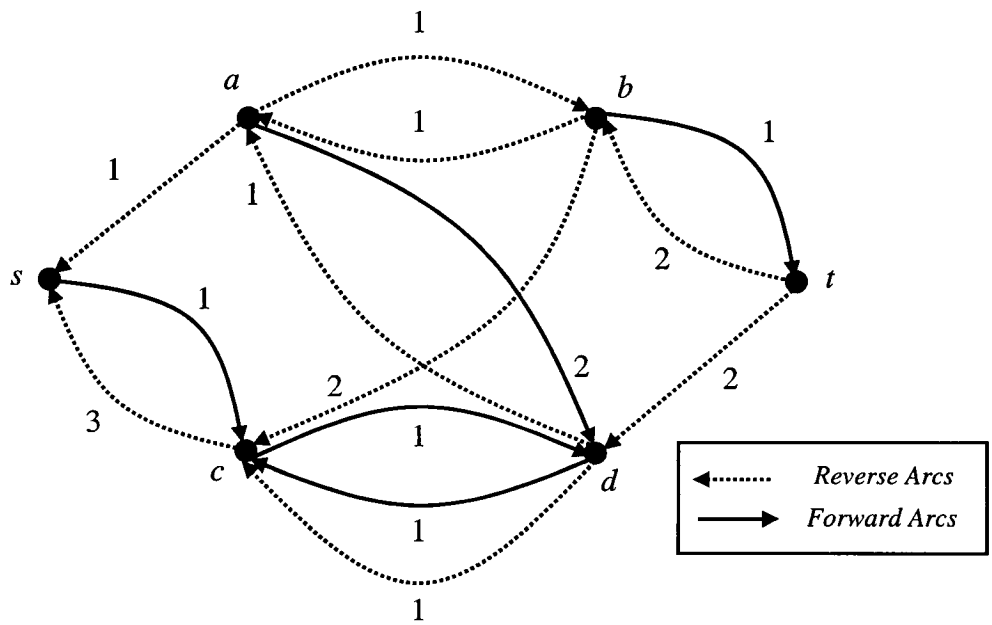

Figure 22. Auxiliary digraph

Now suppose we want to add a flow of priority "Low". In order to avoid bumping higher priority paths we must modify the auxiliary graph such that only reverse arcs where paths are of an equal or lower priority than "Low" should be present (in this case only paths of priority "Low"). The forward arcs of the auxiliary graph remain unchanged. Figure 23 shows the modified auxiliary graph for Figure 21. The auxiliary graph shows no more feasible flows possible in the graph. This ensures that higher priority paths do not get bumped in favor of lower priority ones.

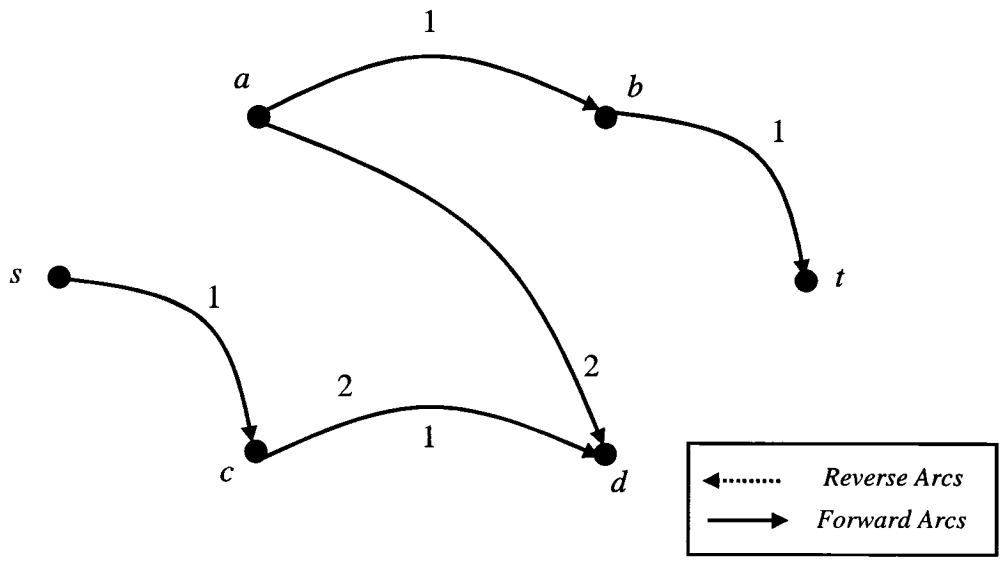

Figure 23. Auxiliary digraph

Figure 22 shows that a flow of priority "Medium" or "High" can be added. This means that by the process of path augmentation, a new flow may "bump" an existing flow such that the "bumped" flow takes a less optimal path in order to maximize the use of the available capacity.

We take this modification one step further. For "High" priority paths we create an auxiliary graph such that a "Low" priority path is "bumped" completely. The way to achieve this is to leave the reverse arcs for "Low" priority flows out of the auxiliary graph and add the capacity taken by such flows to the forward arcs.

For example, Figure 24 shows the auxiliary graph if we wanted to add a "High" priority flow and if flow *{s,c,b,t}* were of priority "Low".

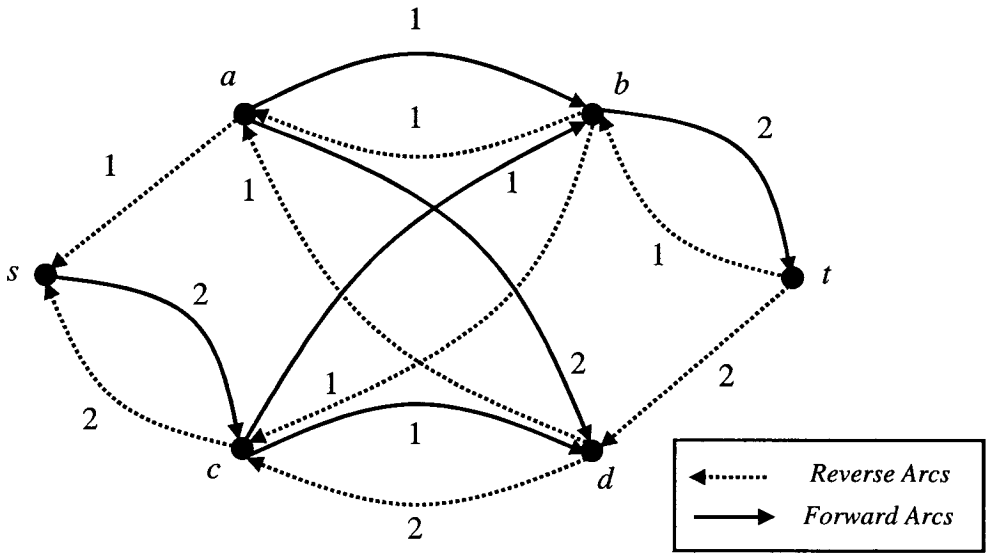

Figure 24. Auxiliary digraph

The result is a more optimal use of network resources based on the costs of providing the service and the amount paid by the customer for the service based on the path priority.

5.3 Using Mobile Agents to implement the Algorithm

Using the mobile agents changes the process of maximum flow discovery from the nodes to the agents. Hence in this approach, the agents carry link utilization information from one node to another rather than propagating the information through a protocol from node to node, or have a central network management station receive this information.

Mobile Agents are adapted to improve bandwidth usage in MPLS networks. Current link costs and bandwidth usage is measured by an artificial forward mobile agent that finds the optimal augmenting path to a destination. Artificial reverse mobile agents travel from

the destination back to the source and improve the network bandwidth resources by using the information gathered by the forward agent.

As discussed in Section 2.4.1, the Ford-Fulkerson Algorithm is made up of two routines. One routine is used in the forward direction and the other in the reverse direction. The Edmonds and Karp Algorithm discussed in Section 2.5.2 is an improvement to Ford-Fulkerson Algorithm and also has a forward and reverse routine. The following sections discuss how mobile agents are used to improve network resource usage.

Using a static agent, applying the algorithm to a network would require a centralized Network Management Station (NMS) that keeps track of the nodes, links between the nodes, their maximum capacities and utilization at a given time. Resource optimization would be done on the centralized station, and the updates to the network resources would require messaging between the NMS and the various nodes on the network. The problem with this approach is the single point of failure, the inability to scale to large number of nodes, the large number of messages that would need to travel between every node in the network and the NMS, and the inability to support dynamic resource reservation that is not initiated by the NMS. The performance analysis obtained by using mobile agents compared with managing a network using an NMS shows the mobile agent is less sensitive to the latency and the bandwidth of the bottleneck link that connects the NMS to the network [38].

Another approach would be to run a static agent on each and every node. This approach would require messaging between the nodes in the network, which would consume a significant amount of network resources. As well, this approach would require every

node to use the same algorithm, and hence the same software to accomplish this task. The algorithm would require the knowledge of all the different criteria for selecting paths through the network and be able to handle the dynamics of the network. This would require writing very flexible but highly complicated software. This increases the likelihood of unstable software. The software would need to be constantly updated to improve the algorithm and to increase stability. This is not desirable for service providers since revenue is dependent on the network credibility and the level of service provided to their customers.

A third approach would be to use mobile agents. The exchange of link utilization information and the process of optimizing network resource usage moves from the nodes to the agents. Hence in this approach, improving resource usage is manifested in the movement of agents that carry the residual capacity information from one node to another rather than propagate messages that carry this information. The algorithm used to search for an augmenting path can be changed at any given time by using a different instance of a mobile agent. One instance of the agent could search for an augmenting path using the least links with the least cost, whereas another instance of an agent could search for an augmenting path using least network latency, and another using random links. An agent could also use a combination of these factors in determining which link to pick. This allows service providers to introduce new versions of the algorithm at any given time without affecting their networks and the services they provide to their customers.

5.3.1 Agents that implement Ford-Fulkerson routines

In section 2.2, the Ford-Fulkerson routines were introduced to show how the algorithm is implemented. Both these routines start at the source node *s*, and end at the sink node *t*. Routine A and Routine B can be run on a single agent that resides on a centralized NMS or on mobile agents. When mobile agents migrate from one node to another in the network they consume network resources. Therefore it would be advantageous to have Routine A implemented on a separate agent from Routine B, making the agent size smaller. The disadvantage of having different agents that implement Routines A and B is the overhead of creating and spawning a new thread at the destination node (for Routine B). However, for this specific application, a new agent must be created for every existing path that needs to be updated in the network. If a new path can be created without having to update existing paths, and if the overhead of spawning a new agent is greater than the overhead of transporting a larger agent (from the source to the destination and back to the source), then it is better to implement both Routine A and B in a single agent. However, for this specific application, when the request to find a new *s-t* path is made at the source node, the number of existing paths that may need to be updated is not known. As well, the distance to the destination node is not known. Therefore it is proposed that the routines A and B be implemented in smaller agents.

The Routine A Agent (RAA) travels in the forward direction from node *s* to node *t* in search of a flow augmenting path. Figure 25 below shows the RAA behaviour. The selection of the next node is based on the algorithm described in Section 2.4.1.

For details on the type of information carried by the agents, and the information left on the router for other agents please refer to Section 5.3.5.

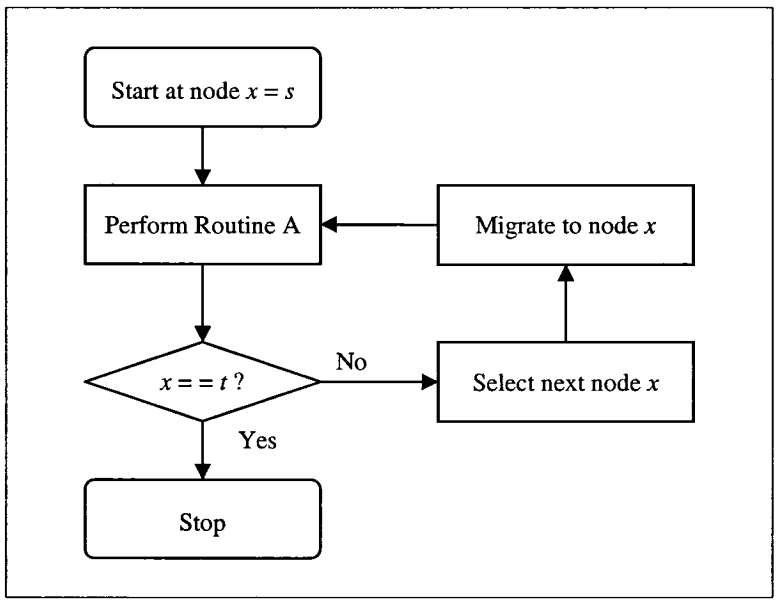

Figure 25. RAA Behaviour

Routine B Agent (RBA) travels in the reverse direction from node *t* to node *s* changing the flow of the path. Figure 26 below shows the RBA behaviour.

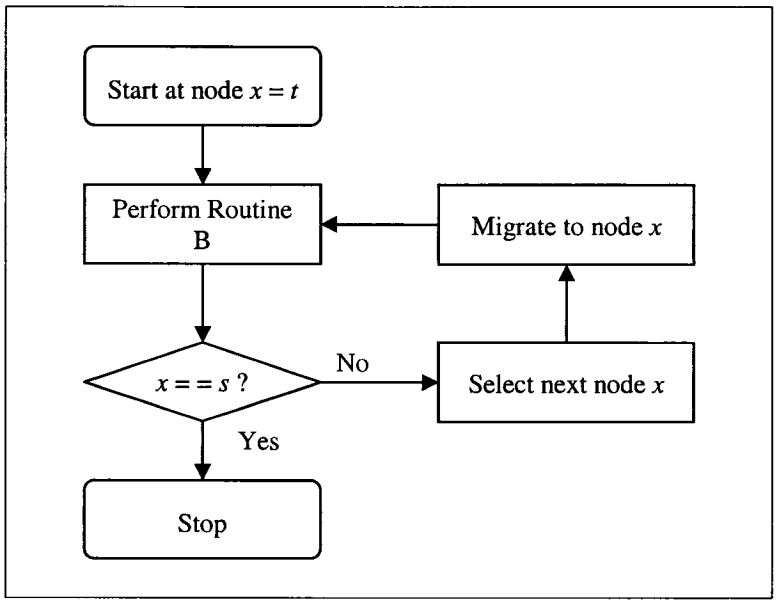

Figure 26. RBA Behaviour

5.3.2 Agents that implement the Edmonds-Karp Algorithm

Since Edmond-Karp is an extension of the Ford-Fulkerson algorithm, the behaviour of the RAA agent discussed in Section 5.3.1 is easily extended to allow it to select a link based on cost factors (Section 5.1) and path priorities (Section 5.2). As well, it is used to select links based on several factors as discussed in Section 5. Figure 27 below shows extension to the RAA behaviour to take cost factors and path priorities into account when selecting the next node.

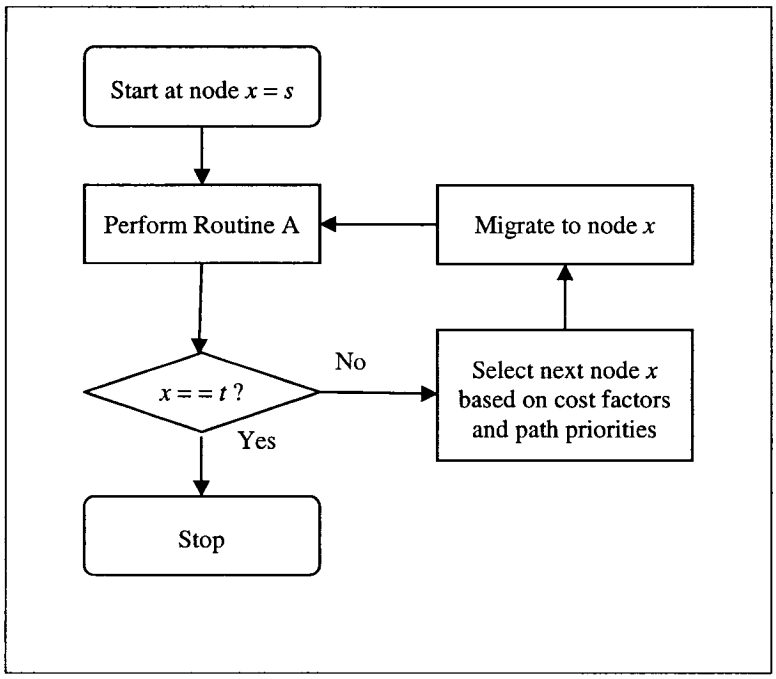

Figure 27. RAA Behaviour

5.3.3 System Architecture

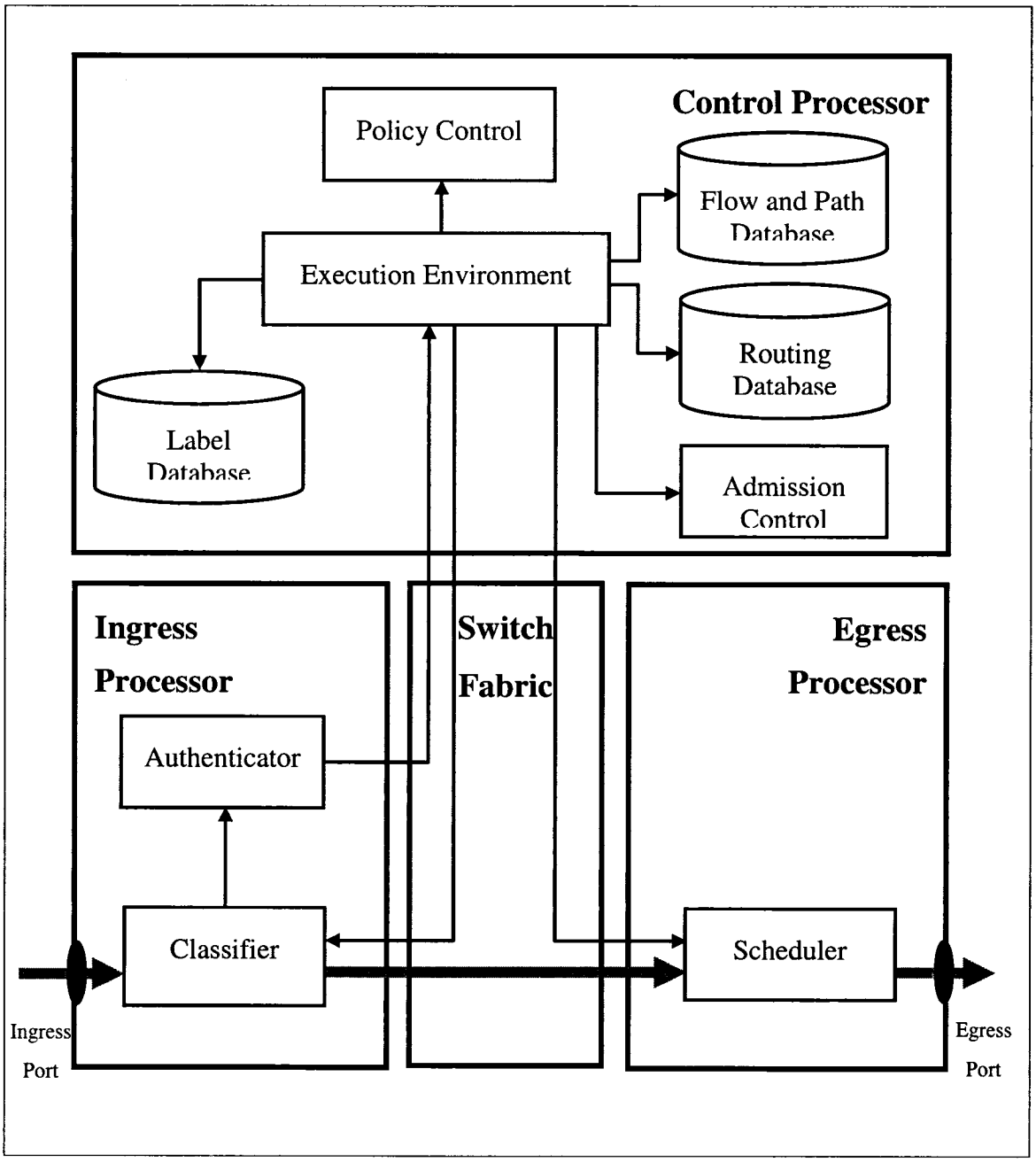

Figure 28. System Architecture

Figure 28 Shows the Router's System Architecture. The router is classified into 3 categories of physical control processors, and a switch fabric that allows communication between the processors. The main router processor is called the *Control Processor.* The *Agent Execution Environment* is located on the *Control Processor.* The mobile agent arrives at the router through the *Ingress Processor.*

The *Classifier* on the *Ingress Processor* is used to distinguish packets that contain control information that is destined for the router and packets that contain regular data that is destined to other routers or hosts. The *Classifier* forwards IP and MPLS data packets from an *Ingress Port* (on the *Ingress Processor*) through the *Switch Fabric* to the appropriate *Egress Port* (on the *Egress Processor*). Once the *Classifier* recognizes the agent as a control entity it forwards it to the *Authenticator* to ensure its integrity. Once authenticated, the mobile agent is forwarded to the *Control Processor* (through the *Switch Fabric*) where it is allowed to execute on the *Agent Execution Environment.*

The *Routing Database* is used to determine the port that is connected to the next-hop of the path to the destination router. The *Flow and Path Database* is used to store LSP flow information (for existing LSPs), and the Path information (for new LSPs being created) as discussed in Section 5.3.5. The *Label Database* contains labels that are available for use by the router. The *Policy Control* module determines whether the agent has administrative permission to make the reservation. The *Admission Control* determines whether the node has sufficient resources for the reservation. Once the *Policy Control* and *Admission Control* checks have passed, the reservation is made in the *Classifier (Ingress Processor)* and the *Scheduler (Egress Processor).* This allows the *Classifier* to

forward data packets from the *Ingress Processor* to the *Scheduler* on the *Egress Processor.* The *Scheduler* is responsible for forwarding the packets out the *Egress Port.*

Once the request or a reservation has been completed on the router, the agent is forwarded from the *Control Processor* to the *Scheduler* on the *Egress Processor.* The *Scheduler* is responsible for forwarding the agent out the *Egress Port* to the next-hop node. For details on the type of information carried by the agents, and the information left on the router for other agents please refer to Section 5.3.5.

5.3.4 Using Mobile Agents to set up MPLS flows

The RAA and RBA agents discussed in Section 5.3.2 can be mapped to provide services in an MPLS backbone. Rather than send a PATH message from R1 to R2, and from R2 to R3, the RAA agent can be extended to migrate from one router to the next and call the PATH interface on the MPLS Agent. Similarly, the RBA can be extended to call the RESV interface of the MPLS Agent, and migrate to the next router with the allocated label.

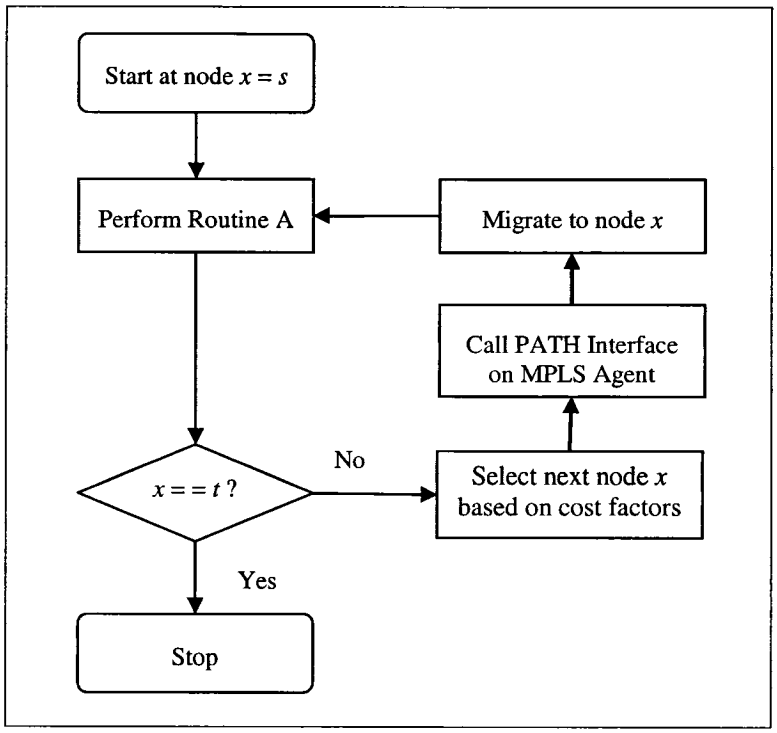

Figure 29. RAA PATH Behaviour

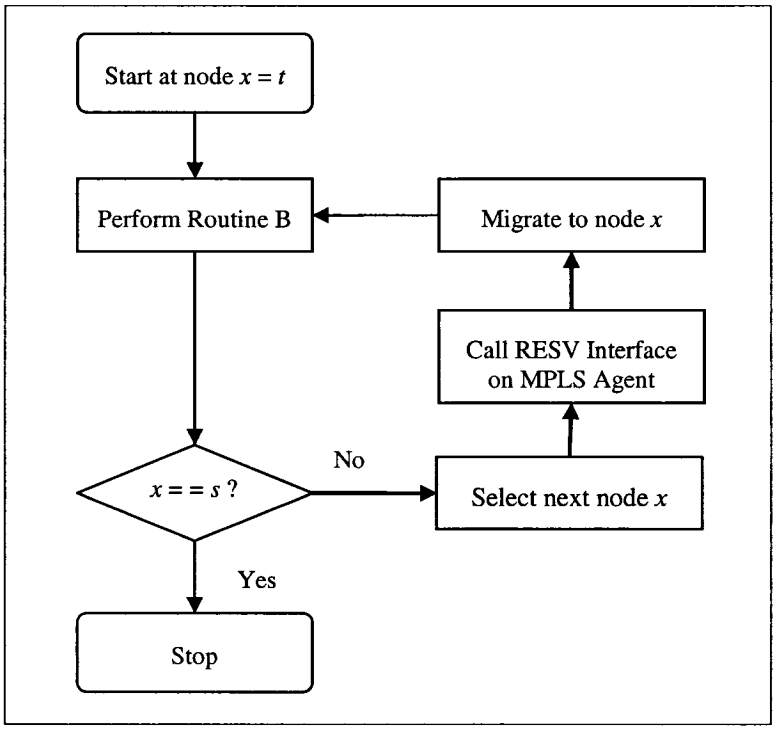

Figure 30. RBA RESV Behaviour

5.3.5 Information carried by Agents

The RAA carries the IP Address of the network source router, an LSP Id which contains a 16 bit value that is unique to the source router, the path priority and the QoS requested for the path and the IP Address of the destination router in the network. Optionally, the request may contain an explicit path through the network. The source IP Address and the LSP Id gives the flow request a network wide unique identifier known as the Flow Id. The path priority is used to determine what other paths can be pre-empted in order to satisfy the new path request (as discussed in Section 5.2). The port that the RAA enters the router is identified as 'PA'.

The RAA uses the information it carries to determine the router resources required. If the router accepts the request, the Flow Id (Source IP Address and the LSP Id) and the *ingress physical port identifier* 'PA' are stored in the Path Database as a *path database entry*. The Path Database entries allow other agents traveling in the reverse direction to retrieve information on how to get back to the source node. The entry can be retrieved from the Path database at a later time using the Flow Id as the key.

The RAA uses the destination IP Address to look up one or more next-hop entries in the routing database. Each routing entry contains the port 'PB' that connects the router to the next-hop router. The RAA requests the router to send it to the next-hop router via the egress port 'PB'. This is repeated until RAA arrives at the destination router.

Once the resources have been successfully allocated and all the request information stored at the destination router, the RAA instantiates the RBA. The Flow Id and the ingress port identifier are transferred to the RBA before the RAA terminates. The RBA requests a label 'L1' to be allocated from the *Label Database* (See Figure 28). Using the Path database entry created by RAA, the RBA creates a LSP Flow database entry for the new LSP. The RBA then requests the router to forward it back through the ingress port towards the source router.

At the next upstream node, the RBA is sent to the *Control Processor* along with the physical port identifier 'PB' that it arrived on. The RBA uses the Flow Id to find the path record in the path database (this contains physical port identifier 'PA'). It then makes a request for a new label 'L2' from the label database and creates a LSP Flow database entry. Through the execution environment, RBA programs an entry in the ingress

Classifier so that any data packet coming in on port 'PA' with label 'L2' is forwarded through the switch fabric towards the *Scheduler* on the egress processor corresponding to port 'PB'. It also programs an entry in the egress *Scheduler* so that any data packet that comes from the switch fabric with label "L2' is swapped with label 'L1' and forwarded through port 'PB'.

Once RBA has successfully completed the label programming it swaps the label 'L1' that it carried with new label 'L2'. It requests the router to send it to the next upstream router through port 'PA'. This process is repeated until the RBA reaches the source router where it transfers the LSP flow data to the router and terminates itself. At this point data packets can be forwarded from the source to the destination router through the LSP that has been set up.

5.3.6 Using Mobile Agents to setup QoS for a flow

MPLS LSPs are used as traffic trunks in provider networks. A traffic trunk LSP is defined as an LSP that is used to transport data from a collection of IP, TCP or UDP flows known as microflows. Microflows are flows that have two common properties. The first is that all traffic in microflows comes from one or more customer sites attached to the same source LER (for example R1 in Figure 31) and are destined to one or more customer sites attached to the same destination LER (for example R3 in Figure 31). The second is that they demand the same QoS. Therefore, microflows follow the same common path through the network and can use the same trunk LSP.

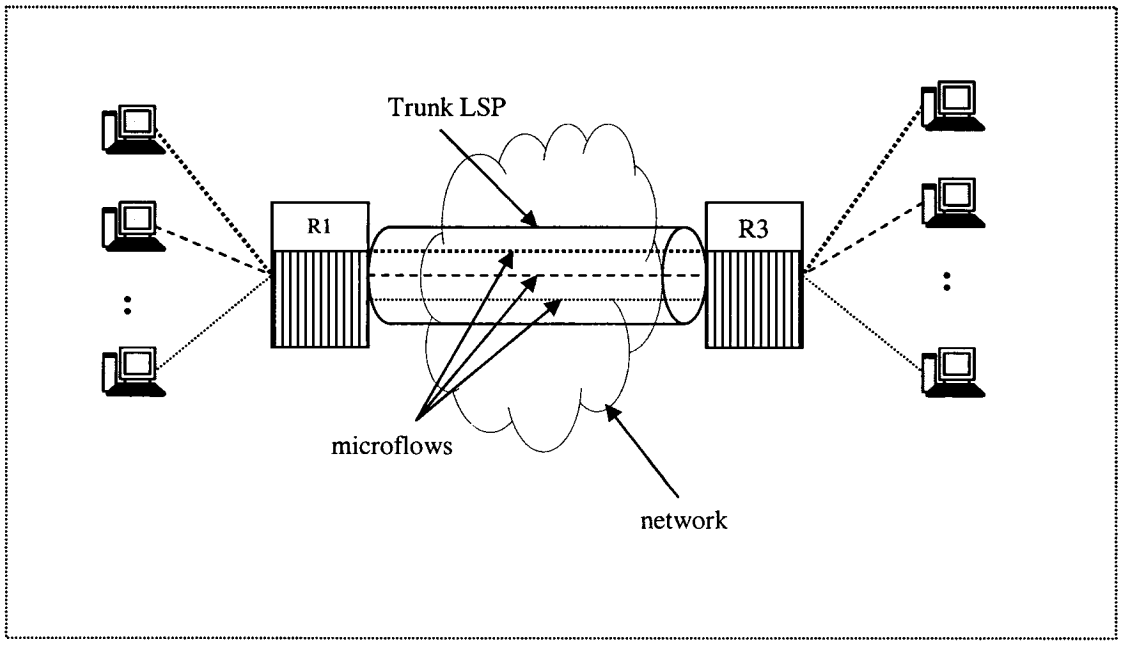

Figure 31. Trunk LSPs and Microflows

By routing at the granularity of a traffic trunk rather than individual microflow, traffic trunk LSPs decouple the amount of forwarding state and control traffic needed to establish and maintain this state from the volume of traffic that flows through the network. So as traffic between LERs increases, the amount of bandwidth required for the individual trunk increases but the amount of forwarding state does not increase.

Setting up and managing 64 E-LSPs between all LERs statically may be time consuming and a waste of resources. As well, it is impossible to know ahead of time how much bandwidth microflows will require from each E-LSP trunk, or even whether there is a need to set up an E-LSP for a given QoS between two LERs.

Mobile agents can be used to dynamically setup one or more E-LSP trunks (each with a given QoS) between two LERs once a route to a given destination is learned. As shown

in Figure 30, mobile agents can be used to call the RESV Interface on the MPLS agent of a node in the network. This can be further extended so that at each node, the mobile agent requests the MPLS agent to bind the label to a particular QoS. Thus the QoS treatment of a packet is known at the time of the LSP establishment.

As well, mobile agents can also be used to dynamically increase the amount of bandwidth required for a given E-LSP using the augmenting path algorithms discussed. Networks that would use mobile agents to setup MPLS LSPs would be very versatile. As discussed in Section 5.3.4 an instance of a mobile agent can use a certain algorithm to find a path through the network to set up an LSP with a certain QoS. Another instance can use a different algorithm and can set up an LSP with a different QoS. If a provider wants to provision an LSP with a new path search mechanism, introducing a new mobile agent to the network would avoid the need to upgrade every node in the network. If mobile agents are not used, every node in the network would need to be aware of the new path search criteria to avoid loops.

6. AGENT SECURITY ISSUES

Security in any network is a concern in all networks, regardless of whether they are active or not. In traditional data networks, user data or control data can be used to attack one or more nodes in the network. Denial of Service (DoS) attacks are becoming more common today. With the introduction of agent based routing and network optimization, there is a concern that a "rogue" agent could contain malicious code that would affect one of more active nodes in the network. In an open network such as the Internet or a Service Provider network that is used to provide VPN services and Internet access to customers, network equipment is exposed to the risk of penetration by malicious agents.

Security is a key part of this mobile agent based achitecture (as opposed to being an afterthought).

The Mobile Agent and Router Security features are categorized as follows:

- Agent Privacy and Integrity
- Agent Authentication
- Access Control
- Resource Time Limitation

6.1 Agent Privacy and Integrity

As discussed in Section 5.3.2, mobile agents are transferred from one active network node to another within a capsule. The capsule contains the code of the mobile agent and

data (Section 5.3.5). The agent's state needs to be updated during its journey through the network. However, the agent's code must not be modified.

The data is sensitive and needs to be kept a secret to avoid eavesdroppers from getting the information. Crypographyic mechanisms [46] provide a secure communication facility that the agent uses to transport data across untrusted networks. Encrypting the data with public keys allows access of the data by the network routers.

A security breach could result in the modification of the agent's code as it traverses the network. It is impossible to prevent such modifications but it is possible to detect them [45]. Mechanisms such as *seals* or *message digests* ensure the integrity of the agent by detecting tampered code.

6.2 Agent Authentication

An agent must be authenticated before it is allowed to execute on the router's execution environment. Conversely, an agent needs to confirm the identity of the router before it reveals any sensitive data. Agents must not carry keys for authentication purposes since these leaves them vulnerable to malicious hosts. Digital Signature systems provide a mutual authentication scheme [46]. In order to verify signatures, the agents and routers need to reliably know the signing entity's public key. Certified Public keys are posted on network-wide directories that can be accessed by agents and routers.

6.3 Access Control

The router needs to protect its resources from unauthorized access. Access Control involves specifying policies for granting access to resources based on agent roles. For example, from Section 5.3.5 we see that RBA has access to the router's Label Database while RAA does not. Similarly RAA can read and write Path Database entries whereas RBA can only read them. RBA can read and write LSP Flow Database entries whereas RAA can only read them. As well, to access or modify specific states or data of the active node, the mobile agent has specific interfaces provided by the execution environment (as discussed in Section 5.3.3). This adds an extra layer of security since the agent must adhere to the specified interface and can only access or modify specific data. Boundary conditions are checked before accessing or modifing any data on the active node.

6.4 Resource Time Limitation

As agents traverse the networks they consume resources such as CPU cycles, disk space, etc. on each router. During "Denial of Service" attacks agents may acquire system resources and never release them, thereby preventing other agents from accessing them. A time limit mechanism is put into place so that the length of time an agent can access resources is limited. Similarly, a malicious host may repeatedly transmit an agent to the router to try to prevent other agents from accessing the router's resources. The router must ensure that it discards such agents immediately.

7. COMPARING A-EK-RSVP AND M-EK-RSVP

Figure 32 below shows an example network with 4 nodes. There are 3 LSPs that originate from node *s* and terminate at node *t*. Each of these LSPs provides connectivity to a customer (for example LSP 1 would provide connectivity between two remote sites of customer 1). The sites are connected to the provider's network at a port on node *s* and a port on node *t*. The capacity of each link, and the amount of bandwidth used is shown in brackets. For illustration purposes each of these LSPs use one unit of bandwidth (in a real network setting it can be up to the link capacity). Packets destined from node *s* to node *t* may take one of the three LSPs based on the QoS the packet requires. For example if a packet were to take LSP 3, it would travel from node *s* to node *x* over link A with label 17, from node *x* to node *y* over link C with label 18, and from node *y* to node *t* over link D with label 20, and finally out a port to customer 3.

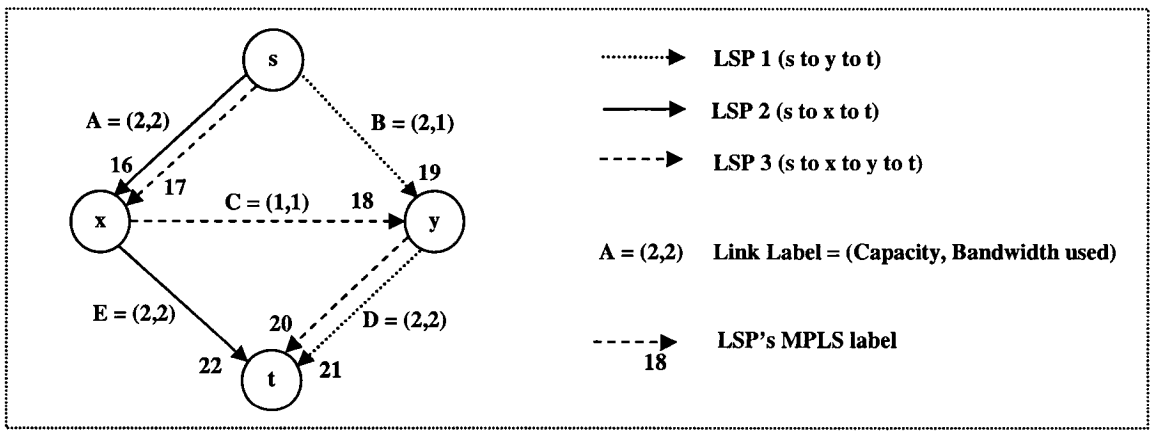

Figure 32. LSPs in a network

Suppose we want to create a new LSP (LSP4) on one unit of bandwidth for customer 4. The LSP is required to connect an incoming port on node *s* to an outgoing port on node *t*. From Figure 32 we see that this would require augmenting the path through the network. Notice that augmenting the path to create LSP 4 in the network requires the update of LSP 3, but leaves LSP 1 and LSP 2 untouched because the flow could not be augmented over certain links. Figure 33 below shows the same network after LSP4 has been created and LSP 3 has been updated. From the figure we see that a packet from Customer 3 traveling through LSP 3 would travel from node *s* to node *x* over link A with label 17, and from node *x* to node *t* over link E with label 20. A packet from Customer 4 traveling through LSP4 would travel from node s to node *y* over link B with label 23, and from node *y* to node *t* with label 24.

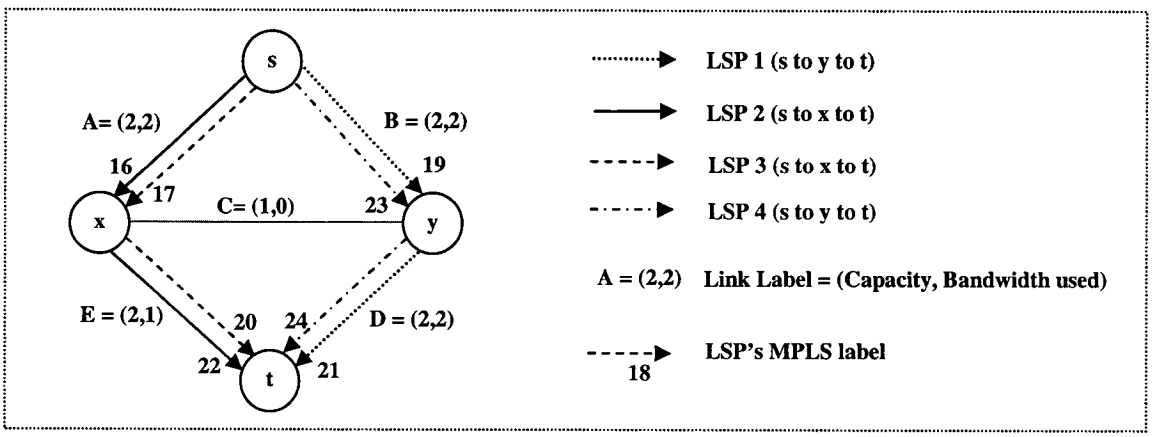

Figure 33. Augmenting LSPs in a network

7.1 Using a message based algorithm (M-EK-RSVP)

RFC 3209 **[53]** describes the message based method that is used to reroute LSPs in existing RSVP based MPLS networks. This section describes how a Message based Edmonds Karp RSVP algorithm (M-EK_RSVP) would be implemented. First we would need to find an augmenting path from the source node *s* to destination node *t*. This can be achieved by sending a message based implementation of Routine A from node *s* to node *t* (through nodes *y* and *x*), and a message based implementation of Routine B from node *t* to node *s* (through nodes *x* and *y*).

Once node *s* has determined that it has to reroute LSP 3 before LSP 4 can find available network resources, it sends a "PATH TEAR" message **[53]** from node *s* through nodes *x* and *y* to node *t*. When node *t* receives the "PATH TEAR" message, it sends a "RESV TEAR" message through nodes *y* and *x* to node *s*.

When node *s* receives the "RESV TEAR" message, it can now set up LSP 3 along a new path, and set up the new LSP 4. To setup LSP 3, node *s* sends a "PATH" message through node *x* to node *t*. Once node *t* receives the "PATH" message it sends a "RESV" message through node *x* to node *s*. To setup LSP 4, node *s* sends a "PATH" message through node *y* to node *t*. Once node *t* receives the "PATH" message it sends a "RESV" message through node *y* to node *s*.

7.2 Using the proposed algorithm with mobile agents (A-EK-RSVP)

This section describes an example using the A-EK-RSVP algorithm. To accomplish the search for a path for LSP4, an RAA agent is instantiated at node *s*. At node *s* the RAA creates a new Path entry in the Path Database ('LSP4 and incoming port customer 4'). It then requests node *s* for a link that still has some available capacity (forward arc) or a link that has consumed capacity by an incoming LSP (reverse arc). Since Link B (forward arc) has 1 unit of capacity still available the RAA now travels from node *s* to node *y*.

At node *y* the RAA requests the node for a forward or reverse arc. At this node there are no forward arcs available (no links with available capacity). However, Link C has an incoming LSP (LSP3) of 1 unit (a reverse arc). At node *y*, RAA creates a Path entry in the Path Database ('LSP4 incoming port Link B'). RAA 'remembers' **node *y*** as the start point for updating LSP3 (also known as 'Start Update Point' or 'SUP'), and then travels over Link C to get from node *y* to node *x*.

At node *x* the RAA requests the node for a forward or reverse arc. There is capacity on link E (forward arc) of 1 unit. At node *x*, it creates a Path entry for LSP4 in the Path Database ('LSP 4 incoming port Link C'), and then travels from node *x* to node *t* over link E. At node *t*, it creates a Path entry in the Path Database (LSP4 incoming port Link E'). As well, **node *t*** is the next point (after **node *y***) that RAA found an entry for LSP3 in the node's database (also known as 'End Update Point' or 'EUP'). Therefore RAA must also 'remember' **node *t*** *(as the ending point for updating LSP3).*

In order to create LSP4 and update LSP3, two RBA agents, RBA_4 and RBA_3 are spawned at node t. RBA_4 creates LSP4 while RBA_3 updates LSP3.

Recall that in section 5.3.3 we discussed how the RBA gets from the destination node back to the source node by using the *Path Database* information that was inserted at every node along the path by the RAA. The trick used here is to tell the RBA_4 to travel back towards the source using the LSP3's Path Database Entries on every node on the reverse path **node *t*** until it reaches **node *y***. On every node on the reverse path (until **node *y***), RBA_4 creates a new LSP Flow Database entry for LSP4 using LSP3's LSP Flow database entry. As well, on every node on the reverse path (except for **node *t***) RBA_4 removes LSP3's LSP Flow Database Entry. At **node *y*** it uses the LSP4's Path Database entries to continue its journey back to the source node *s*.

In the example above, RBA_4 requests a new label (24) at node *t*. It creates a new entry ("Incoming Link D, Incoming Label 24") in the *Classifier* and a corresponding *Scheduler* entry ("Outgoing port customer 4, pop label") for LSP 4. Using LSP3's LSP Flow Database Entry, RBA_4 carries label 24 and travels from node *t* to node *y* through link D. At node *y* RBA_4 requests a new label (23) and uses LSP4's Path Database Entry ('LSP4 Incoming port link B') to create the *Classifier* entry "incoming link B, incoming label 23". It also creates the *Scheduler* entry "outgoing link D, swap label 23 with label 24". It then carries label 23 and travels through link B to source node *s*. Using LSP4's Path Database entry it creates a *Classifier* entry "incoming port customer 4, add label 23" and a *Scheduler* entry "keep label 23, outgoing link B".

On the other hand, RBA_3 is told to use the LSP4's Path database entries on every node on the reverse path from **node *t*** until **node *y***. At node t, it updates LSP3's flow database entry (using but LSP4's incoming link in the Path Database entry). Along the way it looks for LSP3's flow database entry. If one is not found it creates an entry for LSP3 (using LSP3's flow id but LSP4's Path Database entry). On the very first node where it finds the LSP3's Flow database entry, the entry is updated. After this node it removes LSP3 flow database entries and LSP3 *Classifier* and *Scheduler* entries from every node along the path until it reaches **node *y***. At **node *y*** it terminates.

In the above example, RBA_3 changes the label mapping for LSP3 in the *Classifier* entry to "Incoming Link E, incoming label 20" at node *t*. The node's *Scheduler* entry for LSP 3 remains unchanged (since the scheduler would take a packet with label 20, pop the label and send it to the port for customer 3). RBA_3 carries label 20 and travels from node *t* to node *x*.

At node *x*, it updates the *classifier* entry for LSP3 so that data packets will be forwarded through the switch fabric towards link E instead of link C. It removes LSP3's previous *scheduler* entry "outgoing link C, swap label 17 with out going label 18" and creates a new *scheduler* entry "outgoing link to E, swap label 17 with outgoing label 20". Using the Path Database entry for LSP4 it then travels from node *x* to node *y* where it removes the classifier and scheduler entries for LSP3 and terminates.

Table 13 below summarizes the number of hops required by the agents to accomplish the creation of LSP4 and the reroute of LSP 3.

7.2.1 Path and LSP Flow Database entries

The tables below summarize the Path and LSP Flow Database entries at each node along the path of RAA and RBA agents.

RAA:

RAA at node *s*:

Node	LSP3 Flow Database Entries		Path Database entries
	LSP3 incoming link/port	LSP4 incoming link/port	LSP4 incoming link/port
S	port 3	n/a	**port4**
X	Link A	n/a	n/a
Y	Link C	n/a	n/a
T	Link D	n/a	n/a

RAA Table

LSP	Start Update Point	End Update Point

RAA at node *y*:

Node	LSP3 Flow Database Entries		Path Database entries
	LSP3 incoming link/port	LSP4 incoming link/port	LSP4 incoming link/port
S	port 3	n/a	port4
X	Link A	n/a	n/a
Y	Link C	n/a	**Link B**
T	Link D	n/a	n/a

RAA Table

LSP	Start Update Point	End Update Point
LSP3	*y*	

RAA at node *x*:

'LSP3' and 'LSP4' LSP database information (incoming link) and 'LSP4' Pheromone.

Node	LSP3 Flow Database Entries		Path Database entries
	LSP3 incoming link/port	LSP4 incoming link/port	LSP4 incoming link/port
S	port 3	n/a	port 4
X	Link A	n/a	**Link C**
Y	Link C	n/a	Link B
T	Link D	n/a	n/a

RAA Table

LSP	Start Update Point	End Update Point
LSP3	*y*	

RAA at node *t*:

Node	LSP3 Flow Database Entries		Path Database entries
	LSP3 incoming link/port	LSP4 incoming link/port	LSP4 incoming link/port
S	port 3	n/a	port4
X	Link A	n/a	Link B
Y	Link C	n/a	Link C
T	Link D	n/a	**Link E**

RAA Table

LSP	Start Update Point	End Update Point
LSP3	*y*	*t*

RBA_4:

RBA_4 at node *t*:

Node	**LSP3 Flow Database Entries**		**Path Database entries**
	LSP3 incoming link/port	**LSP4 incoming link/port**	**LSP4 incoming link/port**
S	port 3	n/a	port4
X	Link A	n/a	Link B
Y	Link C	n/a	Link C
T	Link D	**Link D**	Link E

RBA_4 at node *y*:

Node	**LSP3 Flow Database Entries**		**Path Database entries**
	LSP3 incoming link/port	**LSP4 incoming link/port**	**LSP4 incoming link/port**
S	port 3	n/a	port4
X	Link A	n/a	Link B
Y	Link C	**Link B**	Link C
T	Link D	Link D	Link E

RBA_4 at node *s*:

Node	**LSP3 Flow Database Entries**		**Path Database entries**
	LSP3 incoming link/port	**LSP4 incoming link/port**	**LSP4 incoming link/port**
S	port 3	**port 4**	port4
X	Link A	n/a	Link B
Y	Link C	Link B	Link C
T	Link D	Link D	Link E

RBA_3:

RBA_3 at node *t*:

Node	LSP3 Flow Database Entries		Path Database entries
	LSP3 incoming link/port	**LSP4 incoming link/port**	**LSP4 incoming link/port**
s	port 3	port 4	port4
x	Link A	n/a	Link B
y	Link C	Link B	Link C
t	~~**Link D**~~ **Link E**	Link D	Link E

RBA_3 at node *x*:

Node	LSP3 Flow Database Entries		Path Database entries
	LSP3 incoming link/port	**LSP4 incoming link/port**	**LSP4 incoming link/port**
s	port 3	port 4	port4
x	**Link A**	n/a	Link B
y	Link C	Link B	Link C
t	~~Link D~~ LinkE	Link D	Link E

RBA_3 at node *y*:

Node	LSP3 Flow Database Entries		Path Database entries
	LSP3 incoming link/port	**LSP4 incoming link/port**	**LSP4 incoming link/port**
s	port 3	port 4	port4
x	Link A	n/a	Link C
y	~~**Link C**~~ **n/a**	Link B	Link B
t	~~Link D~~ Link E	Link D	Link E

7.2.2 Final LSP Flow database

The following tables summarize the label databases for each of the nodes before LSP4 was created.

LSP Name	Incoming Link	Incoming Label	Outgoing link	Outgoing label	Bandwidth
LSP1	Customer 1	N/A	B	19	1
LSP2	Customer 2	N/A	A	16	1
LSP3	Customer 3	N/A	A	17	1

Table 3: Node s LSP Database

LSP Name	Incoming Link	Incoming Label	Outgoing link	Outgoing label	Bandwidth
LSP2	A	16	E	22	1
LSP3	A	17	C	18	1

Table 4: Node x LSP Database

LSP Name	Incoming Link	Incoming Label	Outgoing link	Outgoing label	Bandwidth
LSP1	B	19	D	21	1
LSP3	C	18	D	22	1

Table 5: Node y LSP Database

LSP Name	Incoming Link	Incoming Label	Outgoing link	Outgoing label	Bandwidth
LSP1	D	21	Customer 1	N/A	1
LSP2	E	22	Customer 2	N/A	1
LSP3	D	20	Customer 3	N/A	1

Table 6: Node t LSP Database

The following tables summarize the label databases for each of the nodes after LSP4 is created and LSP3 has been updated.

LSP Name	Incoming Link	Incoming Label	Outgoing link	Outgoing label	Bandwidth
LSP1	Customer 1	N/A	B	19	1
LSP2	Customer 2	N/A	A	16	1
LSP3	Customer 3	N/A	A	17	1
LSP4	Customer 4	N/A	B	23	1

Table 7: Updated Node s LSP Database

LSP Name	Incoming Link	Incoming Label	Outgoing link	Outgoing label	Bandwidth
LSP2	A	16	E	22	1
LSP3	A	17	E	20	1

Table 8: Updated Node x LSP Database

LSP Name	Incoming Link	Incoming Label	Outgoing link	Outgoing label	Bandwidth
LSP1	B	19	D	21	1
LSP4	B	22	D	24	1

Table 9: Updated Node y LSP Database

LSP Name	Incoming Link	Incoming Label	Outgoing link	Outgoing label	Bandwidth
LSP1	D	21	Customer 1	N/A	1
LSP2	E	22	Customer 2	N/A	1
LSP3	E	20	Customer 3	N/A	1
LSP4	D	24	Customer 4	N/A	1

Table 10: Updated Node t LSP Database

7.2.3 Time required by M-EK-RSVP

Table 11 below summarizes the number of hops required by the messages to accomplish the creation of LSP4 and the reroute of LSP 3.

Message Description	Number of hops
Path Augmentation (Routine A): "PATH" with "Record Route" option.	3
Path Augmentation (Routine B): "RESV" with "Record Route" option.	3
LSP 3 "PATH TEAR"	3
LSP 3 "RESV TEAR"	3
LSP 3 "PATH" with "Explicit Path" option.	2
LSP 3 "RESV"	2
LSP 4 "PATH" with "Explicit Path"	2
LSP 4 "RESV"	2
Total number of hops	**20**

Table 11: Summary of hops traversed by messages using M-EK-RSVP

The size of an ordinary IPv4 PATH message is 40 bytes (64 bytes for IPv6) [20],[47]. The time it takes to transfer and process an ordinary IPv4 PATH message between two Cisco 4700 routers is 2.00 ms [57]. The size of an ordinary IPv4 RESV message is 48 bytes (72 bytes for IPv6) [20],[47]. The time it takes to transfer and process an ordinary RESV message between two Cisco 4700 routers is 3.07 ms [57]. The Path augmentation (Routine A) needs to record the routers (from the source to the destination) by using the "Record Route" option [47] in the PATH message. The PATH message would also need

to record the LSPs that need to be updates (LSP Id, starting point of update, ending point of update). The Path Augmentation (Routine B) needs to carry the recorded routers in the "Record Route" option [47] of the RESV message. Each IPv4 router recorded in the "Record route" option takes an additional 12 bytes (24 bytes for each IPv6 router) [47]. The IPv4 PATH TEAR message is 32 bytes (56 bytes for IPv6) [20] while the IPv4 RESV TEAR message is 40 bytes (64 bytes for IPv6).

To set up an explicit path for LSP3 and LSP4, a record of the routers (from the source to the destination) is carried in the "Explicit Path" option of the PATH message. Each IPv4 router in the "Explicit Path" option takes an additional 12 bytes (24 bytes for each IPv6 router) [47]. The following table summarizes the time required by the messages to accomplish the creation of LSP4 and the reroute of LSP 3.

Message Description	Time (ms) for IPv4	Time (ms) for IPv6
Path Augmentation (Routine A): "PATH" with "Record Route" option.	9.60	10.50
Path Augmentation (Routine B): "RESV" with "Record Route" option.	16.12	27.63
LSP 3 "PATH TEAR"	4.80	8.40
LSP 3 "RESV TEAR"	7.67	12.30
LSP 3 "PATH" with "Explicit Path" option.	5.80	10.00
LSP 3 "RESV"	6.14	9.22
LSP 4 "PATH" with "Explicit Path" option.	5.80	10.00
LSP 4 "RESV"	6.14	9.22
Total Time (ms)	**62.07**	**97.27**

Table 12: Summary of time using M-EK-RSVP

The time required to create and update LSPs increases as the number of hops increases and number of LSPs to update increases. However, for each additional hop, the increase is not linear (since the message size increases significantly for every hop).

7.2.4 Time required by A-EK-RSVP

Agent Description	Number of hops
RAA	3
RBA_3	2
RBA_4	2
Total number of hops	**7**

Table 13: Summary of hops traversed by agents using A-EK-RSVP

Since testing was done using simulations, estimates for the overheads for creation, serialization, transfer, and restarting of agent had to be taken from existing literature [58],[59]. The following table shows the total (two ways) response time and size of a Mobile Agents over Java Remote Method Invocation (RMI) [59]:

No. of "Double Numbers" carried by Mobile Agent	Size of Mobile Agent (bytes)	Round trip time (ms)
25	1207	18.28
50	1572	25.25
75	1897	33.02
100	2299	40.50

Table 14: Size and two way response time of agents

Taking the worst case, the one-way cost of serialization and the RMI between a pair of routers is assumed to be 20.25 ms. This assumption is also made by the authors in [58] where a mobile agent implementation using object serialization and Java RMI between two machines in an ATM network is assumed to be between 20 and 27 ms (measuring the round-trip time and dividing it by two). The total time required to select one server, duplicate the agent task for the server, initialize data, Java RMI, serialize agents, transfer it to the server is 29.89ms [59]. The RMI and object serialization dominates the cost [59]. The cost of agent duplication, data initialization is approximately 29.89ms – 20.25 ms, or approximately 9.64 ms.

Given that the Mobile Agent only needs to store a list of LSPs to update (rather than a record of each router hop), the size of data carried by the agent is much less than the size of the data carried by the message implementation (Section 7.1). Therefore an assumption is made that the transfer time between routers will be approximately the same along every hop agent (using the transfer time for the agent with the largest data size from Table 14).

The following table summarizes the time required by the mobile agents to accomplish the creation of LSP4 and the reroute of LSP 3:

Agent Description	Time (ms)
RAA	70.39
RBA_3	50.14
RBA_4	50.14
Total Time (ms)	**170.67**

Table 15: Summary of time using A-EK-RSVP

7.2.5 M-EK-RSVP vs. A-EK-RSVP performance

7.2.5.1 Increasing number of hops

For the specific network and number of LSPs, the M-EK-RSVP completes creating LSP4 and updating LSP3 faster than the A-EK-RSVP algorithm. Figure 34 shows a similar network with a new node *i* that is an additional 'h' hops away from node *t*. Increasing the number of hops between the source node *s* and nodes *x* and *y* would only require RAA_4 and RBA_4 to take additional hops. Increasing the number of hops between destination node *t* and nodes *x* and *y* would allow us to compare the M-EK-RSVP performance with the worst case A-EK-RSVP performance since RAA_4, RBA_4 and RBA_3 would need to travel further.

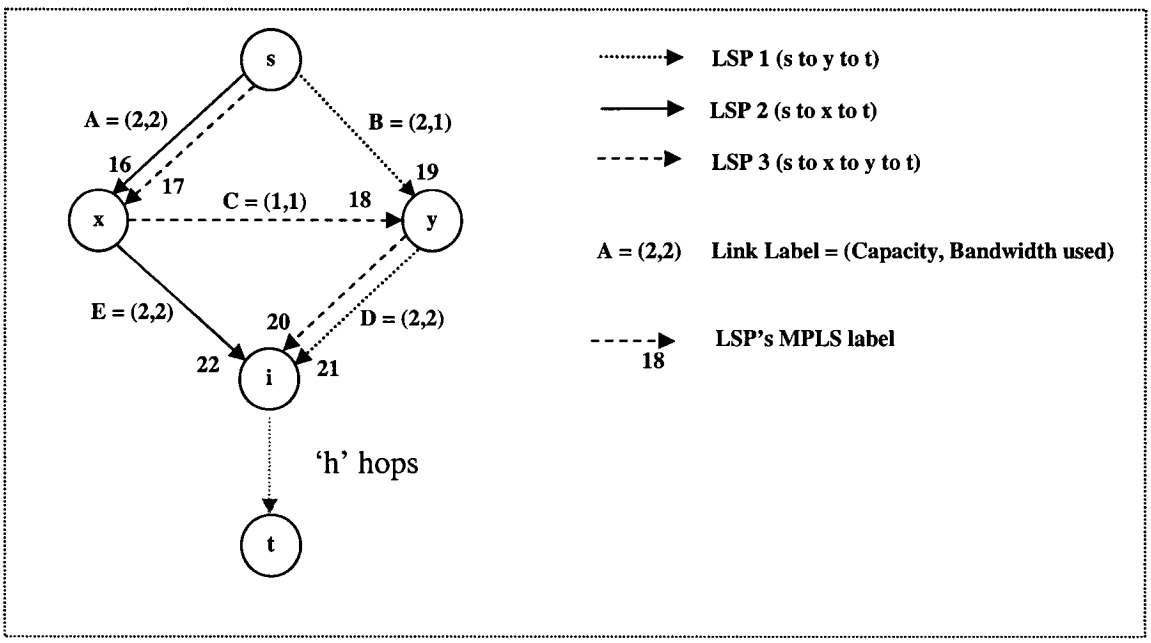

Figure 34. LSPs in a network

Table 16 below shows the number of message hops, agent hops and the time taken to create LSP4 and update LSP3 for various values of 'h'. From the values in the table we see that the A-EK-RSVP performs better than M-EK-RSVP when the number of additional hops in the network is greater than or equal to 22.

Number of additional hops 'h'	Total M-EK-RSVP message hops	Total A-EK-RSVP agent hops	Total M-EK-RSVP time (ms)	Total A-EK-RSVP time (ms)
0	20	7	62.08	161.03
1	28	10	92.82	221.78
2	36	13	126.90	282.53
5	60	22	249.14	464.78
10	100	37	519.58	768.53
20	180	67	1310.58	1376.03
21	188	70	1408.02	1436.78
22	**196**	**73**	**1508.80**	**1497.53**
25	220	82	1831.14	1679.78
50	420	157	5684.58	3198.53
100	820	307	19644.58	6236.03

Table 16: Execution times for different number of hops

Figure 35 shows the graph for the results above.

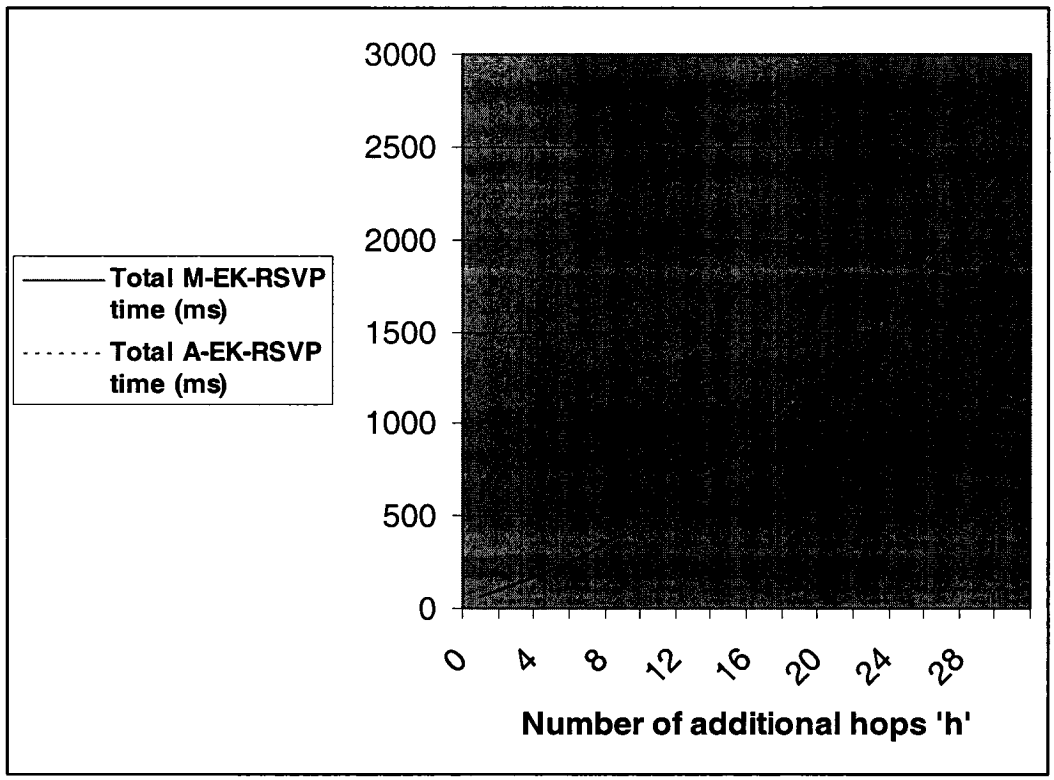

Figure 35. M-EK-RSVP vs. A-EK-RSVP

As seen in Table 17, as 'h' increases, the total number of message hops increases linearly. However, as discussed in Section 7.2.3, as the 'h' increases, the number of entries in the 'Record Route' options of the PATH and RESV messages increases. As well, the number of entries in the 'Explicit Path' option of the PATH message increases. A single additional router entry causes the sum of the sizes of these specific message types to increase polynomially. Therefore, the increase in number of hops causes a polynomial increase in the total time required to transmit messages using the M-EK-RSVP based algorithm. The polynomial function determined by using Microsoft Excel and the data from Table 16 is:

$$y = 6.67x^2 + 44.81x + 10.6$$

where $x = \left(\frac{h}{2} + 1\right)$ and y is the time in milliseconds. The 'goodness of fit' (also known as the coefficient of correlation or R^2) for this curve is 1. Figure 36 shows the 'goodness of fit' for the polynomial equation.

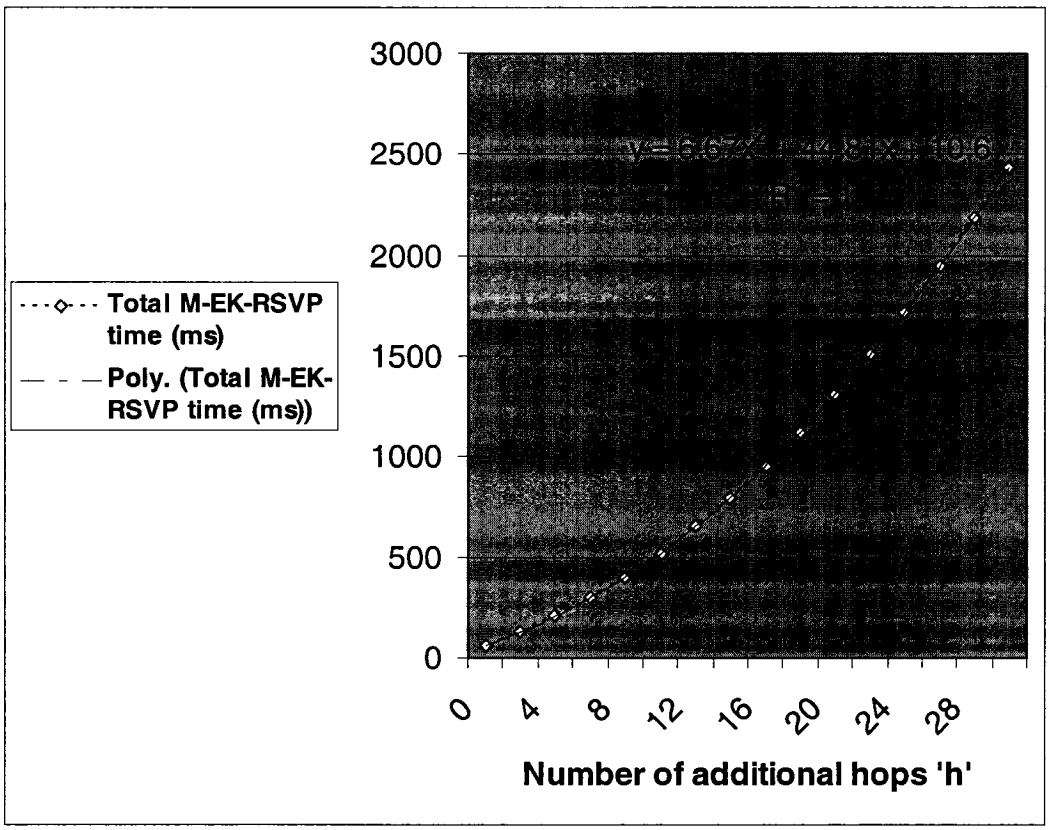

Figure 36. Goodness of fit for the polynomial equation

As discussed in Section 7.2.4, the RMI and object serialization dominates the overhead in the A-EK-RSVP based algorithm. When 'h' is small, this creates a significant overhead in the A-EK-RSVP based algorithm. As seen in Figure 35, as the 'h' increases, the A-EK-RSVP overhead increases linearly. At a certain point, the overhead required to

transmit the larger messages of the M-EK-RSVP based algorithm is greater than the total RMI and object serialization overhead in the A-EK-RSVP based algorithm.

7.2.5.2 Increasing number of LSPs to update

Using a value of 'h' equal to 20, Table 17 below shows the number of message hops, agent hops and the time taken to create LSP4 and update up to 'n' LSPs. For every LSP that needs to be updated, a new instance of RBA needs to be created. From the values in the table we see that the A-EK-RSVP performs better than M-EK-RSVP when the number of LSPs to be updated in the network is greater than or equal to 6 (when 'h' is equal to 20).

Number of LSPs to update 'n'	Total M-EK-RSVP message hops	Total A-EK-RSVP agent hops	Total M-EK-RSVP time (ms)	Total A-EK-RSVP time (ms)
1	180	67	1310.58	1376.03
2	210	87	1669.60	1790.67
3	240	105	2028.62	2164.81
4	270	121	2387.64	2498.45
5	300	135	2746.66	2791.59
6	**330**	**147**	**3105.68**	**3044.23**
7	360	157	3464.70	3256.37
8	390	165	3823.72	3428.01

Table 17: Execution times for different number of LSPs to update

Figure 37 shows the graph for the results above.

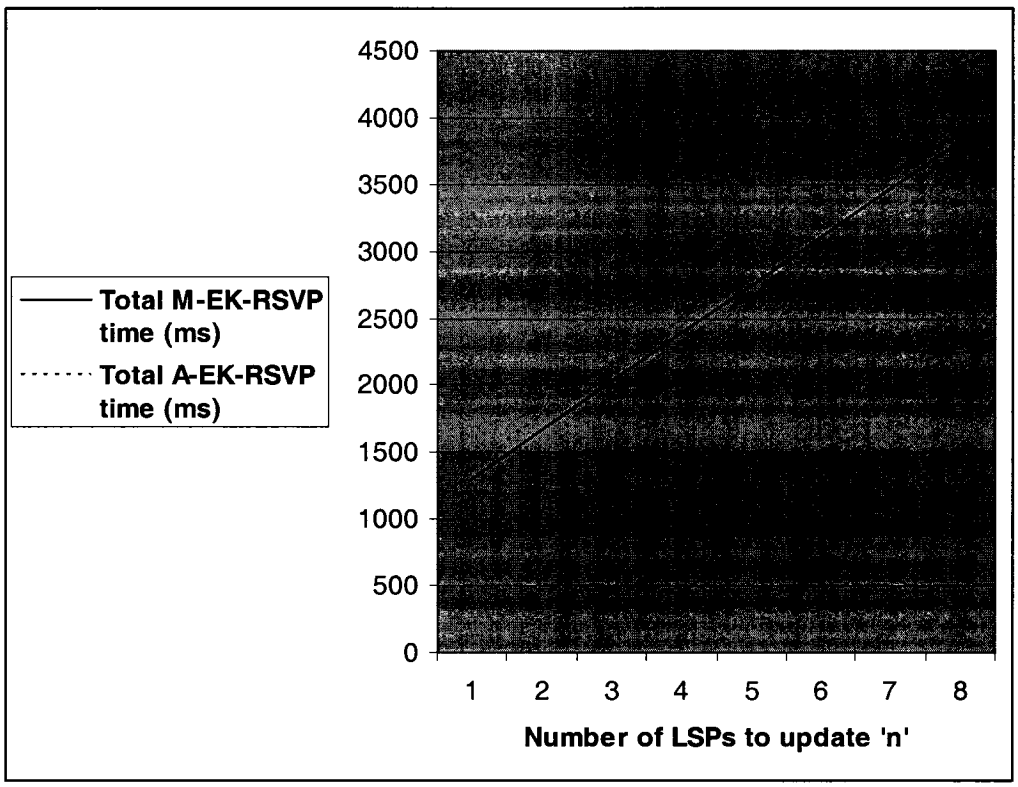

Figure 37. M-EK-RSVP vs. A-EK-RSVP

8. COMPARING A-EK-RSVP AND RSVP

Using networks of different sizes (varying number of nodes and edges), the following simulations were done:

a) A-EK-RSVP based simulations

   i) RAA selects the first available edge on the node.

   ii) RAA selects the link with the least cost.

   iii) RAA select the link with the most available bandwidth

b) RSVP message based simulations

   i) Node selects the link with least cost.

The sections summarize the simulation results for each network. The section shows how many LSPs were created by the A-EK-RSVP and RSVP algorithms. The total number of LSPs created and the time taken to create the LSPs are shown for the algorithms for the various networks. For the A-EK-RSVP algorithm, the number of agents created to search for a path, create and update LSPs and the total number of agent hops is shown. The sub-sections also show the paths taken by the agents (RAA and RBA agents). For the RSVP algorithm, the number of message hops is shown, and the paths taken by the PATH and RESV message is shown. Finally, a summary of all LSP paths created is shown. The paths are indicated by a list of node numbers.

The source code of the A-EK-RSVP simulation is provided in the APPENDIX.

8.1 Network 1

8.1.1 A-EK-RSVP

Agent Link Selection Method	Number of LSPs created	Number of Agents Created	Number of Agent Hops	Total time (ms)
First Available	13	29	144	3195.56
Least Cost	13	28	120	2699.92
Most Bandwidth	13	29	152	3367.19

8.1.2 RSVP

RSVP Link Selection Method	Number of LSPs created	Number of Message Hops	Total time (ms)
Least Cost	12	104	262.57

A-EK-RSVP created 1 more LSP than RSVP (8.3% improvement).

Network 2

8.1.3 A-EK-RSVP

Agent Link Selection Method	Number of LSPs created	Number of Agents Created	Number of Agent Hops	Total time (ms)
First Available	9	20	79	1792.55
Least Cost	9	20	79	1792.55
Most Bandwidth	9	19	66	1519.66

8.1.4 RSVP

RSVP Link Selection Method	Number of LSPs created	Number of Message Hops	Total time (ms)
Least Cost	6	49	123.68

A-EK-RSVP created 3 more LSP than RSVP (50% improvement).

8.2 Network 3

8.2.1 A-EK-RSVP

Agent Link Selection Method	Number of LSPs created	Number of Agents Created	Number of Agent Hops	Total time (ms)
First Available	14	29	84	1980.56
Least Cost	14	32	94	2211.98
Most Bandwidth	14	30	71	1726.95

8.2.2 RSVP

RSVP Link Selection Method	Number of LSPs created	Number of Message Hops	Total time (ms)
Least Cost	6	31	79.98

A-EK-RSVP created 8 more LSPs than RSVP (133% improvement).

8.3 Network 4

8.3.1 A-EK-RSVP

Agent Link Selection Method	Number of LSPs created	Number of Agents Created	Number of Agent Hops	Estimated total time (ms)
First Available	8	20	106	2339.30
Least Cost	8	20	88	1974.80
Most Bandwidth	8	18	84	1874.52

8.3.2 RSVP

RSVP Link Selection Method	Number of LSPs created	Number of Message Hops	Estimated total time
Least Cost	3	32	80.05

A-EK-RSVP created 5 more LSPs than RSVP (166.67% improvement).

8.4 Network 5

8.4.1 A-EK-RSVP

Agent Link Selection Method	Number of LSPs created	Number of Agents Created	Number of Agent Hops	Estimated total time
First Available	6	15	88	1926.60
Least Cost	6	15	88	1926.60
Most Bandwidth	6	13	68	1502.32

8.4.2 RSVP

RSVP Link Selection Method	Number of LSPs created	Number of Message Hops	Total time
Least Cost	3	31	78.05

A-EK-RSVP created 3 more LSPs than RSVP (100% improvement).

8.5 RSVP vs. A-EK-RSVP performance

From the simulation results above we see that the RSVP takes less time to create a single LSP than A-EK-RSVP. However, A-EK-RSVP was able to find more paths through the network than RSVP. As shown in the simulation results, the A-EK-RSVP algorithm improves network resource usage. As well, the simulations show that A-EK-RSVP can be used to find a path using different search criteria.

9. CONCLUSION

In this thesis, an agent (A-EK-RSVP) based approach is used to demonstrate how an augmenting path algorithm can be used to improve network resources. The proposed algorithm can be used by Service Providers to provision LSPs in MPLS networks. The results show that without the use of the proposed algorithm, the network bandwidth usage would be worse (Section 8). The A-EK-RSVP algorithm performs better than the message based algorithm (M-EK-RSVP) in larger networks and when larger number of LSPs need to be updated (Section 7).

Mobile Agents are well suited to implement the proposed algorithm. Intelligent mobile agents leave Path Database entries along the path between the source and destination nodes. Another set of mobile intelligent agents use the Path and Flow Database entries to travel back to the source node, updating the LSP Flow Databases to improve network capacity. Without intelligent mobile agents, static agents on the network nodes would require greater intelligence and multiple messages to coordinate optimization of network resources.

As well, a static agent would require every node to have highly complex and sophisticated software that would need to know how to use information such as user traffic requirements and the priority of the customer to provision LSPs on network links that have unique characteristics. The implementation of such a complicated static agent would still not take every link characteristic or traffic requirement into account. Updates to the static agent would require network devices to be taken temporarily offline.

Frequent revisions of the agent would lead to instability of the network elements and to the network as a whole. Mobile agents allow service providers to introduce new versions of the agent without affecting the network. This allows quicker and easier introduction of new services when customers demand it.

Links in MPLS networks have a finite number of capacities. To simplify managing and billing of network services, service providers sell and provision bandwidth in finite quantities. Using the Hitchcock Transportation method (see Section 2.5.2) allows the scaling of the algorithm that permits capacity optimization to be accomplished in polynomial time [8]. With the gain in popularity of MPLS based VPN services such as BGP MPLS VPNs and VPLS, the demand for network resources in MPLS backbones is on the rise. As well, with the high cost of such networks, providers are searching for ways to maximize return on investment by optimizing bandwidth usage in their networks. The algorithm proposed in this thesis not only provides a means to achieving this goal but also provides the added value of ensuring that higher paying customers are guaranteed a higher level of service. The use of mobile agents provides flexible levels of service and eases the management of such networks.

10. EVOLUTION

The method proposed in this thesis is generic and can be extended to other types of network protocols. Such networks use protocols like ATM, Frame Relay, TDM, SONET and others.

The method could also be extended to other problems that require improved usage of other resources. An example would be optimizing airline or train seat usage based on how much customers pay. The algorithm would favor higher paying customers by allowing them to reach their final destination via the shortest distance and the least number of stopovers. On the other hand, lower paying customers would travel longer distances and make more stopovers in order to reach their final destinations.

The algorithm proposed in the thesis uses the Edmonds Karp Algorithm. Newer Generalized Maximum Flow Algorithms could also be used in the future. As well, further studies need to be conducted to see if *Stigmergy* and 'pheromone' tables can be applied and used to improve the proposed algorithm.

REFERENCES

[1] J. M. Bradshaw, *Software Agents*, AAAI Press, Menlo Park, California, and The MIT Press, Cambridge, Massachusetts, 1997.

[2] A. Fugetta, G. P. Picco, G. Vigna, "Understanding Code mobility", in *IEEE Trans. Software Engineering*, Vol. 24, No. 5, pp. 342-361, 1998.

[3] Donald Goldfarb and Zhiying Jin, "Polynomial-Time Highest-Gain Augmenting Path Algorithms for the Generalized Circulation Problem", Sloan School Working paper 3909-96, June, 1996. A journal version of this paper appeared in *Mathematics of Operations Research*, Vol. 22, pp. 793-802, 1997.

[4] T. Radzik, "Faster algorithms for the generalized network flow problem", *Mathematics of Operations Research*, Vol. 23, pp. 69-100, 1998.

[5] K. D. Wayne, "Generalized Maximum Flow Algorithms", PhD thesis, Department of Operations Research and Industrial Engineering, Cornell University, 1999.

[6] G. B. Dantzig, "Applications of the simplex method to a transportation problem", in *Activity analysis and production and allocation,* Wiley, New York, pp. 359-373, 1951.

[7] L.R. Ford, Jr. and D.R. Fulkerson, "Maximal Flow Through a Network", in *Canadian Journal of Mathematics*, Vol. 8, pp. 399-404, 1956.

[8] L.R. Ford, Jr. and D.R. Fulkerson, "A Simple Algorithm for Finding Maximal Network Flows and an Application to the Hitchcock Problem", in Canadian Journal of Mathematics, Vol. 9, pp. 210-218, 1957.

[9] L.R. Ford, Jr. and D.R. Fulkerson, *Flows in Networks*, Princeton University Press, Princeton, New Jersey, 1962.

[10] J. Edmonds and R. M. Karp, "Theoretical improvements in algorithmic efficiency for network flow problems", in *Journal of the ACM*, Vol. 19, pp. 248-264, 1972.

[11] R. G. Busacker and P. J. Gowen, "A procedure for determining a family of minimum-cost network flow pattern", Technical Report 15, Operations Research Office, Johns Hopkins University, 1961.

[12] Di Caro, Dorigo, "AntNet: a mobile agents approach to adaptive routing", Technical Report, IRIDIA/97-12, Universit Libre de Bruxelles, Belgium, 1997.

[13] N. Minar, K.H. Kramer and P. Maes, "Cooperating Mobile Agents for Mapping Networks", in *Proceedings of the First Hungarian National Conference on Agent Based Computing*, Budapest, Hungary, 1998.

[14] N. Minar, K.H. Kramer and P. Maes, "Cooperating Mobile Agents for Dynamic Network Routing", in *Proceedings of the software Agents for future Communications Systems*, Springer-Verlag, New York, pp. 287-304, 1999.

[15] P. Stone, M. Veloso, "Multiagent Systems: A Survey from a Machine Learning Perspective", Technical Report CMU-CS-97-193, Carnegie Mellon University, Pittsburg, 1997.

[16] Tennenhouse el al., "A Survey of Active Network Research", in *IEEE Communications Magazine*, Vol. 35, No. 1, pp. 80-86, 1997.

[17] Jaeger et al., "An Active Network Services Architecture for Routers with Silicon-Based Forwarding Engines", Joint work done at University of Maryland and Nortel Networks, 1999.

[18] Peterson et al., "An OS Interface for Active Routers", in *IEEE Journal on Selected Areas in Communications*, Vol.19, No.3, 2001.

[19] L. Peterson, "NodeOS Interface Specification", AN Node OS Working Group Technical Report, 2001.

[20] Zhang et al., "RSVP: A new resource reservation protocol", *IEEE Network*, Vol. 7, No. 9, pp. 8–18, 1993.

[21] G. B. Dantzig,, "Application of the Simplex Method to a Transportation Problem", in *Activity Analysis and Production and Allocation*, Wiley, New York, pp. 359-373, 1951.

[22] J. Edmonds, R. M. Karp, "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems", in *Journal of the ACM*, Vol. 19, pp. 248-264, 1972.

[23] R. E. Goldberg, R.E. Tarjan, "A new Approach to the Maximum Flow Problem", in *Journal of the ACM*, Vol. 35, pp. 921-940, 1988.

[24] E. A. Dinic (Dinitz), "Algorithm for Solution of a problem of Maximum Flow in Networks with Estimation", in *Soviet Math. Dolk.,* Vol. 11, pp. 1277-1280, 1970.

[25] H. N. Gabow, "Scaling Algorithms for Network Problems", in *Journal of Computer and System Sciences,* Vol. 31, pp. 148-168, 1985.

[26] A. V. Karnzanov, "Determining the Maximum Flow in a Network by the Method of Preflows", in *Soviet Math. Dok.,* Vol. 15, pp. 434-437, 1974.

[27] D. D. Sleator, R. E. Tarjan, "A Data Structure for Dynamic Trees", in *Journal of Computer and System Sciences,* Vol. 26, pp. 362-391, 1983.

[28] A. V. Goldberg, R. E. Tarjan, "A new Approach to the Maximum Flow Problem", in *Journal of ACM.,* Vol. 35, pp. 921-940, 1988.

[29] A. V. Goldberg, S. Rao, "Length Flow for Flow Computations", Technical Report 97-055, NEC Research Institute, Princeton, New Jersey, 1997.

[30] K. D. Wayne, L. K. Fleischer, "Fast and Simple Approximation Schemes for Generalized Flow", in *Mathematical Programming*, Vol. 91, No. 2, pp. 215-238, 2002.

[31] K.D. Wayne, "A Polynomial Combinatorial Algorithm for Generalized Minimum Cost Flow", in *Proceedings of the thirty-first annual ACM symposium on Theory of computing*, pp. 11-18, 1999. Also appeared in *Mathematics of Operations Research*, Vol. 27, pp. 445-449, 2002.

[32] T. Radzik, "Approximate generalized circulation", Technical Report 93-2, Cornell Computational Optimization Project, Cornell University, 1993.

[33] T. Radzik, "Faster algorithms for the generalized network flow problem", in *Mathematics of Operations Research*, Vol. 23, pp. 69-100, 1998.

[34] Adhichandra, I. et al. "Using Mobile Agents to improve performance of Network Management Operations", Computer Communications Research Group, School of Computing, Leeds Metropolitan University, 2003. Also appeared in *Proceedings of 4th Annual Symposium of Postgraduate Networking Conference* (PGNET'03), Liverpool, UK, 2003.

[35] P. Grassé, "La reconstruction du nid et les coordinations interindividuelles chez Bellicositermes natalenes et cubitermes. La théorie de la stigmergie: essai d'interprétation du comportement des termites constructeurs", *Insectes Sociaux*, Vol. 6, pp. 41-81, 1959.

[36] A. Kaizar, A. Mikler, "Dynamic Agent Population in Agent-Based Distance Vector Routing", in Second International Workshop on Intelligent Systems Design and Applications (ISDA2002), Atlanta, USA, pp. 195-200, 2002.

[37] E. Rosen, A. Viswanathan, R. Callon, "Multiprotocol Label Switching Architecture", *Internet Engineering Task Force (IETF) Standard RFC 3031*, 2001.

[38] M. Rubinstein, O. Duarte, "Evaluating Tradeoffs of Mobile Agents in Network Management, Universidade", in *Networking and Information System Journal*, Vol. 2, No. 2, pp. 237-252, 1999.

[39] R. Schoonderwoerd, O. Holland, J. Bruten, L. Rothkrantz, "Ant-based load balancing in telecommunication networks", in *Adaptive Behavior*, Vol. 5, No. 2, pp. 169-207, 1996.

[40] A. Bieszczad, B. Pagurek, T. White, "Mobile Agents for Network Management", in *IEEE Communications Surveys*, Vol. 1, No.1, pp. 2-9, 1998.

[41] R. R. Choudhary, S. Bhandhopadhyay, K. Paul, "A Distributed Mechanism for Topology Discovery in Ad Hoc Wireless Networks Using Mobile Agents", in *Proceedings of the 1st ACM international symposium on Mobile ad hoc networking & computing*, pp. 145-146, 2000.

[42] S. Marwaha, C. K. Tham, D. Shrinivasan, "Mobile Agents Based Routing Protocols for Mobile Ad Hoc Networks", in *Proceedings of IEEE Global Telecommunications Conference* (GLOBECOM'02), Taipei, Taiwan, 2002.

[43] R. E. Tarjan, A. V. GoldBerg, "Expected Performance of Dijkstra's Shortest Path Algorithm", Technical Report 96-062, NEC Research Institute, Princeton, NJ, 1996.

[44] V. Priebe, "Average-Case Complexity of Shortest-Paths Problems", PhD Thesis, Universität des Saarlandes, 2000.

[45] W. M. Farmer, J. Guttman, V. Swarup, "Security For Mobile Agents: Issues and Requirements", in *Proceedings of 19th National Information Systems Security Conference*, pp. 591-597, October 1996.

[46] B. Schneier, *Applied Cryptography*, John Wiley, 2nd Edition, New York, 1996.

[47] D. Awduche, "MPLS and Traffic Engineering in IP Networks", in *IEEE Communications Magazine*, pp. 42-47, December 1999.

[48] C. L. Hendrick, "Routing Information Protocol", *Internet Engineering Task Force (IETF) Standard RFC 1058*, 1988.

[49] J. P. Bigus, J. Bigus, *Constructing Intelligent Agents Using Java*, 2nd Edition, Professional Developer's Guide Series, Wiley Computer Publishing, New York, 2001.

[50] D. Lange, "Mobile agents: Environments, Technologies, and Applications", in *Proceedings of the Third International Conference on the Practical Applications of Intelligent Agents and Multi-Agent Technology*, pp. 11-14, 1998.

[51] B. Quirk, T. Pissello, "How to Quantify Downtime", in *Network World*, January 5, 2004.

[52] D. Lange, M. Oshima, *Programming and Deploying Java Mobile Agents with Aglets*, Addison Wesley, Reading, Massachusetts, 1998.

[53] D. Awduche, D. Gan, T. Li, V. Srinivasan,, G. Swallow, "RSVP-TE: Extensions to RSVP for LSP tunnels", *Internet Engineering Task Force (IETF) Standard RFC 3209*, 2001.

[54] T. Cormen, C. Leiserson, R. Rivest, *Introduction to Algorithms*, McGraw-Hill, New York, 1990.

[55] K. Nichols, S. Blake, F. Baker, D. Black, "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers", *Internet Engineering Task Force (IETF) Standard RFC 2474*, 1998.

[56] J. Postel et al., "Internet Protocol: DARPA Internet Program Protocol Specification", *Internet Engineering Task Force (IETF) Standard RFC 791*, 1981.

[57] A. Neogi, T. Chuieh, "Performance Analysis of an RSVP-Capable Router", in *IEEE Network*, Vol. 13, No. 5, pp. 56-63, 1999.

[58] B. Wims, C. Z. Xu, "A Mobile Agent Based Push Methodology for Global Parallel Computing", *Concurrency: Practice and Experience,* Vol. 12, No. 8, pp. 705-726, John Wiley & Sons, 2000. Also appeared in "TRAVELER: a mobile agent based infrastructure for wide area parallel computing", *Third International Symposium on Mobile Agents* pp. 258-259, 1999.

[59] C. Bohoris, G. Pavlou, H. Cruickshank, "Using mobile agents for network performance management", in *Network Operations and Management Symposium*, pp. 637-652, 2000.

APPENDIX A: SOURCE CODE DESCRIPTION

This section gives a brief description of the files, classes, attributes and methods used in the A-EK-RSVP simulation. I'd like to thank Dr. Weiss of Carleton University for sending me links to existing ant-based simulation examples. As well, I'd like to give credit to Dr. K. Ikeda of University of Tokushima. I used his MaxFlow algorithm as a starting point for the A-EK-RSVP simulation algorithm.

1) Ant.java : Contains the agent simulation for RAA and RBA. The RAA is created at the start of each LSP path search. Each LSP to be updated may require one or multiple LSP segments to be updated. Since the RBA is created for every LSP that needs to be updated, it is responsible for updating one or more LSP segments for the same LSP.

   a) UpdateLspSegment class: represents an LSP segment that needs to be updated. The RAA instantiates an object for every LSP segment that will need to be updated by the RBA.

      i) Attributes:

         (1) number - Stores the LSP number for the LSP segment to be updated.

         (2) supNode – Stores the "start update point" (SUP) of the LSP segment to be updated.

         (3) eupNode – Stores the "end update point" (EUP) of the LSP segment to be updated.

ii) Methods:

(1) UpdateLspSegment – class constructor. Requires the LSP number and the "start update point" node number.

b) Ant class: Used to simulate RAA and RBA.

i) Attributes:

(1) node – Stores the current node number on which the agent resides.

(2) edge – Stores the incoming edge number the agent used to get to this node.

(3) tour – Vector that stores the edges over which the agent has traveled (for debugging purposes).

(4) visited – Vector that stores the nodes the agent has visited (for debugging purposes).

(5) len – Number of hops the agent has traveled.

(6) lsp – Stores the LSP number for which the agent is responsible for.

(7) action – Used by RBA to determine the specific action that needs to be carried out for the LSP segment on a specific network edge.

(8) totalTime – Used to store the total (simulation) time the agent has been executing.

(9) lspSegmentsToUpdate – Stack that stores all the LSP segments to be updated.

ii) Methods:

(1) Ant – class constructor. Requires the node number for which the agent is being created on as well as the LSP number for the LSP the agent is responsible for.

(2) getTotalTime – returns the total (simulated) execution time.

(3) addEdge – used to add an edge to the agent tour (for debugging purposes), and increment the number of hops the agent has travelled.

(4) setCurrentNode – used to set the node the agent will be visiting, add the node to the Vector of nodes visited (for debugging purposes), and update the total (simulated) execution time.

(5) getTourLength – returns the number of hops the agent has traveled.

(6) getCurrentNodeNum – returns the current node the agent in on.

(7) getLspNum – returns the LSP number the agent is working on.

(8) addSUP – used by RAA to record the "start update point" (SUP) for an LSP segment. This creates and inserts a new LSP segment on the lspSegmentsToUpdate stack.

(9) addEUP - used by RAA to record the "end update point" (EUP) for an LSP segment. This updates the eupNode for LSP segment at the top of the lspSegmentsToUpdate stack.

(10) delEUP - Used by RAA to delete the last EUP. When RAA cannot find a forward or reverse edge towards the destination, it travels back to the previous node to search for other edges. RAA needs to remove the last EUP if it is no longer valid.

(11) delSUP - Used by RAA to delete the SUP. When RAA cannot find a forward or reverse edge towards the destination, it travels back to the previous node to search for other edges. RAA needs to remove the last SUP if it is no longer valid. When the SUP is removed, the top LSP segment in the lspSegmentsToUpdate is popped.

(12) numLspsToUpdate – returns the number of LSP segments to be updated.

(13) currentLspToUpdate – returns the current LSP number for the LSP segment to be updated. This happens to be the top LSP segment in the lspSegmentsToUpdate.

(14) lastSUP – returns the SUP for the LSP segment at the top of the lspSegmentsToUpdate.

(15) lastEUP – returns the EUP for the LSP segment at the top of the lspSegmentsToUpdate.

(16) getSUP – requires an 'index' to be passed in. Returns the SUP for the LSP segment at the 'index' position within the lspSegmentsToUpdate stack.

(17) getSUP – requires an 'index' to be passed in. Returns the EUP for the LSP segment at the 'index' position within the lspSegmentsToUpdate stack.

(18) getLspSegmentToUpdate - requires an 'index' to be passed in. Returns the LSP number for the LSP segment as the 'index' position within the lspSegmentsToUpdate stack.

(19) copyLspSegmentToUpdate – requires an 'index' and the agent to copy the LSP segment to be passed in. This is used by the RAA to copy an LSP segment to the RBA.

(20) setAction – sets the agent action (for RBA). The action is one of the following: 0 (do nothing), 1 (create a new LSP segment on the edge), 2 (clear an old LSP segment on the link), and 3 (replace the old LSP segment on the link with a new one).

2) InfoPanel.java: Used to display the output for the simulation.

a) InfoPanel class:

i) Attributes:

(1) ta – The text area of the information panel applet.

ii) Methods:

(1) init – used to initialize the information panel applet.

(2) updatePanel – used to display results for the simulation.

3) Maxflow.java: Used to display the network. It contains the network nodes and edges. As well, it contains the simulation for Routine A and Routine B for RAA and RBA respectively.

   a) Lsp class: Used to store an LSP instance in the node LSP database.

      i) Attributes:

         (1) inEdge – stores the incoming edge of the LSP. Invalid if the value is -1.

         (2) outEdge – stores the outgoing edge of the LSP. Invalid if the value is -1.

         (3) number – stores the LSP number for the LSP. Invalid if the value is -1. This is used to determine whether an LSP with this LSP number exists in the node database.

   b) Node class: Represents a network node.

      i) Attributes

         (1) x – x coordinate of node on network map.

         (2) y – y coordinate of node on network map.

         (3) delta_plus – Vector of all edges that start at this node.

         (4) delta_minus – Vector of all edges that terminate on this node.

         (5) dist – If the RAA has not visited this node then dist is -1, otherwise it is the distance from the source node. This is also used be RAA to determine whether it has visited this specific node.

         (6) prev – previous node on the path.

(7) p_edge – previous edge on the path.

(8) l – the flow-width or previous edge on the path.

(9) w – node drawing width.

(10) h – node drawing height.

(11) number – node number (unique for the specific network).

(12) name – node name.

(13) lspDatabase – Array used to store the LSPs created on the node. The lspDatabase is indexed by LSP number.

ii) Methods:

(1) visited – returns true if the agent has visited the node.

(2) getMaxLsps – returns the maximum number of LSPs that can be created in the node's LSP database.

(3) getLspInEdge – requires an LSP number to be passed in. Returns the incoming edge for the specified LSP.

(4) getLspOutEdge – requires an LSP number to be passed in. Returns the outgoing edge for the specified LSP.

(5) clearLsp – requires an LSP number to be passed in. Deletes the specified LSP from the node's LSP database.

(6) addLsp - requires an LSP number, incoming edge and outgoing edge to be passed in. Creates the specified LSP in the node's LSP database.

(7) updateLsp - requires an LSP number, incoming edge and outgoing edge to be passed in. Updates the incoming and/or outgoing edges of the specified LSP in the node's LSP database.

(8) hasLsp – requires an LSP number to be passed in. Return true if the specified LSP exists in the node's LSP database.

(9) hasLspOnInEdge - requires an LSP number and incoming edge to be passed in. Returns true if the specified LSP exists on the specified incoming edge in the node's LSP database.

(10) hasLspOnOnEdges - requires an LSP number, incoming edge and outgoing edge to be passed in. Returns true if the specified LSP exists on the specified incoming and outgoing edges in the node's LSP database.

(11) getLspOnInEdge - requires incoming edge to be passed in. Returns the LSP number of the first LSP found that matches this incoming edge in the node's LSP database.

c) Edge class: Represents a network edge.

i) Attributes:

(1) rndd_plus - initial vertex (node) of this edge.

(2) rndd_minus - terminal vertex (node) of this edge.

(3) capacity - edge capacity.

(4) flow - existing edge flow-width.

(5) cost - edge cost.

(6) st - used to color the s-t path.

(7) number - edge number (unique for the specific network).

(8) name – edge name.

d) Maxflow class:

i) Attributes:

(1) numNodes – number of nodes in the network.

(2) numEdges – number of edges in the network.

(3) snode – the source node.

(4) tnode – the destination node.

(5) step – stores the current stage of the algorithm.

(6) edgeSelection type – the current criteria used to select an edge on the node.

(7) v – array of nodes in the network.

(8) e – array of edges in the network.

(9) totalNumberOfFlows – total number of LSPs created in the network.

(10) totalAntHops – total number of agent hops.

(11) totalAntsCreated –total number of ants created.

(12) totalTime – total (simulation) time.

(13) lastAntA – the last instance of RAA created.

ii) Methods:

(1) getCurrentLspNum – get the current LSP number for the LSP being created.

(2) findNode – find node with using the specified node name.

(3) dumpNode – used to display a specific node (for debugging and printing final results).

(4) dumpLsps – used to display all LSPs (and their paths) in the network (for debugging and printing final results).

(5) input_graph – used to setup the network. Reads a file that contains information on all network nodes and edges, and the edge selection criteria to be used. Creates an empty LSP database for every node in the network.

(6) rdb – used to setup the network. Adds forward and reverse edges on every node in the network.

(7) SelectForwardEdge – used by RAA to select a forward edge. This is based on a specific selection criterion. If the RAA has visited the node at the far end of an edge, the edge is not considered.

(8) SelectReverseEdge – used by RAA to select a reverse edge. If the RAA is in the process of searching for a EUP, the LSP number in passed into this method. This allows the RAA to select reverse edges that have this LSP number. If the RAA has visited the node at the far end of an edge, the edge is not considered.

(9) Routine_A – simulates the Routine A of RAA. This replaces the PATH message of the RSVP protocol. When RAA reaches the destination node, this portion of the RAA algorithm ends. The RAA first looks for a forward edge. If one is found it adds travels over the edge to the node at the far end. If RAA is in the process of looking for a EUP, and the node contains the LSP segment to be updated, RAA updates the EUP of the LSP segment. If a forward edge is not found then RAA looks for a reverse edge. If a reverse edge is found and RAA is not in the process of looking for a EUP, it adds a new 'LSP segment to update' (specifying the LSP number and SUP) and moves to the node at the far end of the edge. If the RAA is looking for a EUP and a reverse edge is found with the specified LSP, and moves to the node at the far end of the edge. If the RAA was unable to find a forward or reverse edge, it moves to the previous node. In this case any EUPs or SUPs that are no longer valid must be removed.

(10) step_0 – used to initialize the simulation.

(11) Routine_B – simulates Routine B of RBA. This replaces RESV of the RSVP protocol. The RBA is used to create and or update existing LSPs. The simplest case is when RAA did not find any LSPs to update. In this case a single RBA is used to create the LSP (using the information placed at the nodes by RAA). If there is at least one LSP that needs to be updated, multiple RBAs need to be created. The first RBA created is used to augment the new LSP by using the LSP segments stored in RAA, and the information stored at the nodes by RAA. The rest of the RBAs are used to update existing LSPs. For every LSP that needs to be updated a single RBA is created (even if the LSP has multiple segments to update).

(12) init – used to initialize the applet.

(13) mousePressed – used to move to the next stage of the algorithm.

(14) paintNode, xy, drawArrow, paintEdge, paint, update – used to draw the network.

4) Residue.java: Used to display the auxiliary graph. This part has been unmodified from the original source code.

APPENDIX B: SOURCE CODE

```
1  /*************************************/
2  /* File: Ant.java                    */
3  /* Ant based max flow simulation     */
4  /* (c) Shafiq Pirbhai, 2005, 2006    */
5  /*************************************/
6
7  import java.util.*;
8
9  class UpdateLspSegment
10 {
11     // LSPs are updated on segments.  Ant needs to keep track of the
12     // LSP number to be updated, the start (SUP node) and end (EUP node)
13     // points of the LSP segment that needs to be updated.
14     // For ant_a, LSP segments belong to different LSPs that need to be
15 updated.
16     // For ant_b that is used to
17     // 1) create a new LSP, there will be no LSP segments
18     // 2) augment a new LSP, LSP segments belong to different LSPs that need to
19     //    be updated.
20     // 3) update an old LSP, LSP segments belong to the same LSP that needs to
21     //    be updated.
22     int     number          = -1;  // LSP Number
23     int     supNode         = -1;  // SUP
24     int     eupNode         = -1;  // EUP
25
26     public  UpdateLspSegment(int l, int n)
27     {
28         // Creation of the LSP segment requires the LSP number and the
29         // SUP node.  So what about the EUP node?
30         // ant_a adds the EUP node when it finds it along its tour.
31         // and_b copies the EUP node from ant_a.
32         // it over from another .
33         number    = l;   // Set the LSP number.
34         supNode   = n;   // Set the SUP
35     }
36 }
37
38 public class Ant
39 {
40     private Node   node     = null;            // Current node of ant
41     private Edge   edge     = null;            // Last edge travelled by ant
42     private Vector tour     = new Vector(0,1); // Just for debugging purposes
43     private Vector visited  = new Vector(0,1); // Just for debugging purposes
44     private int    len      = 0;
45     private int    lsp      = -1;              // The ant uses this value to
46                                                // store the LSP number
47                                                // 1) path search (ant_a)
48                                                // 2) create,     (ant_b)
49                                                // 3) augment,    (ant_b)
50                                                // 4) update.     (ant_b)
51     // action only used by ant_b
52     private int    action   = 0;              // 0 = do nothing on link,
53                                                // 1 = create new Lsp on link,
54                                                // 2 = clear old Lsp on link,
55                                                // 3 = replace Lsp on link
56     private double totalTime = 0.0;
```

```
57
58      Stack   lspSegmentsToUpdate = new Stack();
59
60      public  Ant(Node n, int l)
61      {
62          node = n;
63          lsp  = l;
64          visited.addElement(n);
65          totalTime = 9.64;
66      }
67
68      public double getTotalTime()
69      {
70          return totalTime;
71      }
72
73      public void addEdge(Edge e, int l)
74      {
75          edge = e;
76          tour.addElement(e);
77          len += l;
78      }
79
80      public void setCurrentNode(Node n)
81      {
82          node = n;
83          visited.addElement(n);
84          totalTime += 20.25;
85      }
86      public int getTourLength()
87      {
88           return (len);
89      }
90
91      public int getCurrentNodeNum()
92      {
93         return node.number;
94      }
95
96      public int getLspNum()
97      {
98         return lsp;
99      }
100
101     public void addSUP(int l, int n)
102     {
103         UpdateLspSegment ul = new UpdateLspSegment(l,n);
104         lspSegmentsToUpdate.push(ul);
105     }
106
107     public void addEUP(int n)
108     {
109         UpdateLspSegment ul = (UpdateLspSegment) lspSegmentsToUpdate.peek();
110
111         if (ul != null)
112             ul.eupNode = n;
113     }
114
115     public void delEUP()
116     {
```

```
117             UpdateLspSegment ul = (UpdateLspSegment) lspSegmentsToUpdate.peek();
118
119             if (ul != null)
120                 ul.eupNode = -1;
121         }
122         public void delSUP()
123         {
124             lspSegmentsToUpdate.pop();
125         }
126
127         public int numLspsToUpdate()
128         {
129              return (lspSegmentsToUpdate.size());
130         }
131
132         public int currentLspToUpdate()
133         {
134              UpdateLspSegment ul;
135
136              if (numLspsToUpdate() > 0)
137              {
138                  ul = (UpdateLspSegment) lspSegmentsToUpdate.peek();
139                  if (ul.eupNode < 0)
140                      return ul.number;
141              }
142              return -1;
143         }
144
145         public int lastSUP()
146         {
147             UpdateLspSegment ul;
148
149              if (numLspsToUpdate() > 0)
150              {
151                  ul = (UpdateLspSegment) lspSegmentsToUpdate.peek();
152                  return (ul.supNode);
153              }
154              return -1;
155         }
156         public int lastEUP()
157         {
158              UpdateLspSegment ul;
159
160              if (numLspsToUpdate() > 0)
161              {
162                  ul = (UpdateLspSegment) lspSegmentsToUpdate.peek();
163                  if (ul.eupNode >= 0)
164                      return ul.eupNode;
165              }
166              return -1;
167         }
168
169         public int getSUP(int index)
170         {
171              UpdateLspSegment ul = (UpdateLspSegment)
172     lspSegmentsToUpdate.elementAt(index);
173              return ul.supNode;
174         }
175
176         public int getEUP(int index)
```

```
177         {
178              UpdateLspSegment ul = (UpdateLspSegment)
179     lspSegmentsToUpdate.elementAt(index);
180              return ul.eupNode;
181         }
182
183         public int getLspSegmentToUpdate(int index)
184         {
185             UpdateLspSegment ul = (UpdateLspSegment)
186     lspSegmentsToUpdate.elementAt(index);
187             return ul.number;
188         }
189
190         public void copyLspSegmentToUpdate(Ant ant, int lspIdx)
191         {
192             addSUP(ant.getLspSegmentToUpdate(lspIdx), ant.getSUP(lspIdx));
193             addEUP(ant.getEUP(lspIdx));
194         }
195
196         public void setAction(int a)
197         {
198             action = a;
199         }
200
201         public int  getAction()
202         {
203             return action;
204         }
205     }
```

```
1  /*************************************/
2  /* File: InfoPanel.java              */
3  /* Ant based max flow simulation     */
4  /* (c) Shafiq Pirbhai, 2005, 2006    */
5  /*************************************/
6  import java.awt.*;
7  import java.applet.*;
8  import java.util.*;
9  import java.awt.event.*;
10
11 public class InfoPanel extends Applet
12 {
13     TextArea ta = new TextArea();
14
15     public void init()
16     {
17         setLayout(new BorderLayout());
18         add(BorderLayout.CENTER, ta);
19     }
20
21     public void updatePanel(int f, int h, int n, double t, boolean done, Ant a)
22     {
23         ta.setText("");
24         ta.append("totalFlowsCreated   = " + f + "\n");
25         ta.append("totalAntHops        = " + h + "\n");
26         ta.append("totalAntsCreated    = " + n + "\n");
27         ta.append("totalTime           = " + t + "\n");
28
29         if (!done && a != null)
30         {
31             if (a.numLspsToUpdate() > 0)
32                 ta.append("List of LSPs to update ...\n");
33             for (int lspIdx = 0; lspIdx < a.numLspsToUpdate(); lspIdx++)
34             {
35                 ta.append((lspIdx + 1) + ". LspNum  = " +
36 a.getLspSegmentToUpdate(lspIdx) + "\n");
37                 ta.append((lspIdx + 1) + ".   End   = " + a.getEUP(lspIdx) +
38 "\n");
39                 ta.append((lspIdx + 1) + ".   Start = " + a.getSUP(lspIdx) +
40 "\n");
41             }
42         }
43
44         if (done)
45         {
46
47             ta.append("\nDone.\n");
48
49         }
50     }
51 }
```

```
1  /*************************************************************************/
2  /* File: Maxflow.java                                                    */
3  /* Ant based max flow simulation                                         */
4  /* Copyright (C) 1997, 1998 K. Ikeda   - Original Maxflow portion        */
5  /* Copyright (C) 2005, 2006 S. Pirbhai - Maxflow re-written to suite an  */
6  /*                                       agent based simulation that can */
7  /*                                       be used to maximize LSPs in a   */
8  /*                                       MPLS network                    */
9  /*************************************************************************/
10
11 import java.applet.*;
12 import java.awt.*;
13 import java.awt.event.*;
14 import java.util.*;
15 import java.io.*;
16 import java.net.URL;
17
18 class Lsp
19 {
20     int       inEdge   = -1;               /* Incoming Edge  */
21     int       outEdge  = -1;               /* Outgoing Edge  */
22     int       number   = -1;
23
24 }
25 class Node
26 {
27     final int MAX_LSPS_PER_NODE   = 100;
28     int       x;
29     int       y;
30     Vector    delta_plus = new Vector(0,1);  /* edge starts from this node */
31     Vector    delta_minus = new Vector(0,1); /* edge terminates at this node */
32     int       dist;                          /* distance from the start node */
33                                              /* Also used to determine whether
34                                                 node was visited */
35     int       prev;                          /* previous node on path */
36     int       p_edge;                        /* previous edge on path */
37     int       l;
38     int       w;
39     int       h;
40     int       number;                        /* Node number */
41     String    name;
42     Lsp       lspDatabase[] = new Lsp[MAX_LSPS_PER_NODE];
43
44     // Used to determine if ant_a has visited this node
45     boolean   visited()
46     {
47         return (dist>=0);
48     }
49
50     int       getMaxLsps()
51     {
52         return MAX_LSPS_PER_NODE;
53     }
54     // Used to get the incoming edge of a given lsp
55     int       getLspInEdge(int lspNum)
56     {
57         return (lspDatabase[lspNum].inEdge);
58     }
59
60     // Used to get the outgoing edge of a given lsp
```

```
61      int       getLspOutEdge(int lspNum)
62      {
63          return (lspDatabase[lspNum].outEdge);
64      }
65
66      // Used to clear a given lsp entry
67      void      clearLsp(int lspNum)
68      {
69          if (lspDatabase[lspNum].number == -1)
70          {
71              System.out.println("\nLSP " + lspNum + " does not exist on node " +
72  number + "\n");
73          }
74          lspDatabase[lspNum].number   = -1;
75          lspDatabase[lspNum].inEdge   = -1;
76          lspDatabase[lspNum].outEdge  = -1;
77      }
78
79      // Used to add a new lsp entry
80      void      addLsp(int lspNum, int inEdge, int outEdge)
81      {
82          if (lspDatabase[lspNum].number != -1)
83          {
84              System.out.println("\nLSP " + lspNum + " already exists on node " +
85  number + "\n");
86              return;
87          }
88          lspDatabase[lspNum].number  = lspNum;
89          lspDatabase[lspNum].inEdge  = inEdge;
90          lspDatabase[lspNum].outEdge = outEdge;
91      }
92
93      // Used to update an existing lsp entry
94      void      updateLsp(int lspNum, int inEdge, int outEdge)
95      {
96          if (lspDatabase[lspNum].number == -1)
97          {
98              System.out.println("\nLSP " + lspNum + " does not exist on node " +
99  number + "\n");
100             return;
101         }
102         lspDatabase[lspNum].number  = lspNum;
103         lspDatabase[lspNum].inEdge  = inEdge;
104         lspDatabase[lspNum].outEdge = outEdge;
105     }
106
107     // Used to determine whether a given lsp exists on the node.
108     boolean   hasLsp(int lspNum)
109     {
110         if (lspNum > 100) return false;
111         return (lspDatabase[lspNum].number >= 0);
112     }
113
114     boolean   hasLspOnInEdge(int lspNum, int inEdge)
115     {
116         if ((hasLsp(lspNum)) &&
117             (lspDatabase[lspNum].inEdge == inEdge))
118         {
119             return true;
120         }
```

```
121             return false;
122         }
123
124         // Used to determine whether a given lsp exists on specfic edges of the
125     node
126         boolean   hasLspOnEdges(int lspNum, int inEdge, int outEdge)
127         {
128             if ((hasLsp(lspNum)) &&
129                 (lspDatabase[lspNum].inEdge == inEdge) &&
130                 (lspDatabase[lspNum].outEdge == outEdge))
131             {
132                 return true;
133             }
134             return false;
135         }
136
137         // Used to get the first lsp on the given incoming edge
138         int       getLspOnInEdge(int inEdge)
139         {
140
141             for (int lspNum = 0; lspNum < 100; lspNum++)
142             {
143                 if ((lspDatabase[lspNum].inEdge == inEdge) &&
144                     (lspDatabase[lspNum].number >= 0))
145                 {
146                     return lspNum;
147                 }
148             }
149             return -1;
150         }
151     }
152
153     class Edge
154     {
155         int     rndd_plus;  /* initial vertex of this edge */
156         int     rndd_minus; /* terminal vertex of this edge */
157         int     capacity;   /* edge capacity */
158         int     flow;       /* edge flow */
159         int     cost;       /* edge cost */
160         int     st;         /* Used to color the st path */
161         int     number;     /* Edge number */
162         String  name;
163     }
164
165     public class Maxflow extends Applet implements MouseListener
166     {
167         final int       INITIALIZE   = 0; // Initialize
168         final int       FIND_PATH    = 1; // Routine A
169         final int       AUGMENT_FLOW = 2; // Routine B
170
171         final int       FIRST_EDGE   = 0; // Choose first edge found
172         final int       MIN_COST     = 1; // Choose min cost edge
173         final int       MAX_CAPACITY = 2; // Choose max capacity edge
174
175         int             numNodes,numEdges;
176         int             snode,tnode; /* start node, terminate node */
177         int             step;
178         int             edgeSelectionType = FIRST_EDGE;
179         Node            v[]  = new Node[100];
180         Edge            e[]  = new Edge[200];
```

```
181     int             totalNumberOfFlows;
182     int             totalAntHops;
183     int             totalAntsCreated;
184     double          totalTime;
185     Ant             lastAntA;
186
187
188     int getCurrentLspNum()
189     {
190         return (totalNumberOfFlows + 1);
191     }
192
193     int findNode(String name)
194     {
195         for (int i=0; i<numNodes; i++)
196             if (v[i].name.equals(name))
197                 return i;
198         return -1;
199     }
200
201     void dumpNode(int node, int lastNode)
202     {
203         System.out.print(" " + node);
204         if (node != lastNode)
205             System.out.print(",");
206     }
207
208     void dumpLsps()
209     {
210         System.out.print("\n\nPath of all LSPs in the network:");
211         for (int lspNum = 0; lspNum < v[snode].getMaxLsps(); lspNum++)
212         {
213             int nextNode = snode;
214             int outEdge;
215
216             if (v[snode].hasLsp(lspNum))
217             {
218
219                 System.out.print("\nLSP "+ lspNum + ": ");
220                 while (true)
221                 {
222                     outEdge = v[nextNode].getLspOutEdge(lspNum);
223                     dumpNode(nextNode, tnode);
224                     if (nextNode == tnode)
225                         break;
226
227                     nextNode = e[outEdge].rndd_minus;
228                 }
229             }
230         }
231         System.out.print("\n");
232     }
233     void input_graph(InputStream is) throws IOException
234     {
235         int x,y,l;
236         String  s;
237
238         Reader r = new BufferedReader(new InputStreamReader(is));
239         StreamTokenizer st = new StreamTokenizer(r);
240         st.commentChar('#');
```

```
241             st.nextToken(); numNodes = (int)st.nval;
242             st.nextToken(); numEdges = (int)st.nval;
243             st.nextToken(); s = st.sval;
244             st.nextToken(); edgeSelectionType = (int)st.nval;
245             for (int i = 0; i<numNodes; i++)
246             {
247                 Node node = new Node();
248                 st.nextToken(); node.name = st.sval;
249                 st.nextToken(); node.x = (int)st.nval;
250                 st.nextToken(); node.y = (int)st.nval;
251                 node.number = i;
252                 v[i] = node;
253                 for (int j = 0; j<100; j++)
254                 {
255                     Lsp lsp = new Lsp();
256                     v[i].lspDatabase[j] = lsp;
257                 }
258             }
259             for (int i = 0; i<numEdges; i++)
260             {
261                 Edge edge = new Edge();
262                 st.nextToken(); edge.name = st.sval;
263                 switch (st.nextToken())
264                 {
265                     case StreamTokenizer.TT_NUMBER:
266                         edge.rndd_plus = (int)st.nval;
267                         break;
268                     case StreamTokenizer.TT_WORD:
269                         edge.rndd_plus = findNode(st.sval);
270                         break;
271                     default:
272                         break;
273                 }
274                 switch (st.nextToken())
275                 {
276                     case StreamTokenizer.TT_NUMBER:
277                         edge.rndd_minus = (int)st.nval;
278                         break;
279                     case StreamTokenizer.TT_WORD:
280                         edge.rndd_minus = findNode(st.sval);
281                         break;
282                     default:
283                         break;
284                 }
285                 // Edge Capacity
286                 st.nextToken();
287                 edge.capacity = (int)st.nval;
288                 st.nextToken();
289                 // Edge Cost
290                 edge.cost = (int)st.nval;
291                 edge.flow = 0;
292                 edge.number = i;
293                 e[i] = edge;
294             }
295
296             step = FIND_PATH;
297         }
298
299         void rdb()
300         {
```

```
301         int i,k,linkNum=0;
302
303         for (i=0; i<numEdges; i++)
304         {
305             k = e[i].rndd_plus;
306             v[k].delta_plus.addElement(e[i]); // Add forward edge
307             k = e[i].rndd_minus;
308             v[k].delta_minus.addElement(e[i]); // Add reverse edge
309         }
310     }
311
312     int SelectForwardEdge(int nodeNum)
313     {
314         int  nextNodeNum;
315         int  edgeNum, nextEdgeNum = -1;
316         int  edgeIdx, edgesOnNode;
317         Edge edge;
318         int  minCost = Integer.MAX_VALUE;
319         int  bandwidth, maxBandwidth = 0;
320
321         edgesOnNode = v[nodeNum].delta_plus.size();
322
323         for (edgeIdx = 0; edgeIdx < edgesOnNode; edgeIdx++)
324         {
325             edge = (Edge) v[nodeNum].delta_plus.elementAt(edgeIdx);
326             edgeNum = edge.number;
327             if ((bandwidth = edge.capacity-edge.flow) == 0)
328                 continue;
329             nextNodeNum = edge.rndd_minus;
330             if (!v[nextNodeNum].visited())
331             {
332                 // Select edge based on search criteria (first, min-cost,
333                 // or max available bandwidth).
334                 switch (edgeSelectionType)
335                 {
336                     case FIRST_EDGE:
337                         return edgeNum; //return first edge found
338                     case MIN_COST:
339                         if (edge.cost < minCost)
340                         {
341                             minCost = edge.cost;
342                             nextEdgeNum = edgeNum;
343                         }
344                         break;
345                     case MAX_CAPACITY:
346                         if (bandwidth > maxBandwidth)
347                         {
348                             maxBandwidth = bandwidth;
349                             nextEdgeNum = edgeNum;
350                         }
351                         break;
352                 }
353             }
354         }
355         return nextEdgeNum;
356     }
357
358     int SelectReverseEdge(int nodeNum, int lspNum)
359     {
360         int  nextNodeNum;
```

```
361          int  edgeNum, nextEdgeNum = -1;
362          int  edgeIdx, edgesOnNode;
363          Edge edge;
364          int  minCost = Integer.MAX_VALUE;
365          int  bandwidth, maxBandwidth = 0;
366
367          edgesOnNode = v[nodeNum].delta_minus.size();
368          for (edgeIdx = 0; edgeIdx < edgesOnNode; edgeIdx++)
369          {
370              edge = (Edge) v[nodeNum].delta_minus.elementAt(edgeIdx);
371              edgeNum = edge.number;
372              if ((bandwidth = e[edgeNum].flow) == 0)
373                  continue;
374              nextNodeNum = e[edgeNum].rndd_plus;
375              if (!v[nextNodeNum].visited())
376              {
377                  if (lspNum != -1)
378                  {
379                      // This is the case where ant already crossed a reverse
380  edge
381                      // and has a Lsp Segment and SUP node in it's memory.  In
382                      // this case ant can select another reverse edge with the
383                      // same Lsp.
384                      if (v[nodeNum].hasLspOnInEdge(lspNum, edgeNum))
385                          nextEdgeNum = edgeNum;
386                  }
387                  else
388                  {
389                      // This is the case where ant has not crossed any reverse
390                      // edge, or has crossed one or more reverse edges but has
391                      // already found the EUP for the Lsp segments.  Therefore
392                      // select any reverse link that satisfies the search
393                      // criteria (first, min cost, or max capacity edge).
394                      switch (edgeSelectionType)
395                      {
396
397                          case FIRST_EDGE:
398                              return edgeNum;
399                          case MIN_COST:
400                              if (edge.cost < minCost)
401                              {
402                                  minCost = edge.cost;
403                                  nextEdgeNum = edgeNum;
404                              }
405                              break;
406                          case MAX_CAPACITY:
407                              if (bandwidth > maxBandwidth)
408                              {
409                                  maxBandwidth = bandwidth;
410                                  nextEdgeNum = edgeNum;
411                              }
412                             break;
413                       }
414                  }
415              }
416          }
417          return nextEdgeNum;
418      }
419
420      void Routine_A()      /* find an s-t path */
```

```
421     {
422         int    i,j,d, edgesOnNode, edgeIdx, currentNode;
423         Edge   edge;
424         Ant    ant_a = new Ant(v[snode], getCurrentLspNum());
425         int    lspToUpdate = -1;
426
427         totalAntsCreated++;
428
429         for (i=0; i<numNodes; i++)
430         {
431             v[i].p_edge = -1;
432             v[i].prev = v[i].dist = -1;
433             v[i].l = 0;
434         }
435         for (i=0; i<numEdges; i++)
436             e[i].st = -1;
437
438         d = 0;
439
440         v[snode].dist = 0;
441
442         i = 0;
443         edgesOnNode = v[snode].delta_plus.size();
444         for (edgeIdx = 0; edgeIdx < edgesOnNode; edgeIdx++)
445         {
446             edge = (Edge) v[snode].delta_plus.elementAt(edgeIdx);
447             if (i < e[edge.number].capacity)
448                 i = e[edge.number].capacity;
449         }
450
451         v[snode].l = i;
452         System.out.print("\nRAA: Search path LSP "+ ant_a.getLspNum() + ": ");
453         while (true)
454         {
455             // Ant starts processing on "currentNode"
456             currentNode = ant_a.getCurrentNodeNum();
457             dumpNode(currentNode, tnode);
458
459             if (currentNode == tnode)
460             {
461                 // We're at the destination node
462                 // Color the s-t path
463                 for (i = tnode; (j=v[i].prev)>=0; i = j)
464                     e[v[i].p_edge].st++;  // color the edges red
465
466                 // ant is done
467                 break;
468             }
469
470             d = v[currentNode].dist;
471
472             // Ant looks for an available forward edge
473             j = SelectForwardEdge(currentNode);
474             if (j >= 0)
475             {
476                 // Found forward edge
477                 // Ant updates tables on this node
478                 i = e[j].rndd_minus;
479                 v[i].dist = d+1;
480                 v[i].prev = currentNode;
```

```
481                 v[i].p_edge = j;
482                 v[i].l = Math.min(1, e[j].capacity-e[j].flow);
483                 e[j].st++;
484
485                 if ((lspToUpdate = ant_a.currentLspToUpdate()) >= 0)
486                    if (v[i].hasLsp(lspToUpdate))
487                        ant_a.addEUP(i);  // Remember the EUP node
488
489                 // Ant moves over forward edge to next node
490                 ant_a.addEdge(e[j], 1);
491                 ant_a.setCurrentNode(v[i]);
492
493             }
494             else
495             {
496                 // No forward edge found on this node.
497                 // Look for reverse edge on the node.
498                 j = SelectReverseEdge(currentNode, ant_a.currentLspToUpdate());
499                 if ((j >= 0) &&
500                     (ant_a.currentLspToUpdate() < 0) &&
501                     (currentNode != ant_a.lastEUP()))
502                 {
503                     // Found reverse edge
504                     // Ant updates tables on this node
505                     lspToUpdate = v[currentNode].getLspOnInEdge(j);
506
507                     i = e[j].rndd_plus;
508                     v[i].dist = d+1;
509                     v[i].prev = currentNode;
510                     v[i].p_edge = j;
511                     v[i].l = Math.min(1,e[j].flow);
512                     e[j].st++;
513
514                     ant_a.addSUP(lspToUpdate, currentNode); // Remember SUP
515 node
516                     // Ant moves over reverse edge to next node
517                     ant_a.addEdge(e[j], 1);
518                     ant_a.setCurrentNode(v[i]);
519
520                 }
521                 else if ((j >= 0) &&
522                          ((lspToUpdate = ant_a.currentLspToUpdate()) >= 0) &&
523                          v[currentNode].hasLspOnEdges(lspToUpdate, j,
524 v[currentNode].p_edge))
525                 {
526                     // Found reverse edge with the same LSP we're updating
527                     // Ant updates tables (no need to add an SUP node because
528 it
529                     // is the same LSP
530                     i = e[j].rndd_plus;
531                     v[i].dist = d+1;
532                     v[i].prev = currentNode;
533                     v[i].p_edge = j;
534                     v[i].l = Math.min(1,e[j].flow);
535                     e[j].st++;
536                     // Ant moves over reverse edge to next node
537                     ant_a.addEdge(e[j], 1);
538                     ant_a.setCurrentNode(v[i]);
539                 }
540                 else
```

```
                {
                    // No more edges available on this node
                    if (currentNode == snode)
                    {
                        // We're at the start node, we're done.
                        break;
                    }
                    else
                    {
                        Node prevNode = v[v[currentNode].prev];
                        // Ant moves to previous node.
                        ant_a.addEdge(e[v[currentNode].p_edge], 1);
                        ant_a.setCurrentNode(prevNode);

                        if (currentNode == ant_a.lastEUP())
                        {
                            ant_a.delEUP();
                        }
                        else if (currentNode == ant_a.lastSUP())
                        {
                            ant_a.delSUP();
                        }
                    }
                }
            }
        }  // while true
        // Get total number of ant hops
        totalAntHops += ant_a.getTourLength();
        totalTime += ant_a.getTotalTime();
        lastAntA = ant_a;
        return;
    }

    void step0()  /* initialize */
    {
        for (int i=0; i<numEdges; i++)
        {
            e[i].flow = 0;
            //for (int j = 0; j < 100; j++)
            //    e[i].lsp[j] = -1;
        }

        for (int i=0; i<numNodes; i++)
        {
            v[i].prev = v[i].dist = -1;
            v[i].l = 0;
            for (int j=0; j < 100; j++)
            {
                if (v[i].hasLsp(j))
                    v[i].clearLsp(j);
            }
        }
        totalNumberOfFlows = 0;
        totalAntHops       = 0;
        totalAntsCreated   = 0;
        totalTime          = 0;
        System.out.print("\nPath of all agents (Search path, Create, Augment
and Update LSPs):");
    }
```

```
601     void Routine_B()      /* augment the flow */
602     {
603         int i,j,f;
604         int prevEdge = -1;
605         int nextEdge = -1;
606         int prevNode = -1;
607         int currentNode = tnode;
608         Ant ant_b;
609         Vector  bAnts = new Vector(0,1);
610         int numLspsToUpdate, lspIdx, lspNum;
611         int numBAnts, bAntIdx;
612
613         if (v[tnode].dist<0)
614             return;
615
616         ant_b = new Ant(v[tnode], getCurrentLspNum());
617         totalAntsCreated++;
618
619         f = v[tnode].1;
620
621         if ((numLspsToUpdate = lastAntA.numLspsToUpdate()) == 0)
622         {
623             System.out.print("\n    RBA: Create  LSP "+ ant_b.getLspNum() + ":
624 ");
625             // Simplest case:  This ant_b just needs to create the new LSP
626             nextEdge = -1;
627
628             while (true)
629             {
630                 currentNode = ant_b.getCurrentNodeNum();
631                 dumpNode(currentNode, snode);
632
633                 // Get previous edge
634
635                 prevNode = v[currentNode].prev;
636                 prevEdge = v[currentNode].p_edge;
637                 v[currentNode].addLsp(ant_b.getLspNum(), prevEdge, nextEdge);
638
639                 if (prevEdge < 0)
640                 {
641                     // We're done
642                     break;
643                 }
644
645                 // Update the edge flow bandwidth
646                 e[prevEdge].flow += f;
647                 if (e[prevEdge].flow > e[prevEdge].capacity)
648                 {
649                     // Should never happen
650                     System.out.println("Flow greater than capacity while
651 creating LSP " + ant_b.getLspNum() + "on node " + currentNode);
652                 }
653
654                 // Update ant tour ...
655                 ant_b.addEdge(e[prevEdge], 1);
656                 // ant moved to previous node
657                 ant_b.setCurrentNode(v[prevNode]);
658
659                 nextEdge = prevEdge;
660             }
```

```
661             totalAntHops += ant_b.getTourLength();
662             totalTime += ant_b.getTotalTime();
663         }
664         else
665         {
666             System.out.print("\n    RBA: Augment LSP "+ ant_b.getLspNum() + ":
667 ");
668             // This ant_b needs to augment the new LSP, and replacing some
669             // other LSPs segments along the way
670             for (lspIdx = 0; lspIdx < numLspsToUpdate; lspIdx++)
671             {
672                 ant_b.copyLspSegmentToUpdate(lastAntA, lspIdx);
673             }
674             nextEdge = -1;
675
676             while (true)
677             {
678                 currentNode = ant_b.getCurrentNodeNum();
679                 dumpNode(currentNode, snode);
680
681                 // Get previous edge
682                 if (ant_b.lastEUP() == currentNode)
683                 {
684                     ant_b.delEUP();
685                     prevEdge =
686 v[currentNode].getLspInEdge(ant_b.currentLspToUpdate());
687                     prevNode = e[prevEdge].rndd_plus;  // Initial vertex of
688 edge
689                     ant_b.setAction(3); // change to replace mode
690                 }
691                 else if ((ant_b.lastEUP() < 0) && (ant_b.lastSUP() >= 0))
692                 {
693                     if (ant_b.lastSUP() == currentNode)
694                     {
695                         v[currentNode].clearLsp(ant_b.currentLspToUpdate());
696                         ant_b.delSUP();
697                         prevNode = v[currentNode].prev;
698                         prevEdge = v[currentNode].p_edge;
699                         ant_b.setAction(1); // change to create mode
700                     }
701                     else
702                     {
703                         prevEdge =
704 v[currentNode].getLspInEdge(ant_b.currentLspToUpdate());
705                         prevNode = e[prevEdge].rndd_plus; // Initial vertex of
706 edge
707                         v[currentNode].clearLsp(ant_b.currentLspToUpdate());
708                         // still in replace mode - no need to set it again
709                     }
710
711                 }
712                 else
713                 {
714                     prevNode = v[currentNode].prev;
715                     prevEdge = v[currentNode].p_edge;
716                     ant_b.setAction(1); // change to create mode
717                 }
718
719                 v[currentNode].addLsp(ant_b.getLspNum(), prevEdge, nextEdge);
720                 if (prevNode < 0)
```

```
721                 {
722                     // We're done
723                     break;
724                 }
725
726                 // Update the edge flow bandwidth
727                 if (ant_b.getAction() != 3)
728                 {
729                     if (e[prevEdge].rndd_minus==currentNode)
730                     {
731                         e[prevEdge].flow += f;
732                         if (e[prevEdge].flow > e[prevEdge].capacity)
733                         {
734                             // Should never happen
735                             System.out.println("\nFlow greater than capacity
736 while augmenting LSP " + ant_b.getLspNum() + "on node " + currentNode);
737                         }
738                     }
739                 }
740                 else if (e[prevEdge].rndd_plus==currentNode)
741                 {
742                     // Should never happen
743                     System.out.println("\nError while augmenting LSP " +
744 ant_b.getLspNum() + "on node " + currentNode);
745                     //e[prevEdge].flow -= f;
746                 }
747
748                 // Update ant tour ...
749                 ant_b.addEdge(e[prevEdge], 1);
750                 // Ant moves to previous node
751                 ant_b.setCurrentNode(v[prevNode]);
752
753                 nextEdge = prevEdge;
754             }
755             totalAntHops += ant_b.getTourLength();
756             totalTime += ant_b.getTotalTime();
757
758             // Create all other ant_b's that are needed to update LSP segments
759             for (lspIdx = 0; lspIdx < numLspsToUpdate; lspIdx++)
760             {
761                 // For every LSP that needs to be updated we create a new ant
762                 boolean matchFound = false;
763
764                 // Get the LspNum for the Lsp segment to update
765                 lspNum = lastAntA.getLspSegmentToUpdate(lspIdx);
766
767                 // Get number of ant_b's already created.
768                 numBAnts = bAnts.size();
769                 for (bAntIdx = 0; bAntIdx < numBAnts; bAntIdx++)
770                 {
771                     ant_b = (Ant) bAnts.elementAt(bAntIdx);
772                     // For each LSP that needs to be updated. Use one ant.
773                     // This means that the same ant is used to update
774                     // multiple LSP segments of the same LSP.
775                     if (ant_b.getLspNum() == lspNum)
776                     {
777                         // We already have a ant_b responsible for updating
778                         // one or more Lsp segments for this Lsp
779                         matchFound = true;
780                         ant_b.copyLspSegmentToUpdate(lastAntA, lspIdx);
```

```
781                     }
782                 }
783                 if (!matchFound)
784                 {
785                     // No ant_b currently exists to update this LSP so
786                     // create a new one
787                     ant_b = new Ant(v[tnode], lspNum);
788                     totalAntsCreated++;
789                     ant_b.copyLspSegmentToUpdate(lastAntA, lspIdx);
790                     bAnts.addElement(ant_b);
791                 }
792             }
793             numBAnts = bAnts.size();
794
795             for (bAntIdx = 0; bAntIdx < numBAnts; bAntIdx++)
796             {
797                 // Send the ants marching
798                 System.out.print("\n        RBA: Update LSP "+
799 ant_b.getLspNum() + ": ");
800                 ant_b = (Ant) bAnts.elementAt(bAntIdx);
801                 nextEdge = -1;
802                 while ((currentNode = ant_b.getCurrentNodeNum()) !=
803 ant_b.lastEUP())
804                 {
805                     dumpNode(currentNode, snode);
806                     // Need to get to the first node to update
807                     prevEdge = v[currentNode].getLspInEdge(ant_b.getLspNum());
808                     prevNode = e[prevEdge].rndd_plus;  // Initial Vertex of e
809                     // Update ant tour ...
810                     ant_b.addEdge(e[prevEdge], 1);
811                     ant_b.setCurrentNode(v[prevNode]);
812                     nextEdge = prevEdge;
813                 }
814                 while (true)
815                 {
816                     currentNode = ant_b.getCurrentNodeNum();
817                     dumpNode(currentNode, snode);
818
819                     prevNode = v[currentNode].prev;
820                     prevEdge = v[currentNode].p_edge;
821                     // Get previous edge
822                     if (ant_b.lastEUP() == currentNode)
823                     {
824                         int outEdge =
825 v[currentNode].getLspOutEdge(ant_b.getLspNum());
826
827                         ant_b.delEUP();
828                         v[currentNode].updateLsp(ant_b.getLspNum(), prevEdge,
829 outEdge);
830                         ant_b.setAction(1); // Change to create mode
831                     }
832                     else if ((ant_b.lastEUP() < 0) && (ant_b.lastSUP() >= 0))
833                     {
834                         if (ant_b.getAction() == 1)
835                         {
836                             // Currently in "create mode".
837                             int inEdge =
838 v[currentNode].getLspInEdge(ant_b.getLspNum());
839                             if (inEdge != -1)
840                             {
```

```
                                    // Update LSP
                                    v[currentNode].updateLsp(ant_b.getLspNum(),
inEdge, nextEdge);
                                    ant_b.setAction(2); // Change to "clear mode"
                                }
                                else
                                {
                                    // Add LSP
                                    v[currentNode].addLsp(ant_b.getLspNum(),
prevEdge, nextEdge);
                                }
                            }
                            else if (ant_b.getAction() == 2)
                            {
                                if (ant_b.lastSUP() == currentNode)
                                {
                                    ant_b.delSUP();
                                    ant_b.setAction(0); // Change to no-op mode
                                }
                                else
                                {
                                    // Clear LSP
                                    v[currentNode].clearLsp(ant_b.getLspNum());
                                }
                            }
                        }
                        else
                        {
                            if (ant_b.getAction() != 0)
                            {
                                // Should never happen - Putting this just in case
                                System.out.println("Ant at node " + currentNode + "
in bad state: " + ant_b.getAction());
                            }
                        }

                        if ((prevNode < 0) || (ant_b.numLspsToUpdate() == 0))
                        {
                            // We're done
                            break;
                        }
                        // Update the the edge flow bandwidth
                        if (ant_b.getAction() != 0)
                        {
                            if (e[prevEdge].rndd_minus==currentNode)
                            {
                                e[prevEdge].flow += f;
                                if (e[prevEdge].flow > e[prevEdge].capacity)
                                {
                                    // Should never happen
                                    System.out.println("\nFlow greater than
capacity while updating LSP " + ant_b.getLspNum() + "on node " + currentNode);
                                }
                            }
                            else if (e[prevEdge].rndd_plus==currentNode)
                            {
                                e[prevEdge].flow -= f;
                                if (e[prevEdge].flow < 0)
                                {
                                    // Should never happen
```

```
                                System.out.println("\nFlow less than 0 while
updating LSP " + ant_b.getLspNum() + "on node " + currentNode);
                            }
                        }
                    }
                    // Update ant tour ...
                    ant_b.addEdge(e[prevEdge], 1);
                    // Ant moves to previous node
                    ant_b.setCurrentNode(v[prevNode]);

                    nextEdge = prevEdge;
                }
                totalAntHops += ant_b.getTourLength();
                totalTime += ant_b.getTotalTime();
            }
        }
    }

    public void init()
    {
        String mdname = getParameter("inputfile");
        //String mdname = "d2.obj";
        try
        {
            InputStream is;

            is = new URL(getDocumentBase(),mdname).openStream();
            //is = new URL(mdname).openStream();
            input_graph(is);
            try
            {
                if (is != null)
                    is.close();
            } catch(Exception e)
            {
            }
        }
        catch (FileNotFoundException e)
        {
            System.err.println("File not found.");
        }
        catch (IOException e)
        {
            System.err.println("Cannot access file.");
        }

        String s = getParameter("s");
        if (s != null)
            snode = Integer.parseInt(s);
        else
            snode = 0;

        s = getParameter("t");
        if (s != null)
            tnode = Integer.parseInt(s);
        else
            tnode = numNodes-1;

        setBackground(Color.white);
        rdb();
```

```
961             addMouseListener(this);
962             step0();
963         }
964
965         public void paintNode(Graphics g, Node n, FontMetrics fm)
966         {
967             String s;
968             int x = n.x;
969             int y = n.y;
970             int w = fm.stringWidth(n.name) + 10;
971             int h = fm.getHeight() + 4;
972             n.w = w;
973             n.h = h;
974             Color   c;
975
976             if (n.dist<0)
977                 c = Color.gray;
978             else
979                 c = Color.blue;
980             g.setColor(c);
981             g.drawRect(x-w/2,y-h/2,w,h);
982
983             g.setColor(getBackground());
984             g.fillRect(x-w/2+1,y-h/2+1,w-1,h-1);
985
986             g.setColor(c);
987             g.drawString(n.name,x-(w-10)/2,(y-(h-4)/2)+fm.getAscent());
988         }
989
990         int [] xy(int a, int b, int w, int h)
991         {
992             int x[] = new int[2];
993
994             if (Math.abs(w*b)>=Math.abs(h*a))
995             {
996                 x[0] = ((b>=0)?1:-1)*a*h/b/2;
997                 x[1] = ((b>=0)?1:-1)*h/2;
998             }
999             else
1000            {
1001                x[0] = ((a>=0)?1:-1)*w/2;
1002                x[1] = ((a>=0)?1:-1)*b*w/a/2;
1003            }
1004            return x;
1005        }
1006
1007        void drawArrow(Graphics g,int x1,int y1,int x2,int y2)
1008        {
1009            int a = x1-x2;
1010            int b = y1-y2;
1011
1012            double  aa = Math.sqrt(a*a+b*b)/16.0;
1013            double  bb = b/aa;
1014                aa = a/aa;
1015            g.drawLine(x2,y2,x2+(int)((aa*12+bb*5)/13),
1016                    y2+(int)((-aa*5+bb*12)/13));
1017            g.drawLine(x2,y2,x2+(int)((aa*12-bb*5)/13),
1018                    y2+(int)((aa*5+bb*12)/13));
1019            g.drawLine(x1,y1,x2,y2);
1020        }
```

```
1021
1022        public void paintEdge(Graphics g, Edge e, FontMetrics fm)
1023        {
1024            Node v1 = v[e.rndd_plus];
1025            Node v2 = v[e.rndd_minus];
1026            Color   c;
1027
1028            int a = v1.x-v2.x;
1029            int b = v1.y-v2.y;
1030
1031            int x1[] = xy(-a,-b,v1.w,v1.h);
1032            int x2[] = xy(a,b,v2.w,v2.h);
1033
1034            if (e.st>0)
1035            {
1036                c = Color.red;
1037            }
1038            else if ((v1.dist>=0)&&(v2.dist>=0)&&(v[tnode].dist<0))
1039            {
1040                c = Color.blue;
1041            }
1042            else
1043            {
1044                c = Color.gray;
1045            }
1046            g.setColor(c);
1047            drawArrow(g,v1.x+x1[0],v1.y+x1[1],v2.x+x2[0],v2.y+x2[1]);
1048
1049            //int w = fm.stringWidth(""+e.flow+"/"+e.capacity+"/"+e.cost);
1050            int w = fm.stringWidth(""+e.flow+"/"+e.capacity+"/"+e.number);
1051            int h = fm.getHeight();
1052
1053            g.setColor(getBackground());
1054            g.fillRect((v1.x+v2.x-w)/2,(v1.y+v2.y-h)/2,w,h);
1055            g.setColor(Color.black);
1056            //g.drawString(""+e.flow+"/"+e.capacity+"/"+e.cost,
1057            //    (v1.x+v2.x-w)/2,(v1.y+v2.y-h)/2+fm.getAscent());
1058            g.drawString(""+e.flow+"/"+e.capacity+"/"+e.number,
1059                (v1.x+v2.x-w)/2,(v1.y+v2.y-h)/2+fm.getAscent());
1060        }
1061
1062        public void paint(Graphics g)
1063        {
1064            FontMetrics fm = g.getFontMetrics();
1065            for (int i=0; i<numNodes; i++)
1066                paintNode(g,v[i],fm);
1067            for (int i=0; i<numEdges; i++)
1068                paintEdge(g,e[i],fm);
1069            Residue res = (Residue)getAppletContext().getApplet("Residue");
1070            if (res != null)
1071                res.set(1,numNodes,numEdges,snode,tnode,v,e);
1072            InfoPanel inf = (InfoPanel)getAppletContext().getApplet("InfoPanel");
1073            if (inf != null)
1074                inf.updatePanel(totalNumberOfFlows,
1075                                totalAntHops,
1076                                totalAntsCreated,
1077                                totalTime,
1078                                ((!v[tnode].visited()) && (v[snode].visited())),
1079                                lastAntA);
1080        }
```

```
1081
1082        public void update(Graphics g)
1083        {
1084            paint(g);
1085        }
1086
1087        public void mousePressed(MouseEvent ev)
1088        {
1089            if (step==AUGMENT_FLOW)
1090            {
1091                Routine_B();
1092                if (v[tnode].dist<0)
1093                {
1094                    dumpLsps();
1095                    step = INITIALIZE;
1096                }
1097                else
1098                {
1099                    totalNumberOfFlows++;
1100                    step = FIND_PATH;
1101                }
1102            }
1103            else if (step==FIND_PATH)
1104            {
1105                Routine_A();
1106                step = AUGMENT_FLOW;
1107            }
1108            else
1109            {
1110                step0();
1111                step = FIND_PATH;
1112            }
1113            repaint();
1114        }
1115        public void mouseClicked(MouseEvent event) {}
1116        public void mouseReleased(MouseEvent event) {}
1117        public void mouseEntered(MouseEvent event) {}
1118        public void mouseExited(MouseEvent event) {}
1119
1120    }
```

```
1   /*************************************/
2   /* File: Residue.java                */
3   /*************************************/
4   import java.applet.*;
5   import java.awt.*;
6
7   public class Residue extends Applet implements Runnable {
8         Thread th;
9         int    n=0,m=0;
10        int    snode=0,tnode=0;    /* start node, terminate node */
11        int    flag = -1;
12        Node   v[];
13        Edge   e[];
14
15        public void set(int flag, int n, int m,
16                        int snode, int tnode, Node v[], Edge e[])  {
17              this.flag = flag;
18              this.n = n;
19              this.m = m;
20              this.snode = snode;
21              this.tnode = tnode;
22              this.v = v;
23              this.e = e;
24              setBackground(Color.white);
25        }
26
27        public void paintNode(Graphics g, Node n, FontMetrics fm) {
28              String s;
29              int x = n.x;
30              int y = n.y;
31              int w = fm.stringWidth(n.name) + 10;
32              int h = fm.getHeight() + 4;
33              n.w = w;
34              n.h = h;
35              Color  c;
36
37              if (n.dist<0)
38                    c = Color.gray;
39              else
40                    c = Color.blue;
41              g.setColor(c);
42              g.drawRect(x-w/2,y-h/2,w,h);
43
44              g.setColor(getBackground());
45              g.fillRect(x-w/2+1,y-h/2+1,w-1,h-1);
46
47              g.setColor(c);
48              g.drawString(n.name,x-(w-10)/2,(y-(h-4)/2)+fm.getAscent());
49        }
50
51        int [] xy(int a, int b, int w, int h, int dx, int dy) {
52              int    x[] = new int[2];
53
54              w -= ((a>=0)?2:-2)*dx-3;
55              h -= ((b>=0)?2:-2)*dy-2;
56
57              if (Math.abs(w*b)>=Math.abs(h*a)) {
58                    x[0] = ((b>=0)?1:-1)*a*h/b/2;
59                    x[1] = ((b>=0)?1:-1)*h/2;
60              } else {
```

```
                x[0] = ((a>=0)?1:-1)*w/2;
                x[1] = ((a>=0)?1:-1)*b*w/a/2;
        }

        x[0] += dx;
        x[1] += dy;
        return x;
    }

    void drawArrow(Graphics g,int x1,int y1,int x2,int y2) {
        int    a = x1-x2;
        int    b = y1-y2;

        double aa = Math.sqrt(a*a+b*b)/16.0;
        double bb = b/aa;
               aa = a/aa;
        g.drawLine(x2,y2,x2+(int)((aa*12-bb*5)/13),
                         y2+(int)((aa*5+bb*12)/13));
        g.drawLine(x1,y1,x2,y2);
    }

    void drawEdge(Graphics g, FontMetrics fm,
                  int x1, int y1, int x2, int y2, int c) {
        int a = x1-x2;
        int b = y1-y2;

        drawArrow(g,x1,y1,x2,y2);

        int w = fm.stringWidth("" + c);
        int h = fm.getHeight();

        int x = (x1+x2)/2;
        int y = (y1+y2)/2;
        if ((b<0)||((b==0)&&(a<0))) x += fm.stringWidth(" ");
        else x -= w+fm.stringWidth(" ");
        if ((a>0)||((a==0)&&(b<0))) y += 1;
        else   y -= h;

        g.setColor(getBackground());
        g.fillRect(x,y,w,h);
        if (c!=0) {
            g.setColor(Color.black);
            g.drawString("" + c,x,y+fm.getAscent());
        }
    }

    public void paintEdge(Graphics g, Edge e, FontMetrics fm) {
        Node v1 = v[e.rndd_plus];
        Node v2 = v[e.rndd_minus];
        Color  c;

        if ((e.st>0)&&(v1.dist<v2.dist))
            c = Color.red;
        else if ((e.st==0)&&(v1.dist<v2.dist))
            c = Color.blue;
        else
            c = Color.gray;

        int a = v1.x-v2.x;
        int b = v1.y-v2.y;
```

```
121
122             double aa = Math.sqrt(a*a+b*b)/2.0;
123             double bb = -b/aa;
124                     aa = a/aa;
125
126             int x1[] = xy(-a,-b,v1.w,v1.h,(int)bb,(int)aa);
127             int x2[] = xy(a,b,v2.w,v2.h,(int)bb,(int)aa);
128             if (e.capacity-e.flow==0)
129                     g.setColor(getBackground());
130             else
131                     g.setColor(c);
132             drawEdge(g,fm,v1.x+x1[0],v1.y+x1[1],v2.x+x2[0],v2.y+x2[1],
133                             e.capacity-e.flow);
134
135             if ((e.st>0)&&(v1.dist>v2.dist))
136                     c = Color.red;
137             else if ((e.st==0)&&(v1.dist>v2.dist))
138                     c = Color.blue;
139             else
140                     c = Color.gray;
141             x1 = xy(a,b,v2.w,v2.h,-(int)bb,-(int)aa);
142             x2 = xy(-a,-b,v1.w,v1.h,-(int)bb,-(int)aa);
143             if (e.flow==0)
144                     g.setColor(getBackground());
145             else
146                     g.setColor(c);
147             drawEdge(g,fm,v2.x+x1[0],v2.y+x1[1],v1.x+x2[0],v1.y+x2[1],
148                             e.flow);
149     }
150
151     public void paint(Graphics g) {
152             FontMetrics fm = g.getFontMetrics();
153             for (int i=0; i<n; i++)
154                     paintNode(g,v[i],fm);
155             for (int i=0; i<m; i++)
156                     paintEdge(g,e[i],fm);
157     }
158
159     public void update(Graphics g) {
160             paint(g);
161     }
162
163     public void start() {
164             if (th==null) {
165                     th = new Thread(this);
166                     th.start();
167             }
168     }
169
170     public void run() {
171             while (true) {
172                     try {
173                             th.sleep(100);
174                     }
175                     catch (InterruptedException e) {}
176                     if (flag-- > 0)
177                             repaint();
178             }
179     }
```