Enhancing the Internet Routing Architecture

J.J. Garcia-Luna-Aceves and Bradley R. Smith
Computer Engineering Department. University of California
Santa Cruz, CA 95064
jj@soe.ucsc.edu, brad@soe.ucsc.edu

Abstract

The best-effort communication model used in the Internet architecture has proven surprisingly powerful. However, limitations are being encountered as it is applied to more demanding applications and environments. Specifically, new requirements for policy-based control and performance constraint of forwarding topologies are not well supported by the best-effort model. Proposed centralized, on-demand solutions suffer from limitations in robustness, efficiency, responsiveness, and effectiveness. We introduce an enhancement to the Internet architecture called network layer enclaves (NLEs). NLEs provide a solution for policy-based management of network resources without sacrificing the strengths of the existing Internet architecture. They expand the Internet architecture goal of supporting the development and integration of diverse network technologies to include supporting the exploitation of special capabilities of specific network technologies. They accomplish this by augmenting the original Internet strategy of making minimal assumptions of underlying network technologies in the forwarding plane with support for the maintenance of detailed knowledge of special network capabilities in the routing plane. As a result, NLEs provide effective policy-based routing control while maintaining the distributed, end-to-end nature of the Internet.

1 Motivation

1.1 The Internet Routing Architecture

The current Internet architecture, as implemented in the global Internet, is based on the *catenet* model for internetworking [6, 7, 8]. The two basic components of a catenet are networks and gateways, where a catenet is formed by the interconnecting of networks with gateways. A primary goal of the catenet model, and therefore the Internet architecture, was to encourage the development and integration of new networking technologies into the developing catenets. To achieve this goal, only minimal assumptions were made of networks and the routing computation by the catenet model. Networks were assumed to support the attachment of a number of computers, transport datagrams, allow switched access so that attached computers could "quickly" send datagrams to different destinations, and provide best-effort delivery. The definition of *best-effort* allowed datagrams to be dropped, or delivered out of order, and assumes a routing computation that computes routes for a single forwarding class chosen by the optimization of a single, typically delay-related metric.

Underlying the current architecture are a number of design principles that are largely responsible for the scalable, robust characteristics of Internet technologies. A primary design principle is that the Internet is *distributed* in the sense that state existing in an internet is autonomously controlled by a process collocated with the state. The goal being that state can only be destroyed if the controlling process itself is destroyed. This property is also called *fate-sharing* [4].

Another design principle is that the network core should be kept architecturally *simple*, with complexity being pushed to the network edges. The goal of this principle is both performance and expandability. By keeping the network itself conceptually simple and clean, it becomes easier to implement efficient, high performance systems, and it allows for easier enhancement of the architecture.

The final design principle discussed here is a generalization of the *end-to-end* principle [18]. The end-to-end principle argues that a function that can completely and correctly be implemented only with involvement of the application endpoints should, in general, only be implemented in the endpoints. An example of the application of this principle is the implementation of reliable network communications. Since reliable communications cannot be completely implemented without the participation of the end-system application processes (to protect against corruption within the end system), the end-to-end principle argues reliable communication should be implemented in the end systems.

The generalization of this principle discussed here, which might be called the *right-sizing* principle, argues that a function should be implemented in the smallest scope required for a complete and correct implementation, *and no smaller.* An example application of this principle in the Internet is that the routing function is implemented on *all* routers in an Internet, and not just on some subset. The goal of this principle is to encourage architectural integrity by avoiding or discouraging partial solutions and solutions cobbled together from disparate parts, and to discourage architectural bloat from the spread of functionality throughout the architecture to address special case circumstances.

As a result of these principles, the Internet architecture is *robust* in the sense that it minimizes the existence of *brittle* state that causes the failure of a single component to affect whole subsystems of an internet, effectively *localizing* the affects of failures in an internet.

The Internet architecture is *efficient* and *responsive* for a number of reasons. First, by discouraging the use of centralized control systems, it encourages the use of *simplex* control communication over *duplex* control communication. In a distributed control system, since state and its controlling process are collocated, only simplex notification of the controlling process is required to respond to an event. In contrast, in a centralized control system, duplex control communication is required to, first, notify the controlling process of an event, and for that process to then update remote state in order to respond to the event. Second, by avoiding the need for soft-state timers and repair mechanisms required by centralized control systems, it encourages the use of simple state management mechanisms. Third, by limiting the scope of implementation of a function to only the extent actually required to correctly implement the function, it encourages simplicity and efficiency in its implementation.

Lastly, by discouraging the use of centralized, on-demand, routing algorithms, it encourages the use of more efficient and responsive distributed routing algorithms [12]. The routing protocols used in most of today's computer networks are based on shortest-path algorithms that can be classified as distance-vector or link-state. Distance-vector protocols work by propagating updates giving the distance to a destination to neighboring routers whose routing tables may change as a result of the update. Link-state protocols work by flooding updates describing the state of links in the network to all routers in the network. Recently, a hybrid class of protocols, called link-vector [12], has been defined that works by propagating link-state updates only to routers whose routing tables may change as a result of the update. However, in a system depending on on-demand routing computations a link-state, complete topology protocol is required to ensure an ingress router has the information it needs to compute an optimal route. In contrast, distributed, hop-by-hop routing systems can work with link-vector, partial topology protocols as each routing process is ensured of learning from its neighbors of all links composing optimal routes to all destinations in the internet. As a result, by encouraging the use of distributed, partial-topology routing solutions, the Internet architecture encourages more efficient and responsive routing solutions.

The Internet architecture is *effective* because it encourages, via the right-sizing principle, the integrated and comprehensive implementation of functionality in contrast to ad-hoc, patch-work, or partial implementations.

1.2 The Problems with Today's Internet Architecture

The best-effort model of communication has proven surprisingly powerful. Indeed, much of the success of the Internet architecture can be attributed to this inspired design decision. However, implicit in this best effort communications model are the assumptions of homogeneous network use policies, and homogeneous network performance requirements.

Largely as a product of its own success, limitations of the Internet architecture resulting from these assumptions are being encountered as it is applied to more demanding applications and environments [3]. Examples of non-homogeneous network use polices are easy to find in the Internet today.

The inability to provide differentiated services has become a stumbling block to realizing the commercial potential of Internet technologies. Commercial Internet Services Providers (ISPs) would benefit from the ability to provide different levels of services (e.g. bronze, silver, gold, etc.), and a suite of service options (e.g. on-demand video or audio, interactive video or audio conferencing, etc.) that would allow them to extract additional revenue from existing infrastructure.

Non-commercial application of similar capabilities would enable the management of network resources. For example, portions of a network could be allocated along departmental (e.g. accounting, engineering, sales, etc.), functional (e.g. instruction vs. research), and use (e.g. video, audio, web, e-mail, etc) lines. Such service differentiation and resource management capabilities are not, in general, possible in the single forwarding class communications model used in the Internet today.

As a special case of service differentiation, the issues of security and trust have become critical for many modern applications of Internet technology. While security was important in the design of the Internet architecture, its implementation and deployment took lower priority to the implementation and deployment of the basic technology for what was still a very proof-of-concept communications architecture. More recent work (e.g. SSL [2] and SSH [21]) has focused on application-level, end-to-end security. This has left network layer security and trust largely unresolved.

In general, security and trust in the network layer revolve around questions of who can see traffic as it traverses an internet, and who can generate traffic load targeted at some point in an internet. The former represents a disclosure threat even for end-to-end protected traffic where traffic analysis may result in significant disclosures. The latter represents a critical denial-of-service threat as has been demonstrated in the many large-scale DDoS attacks perpetrated in the Internet.

Given the single forwarding class communications model underlying the current Internet architecture, these vulnerabilities are fundamentally unresolvable. While end-to-end solutions, such as those mentioned above, can help mitigate the problems, the basic vulnerabilities remain.

Similarly, it is easy to find examples of non-homogeneous network performance requirements in the Internet today. While the minimum delay paths used in the best-effort communications model are well suited for the data services (e.g. e-mail, telnet, http, etc.) prevalent in the early Internet, they are seriously inadequate for new applications of Internet technologies.

For example, the on-demand delivery of isochronous streams of data (i.e. data requiring delivery within specific time constraints, such as video and audio) requires low delay variance (called jitter), while the interactive delivery of isochronous data (e.g. Internet telephony) requires both low delay and low delay variance. Similarly, the delivery of streaming video requires high bandwidth, and is relatively loss-tolerant, while streaming audio requires relatively low bandwidth, but is loss-intolerant.

Due to the single-class forwarding model used in the Internet architecture, only one of such a set of diverse service models can be effectively supported in an internet today. While some service models satisfy the requirements of others (e.g. a high-bandwidth, low delay, and low delay variance model can satisfy the requirements of both video and audio conferencing) it will not utilize the network resources as effectively as a set of custom service models.

In summary, the assumptions of homogeneous network usage policies and performance requirements significantly limit the effectiveness of Internet technologies in supporting the demanding communication applications to which these technologies are being applied. This is the result of a new set of requirements being made of the Internet architecture by new applications.

1.3 New Requirements

In general, these new requirements involve the need for policy-based control of the forwarding topologies used for different classes of traffic. This need manifests itself in two distinct forms called traffic engineering, and quality-of-service (QoS) routing. *Traffic engineering* involves the constraint of the routing and forwarding functions to satisfy administrative policies. Interested parties (e.g. network management, users, etc.) define administrative policies specifying what classes of traffic can flow over what portions of an internet. Enhanced path selection algorithms then compute routes that honor the administrative policy constraints, and enhanced forwarding mechanisms are used to forward traffic over, in general, multiple paths per destination.

QoS routing involves the constraint of the routing and forwarding functions to satisfy performance requirements of specific traffic classes. Again, interested parties (users, network management, etc.) specify performance requirements for different traffic classes, and enhanced path selection algorithms and forwarding mechanisms are used to compute routes for, and forward traffic over paths that satisfy the requirements.

1.4 Previous Solutions

A number of solutions have been developed over the years to attempt to satisfy these new requirements for policy-based routing and forwarding. *Firewalls* implement (among other services) a rudimentary form of traffic engineering in the sense that they limit what traffic is allowed to flow over a protected subset of an internet. *Overlay networks* provide (again, among other services) a quasi-static form of QoS routing where alternatives to the currently selected network layer route to a given destination may be found in an overlay network that provide better performance than the current route for certain applications.

On-demand routing attempts to provide a comprehensive solution by enhancing the network-layer routing and forwarding mechanisms to support policy-based routing. In general, on-demand routing works by a route request-compute-setup process where route computations are requested with a set of administrative and performance constraints, route computations are performed which compute a path that satisfies the given constraints, and the path is then setup, typically using label-swap mechanisms [11], which allows traffic from multiple flows to be forwarded over potentially multiple paths to the same destination.

The previously proposed solutions discussed above are *less robust* due to their use of centralized control of state. For example, the forwarding paths in on-demand routing are brittle because the ingress router controls remote forwarding state in routers along paths it has set up. Similarly, the tunnels comprising an overlay network are brittle due to the centralized management of remote tunnel state in routers composing the overlay network.

Furthermore, these solutions are *less efficient* and *responsive* due to their use of centralized control of state, and requirement of overly complex mechanisms for implementing some functions. Specifically, due to its centralized nature, on-demand routing requires the use of duplex control communication to respond to topology changes in an internet. Additionally, on-demand routing and firewalls are less efficient and responsive due to their requirement of complex mechanisms to implement their functionality. On-demand routing requires the use of full-topology routing algorithms to compute optimal paths to any destination in an internet. In addition, on-demand routing requires the use of more complex state management mechanisms, such as soft-state timers and repair mechanisms to manage forwarding state. Similarly, firewalls impose heavy processing overhead on traffic to implement their traffic engineering functionality. Lastly, existing policy-based path selection algorithms, required by on-demand routing are computationally expensive and functionally incomplete.

Finally, the proposed solutions are *less effective* than existing Internet solutions due to the "wrong-sizing" of the implementation of their functionality. Specifically, overlay networks attempt to compute optimal routes with only partial visibility of the network topology, and firewalls attempt to implement policy-based forwarding control from only a single point in an internet.

In summary, the proposed solutions for policy-based routing and forwarding incur significant penalties in robustness, efficiency, reliability, and effectiveness relative to the existing Internet architecture. These costs can be largely accounted for by the violation of a number of design principles underlying the Internet architecture. In the remainder of this paper we present a new proposal for providing policy-based routing and forwarding that addresses the shortcomings of these previous proposals. We call the new architecture network-layer enclaves.

2 Network Layer Enclaves

Network Layer Enclaves (NLEs) are an enhancement to the Internet architecture that provide a solution for policy-based management of network resources without sacrificing the robust, efficient, responsive, and effective properties of the existing architecture discussed above. While maintaining the original architectural goal of supporting the development and integration of diverse network technologies, NLEs expand this goal to include supporting the exploitation of special capabilities of available network technologies and opportunities available in specific circumstances. Similarly, NLEs expand the original architecture's strategy of making minimal assumptions of network technologies in the forwarding plane by maintaining detailed knowledge of the special capabilities of available network technologies in the routing plane.

NLEs achieve these goals by allowing network resources to be assigned to one or more *enclaves*. Administrative policies can then be defined for each enclave specifying what traffic is allowed in the enclave. New, efficient policy-based path-selection algorithms are then used by the routing protocols to pre-compute paths than honor these policies for all destinations and enclaves. These new path selection algorithms can, optionally, compute routes satisfying QoS performance constraints within the enclaves. The resulting routing tables will, in general, contain multiple routes per destination, representing different paths which satisfy different combinations of enclave administrative policies and QoS performance requirements.

To support distributed forwarding over multiple paths per destination, NLEs define a new forwarding architecture called Distributed Label-Swap Forwarding (DLSF). In the DLSF architecture, traditional label-swap forwarding is used, but under the distributed control of the policy-based routing processes described above.

The resulting system implements a fully distributed routing computation that pre-computes routes for every destination, administrative policy, and, optionally, unique set of performance characteristics available in an internet. Forwarding over these routes is then performed by a simple and efficient label-swap data plane where control of the forwarding state is fully distributed in the sense that forwarding state is always controlled by a collocated routing process.

The resulting routing architecture can be seen as analogous to the Reduced Instruction Set Computer (RISC) processor architecture in which researchers shifted much of the intelligence for managing the use of processor resources to the compilers that were able to bring a higher-level perspective to the task, thus allowing much more efficient use of the physical resources, as well as freeing the hardware designers to focus on performance issues of much simpler processor architectures. Similarly, the communications architecture proposed here requires a shift in intelligence for customized (i.e. policy-based) path composition to the routing protocols and frees the network layer to focus solely on hop-by-hop forwarding issues, adding degrees of freedom to the network hardware engineering problem that, hopefully, allow for significant advances in the performance and effectiveness of network infrastructure.

The remainder of this paper presents a design for implementing the Network Layer Enclave architecture in the

Internet. Section 3 presents new policy-based routing algorithms that efficiently compute routes which honor traffic engineering and QoS constraints. Section 4 presents the distributed label-swap forwarding architecture which supports the efficient forwarding of traffic over multiple routes per destination. And Section 5 presents conclusions.

3 Efficient Policy-Based Routing

In this paper, *policy-based* routing is defined as the inclusion of multiple metrics in a routing computation [10, 23]. Policy-based routing supports *traffic engineering* by the computation of routes in the context of administrative constraints on the type of traffic allowed over portions of an internet. Analogously, policy-based routing supports *quality-of-service* (QoS) by the computation of routes in the context of performance-related constraints on the paths specific traffic flows are allowed to use.

The metrics used in routing computations are assigned to individual links in the network. For a given routing application, a set of link metrics is identified for use in computing the path metrics used in the routing decision. Link metrics can be assigned to one of two classes based on how they are combined into path metrics. *Concave (or minmax) metrics* are link metrics for which the minimum (or maximum) value (called the bottleneck value) of a set of link metrics defines the path metric of a path composed of the given set of links. Examples of concave metrics include residual bandwidth, residual buffer space, and administrative traffic constraints. *Additive metrics* are link metrics for which the sum (or product, which can be converted to a sum of logarithms) of a set of link metrics defines the path metric of the path composed of the given set of links. Examples of additive metrics include delay, delay jitter, cost, and reliability.

While, in general, routing with multiple constraints is an NP-complete problem [13, 14], there are many subclasses of this general problem that have been shown to have polynomial-time solutions. For example, any problem involving two metrics with at least one of them being concave can be solved in polynomial-time by a traditional shortest path algorithm on the graph in which all links that do not comply with the concave constraints have been pruned [10, 15, 23]. However, even for this case, as the number of constraints becomes exponential in the size of the graph, this result no longer holds.

The foundational work on the problem of computing routes in the context of more than one additive metric was done by Jaffe [14], who defined the multiply-constrained path problem (MCP) as the computation of routes in the context of two additive metrics. He presented an enhanced distributed Bellman-Ford algorithm that solved this problem with time complexity of $O(n^4 W \log(nW))$ (where n is the number of nodes in a graph, and W is the largest possible metric value). Since Jaffe, a number of solutions have been proposed for computing exact routes in the context of multiple metrics for special situations. Wang and Crowcroft [23] were the first to present the solution to computing routes in the context of a concave and an additive metric discussed above. Ma and Steenkist [16] presented a modified Bellman-Ford algorithm that computes paths satisfying delay, delay-jitter, and buffer space constraints in the context of weighted-fair-queuing scheduling algorithms in polynomial time. Cavendish and Gerla [5] presented a modified Bellman-Ford algorithm with complexity of $O(n^3)$ which computes multi-constrained paths if all metrics of paths in an internet are either non-decreasing or non-increasing as a function of the hop count. Recent work by Siachalou and Georgiadis [19] on MCP has resulted in an algorithm with complexity $O(nW \log(n))$. This algorithm is similar to the QoS algorithm presented in Section 3.3 in that it is an enhanced version of the Dijkstra algorithm based on invariants similar to those underlying the algorithms presented in Sections 3.2 and 3.3. However, due to errors in the algorithm, it does not compute correct results.

Several other algorithms have been proposed for computing approximate solutions to the QoS routing problem. Both Jaffe [14] and Chen and Nahrstedt [10] propose algorithms which map a subset of the metrics comprising a link weight to a reduced range, and show that using such solutions the cost of a policy-based path computation can be controlled at the expense of the accuracy of the selected routes. Similarly, a number of researchers [14, 17] have presented algorithms which compute routes based on a function of the multiple metrics comprising a link weight. These approximation solutions do not work with administrative traffic constraints.

In summary, the drawbacks of the current policy-based routing solutions are that they have poor average case performance, they implement inflexible routing models, and solutions for computing approximate solutions do not work with the traffic constraints used for traffic engineering.

3.1 Network Model

In this paper a network is modeled as a weighted undirected graph $G = (N, E)$, where N and E are the node and edge sets, respectively. By convention, the size of these sets are given by $n = | N |$ and $m = | E |$. Elements of E are unordered pairs of distinct nodes in N. $A(i)$ is the set of edges adjacent to i in the graph. Each link $(i, j) \in E$ is assigned a weight, denoted by ω_{ij}. A *path* is a sequence of nodes $< x_1, x_2, \ldots, x_d >$ such that

$(x_i, x_{i+1}) \in E$ for every $i = 1, 2, \ldots, d-1$, and all nodes the path are distinct. The weight of a path is given by $\omega_p = \sum_{i=1}^{d-1} \omega_{x_i x_{i+1}}$. The nature of these weights, and the functions used to combine these link weights into path weights are specified for each algorithm.

In the following, we propose a *declarative* traffic engineering model where network links are labeled with statements declaring *what* the desired routing policies are in the form of constraints of the traffic allowed on each link. These constraints take the form of *link expressions* in a boolean *traffic algebra* which describe the traffic allowed on a link. New, efficient policy-based routing algorithms then compute a minimal set of routes, composed of a *path expression* and a next hop, for each destination in an internet. These algorithms, in effect, *discover* the optimal set of forwarding classes needed at a given source in the internet to implement the desired policies. These path expressions are then installed in the appropriate traffic classifiers.

The traffic algebra is a boolean algebra used to define traffic classes in a flexible and efficient way. Specifically, it is composed of the standard boolean operations on the set $\{0, 1\}$, where p primitive propositions (variables) are true/false statements describing characteristics of network traffic. The syntax for expressions in the algebra is specified by the BNF grammar:

$$\begin{aligned}\varphi \ ::= \ & 0 \mid 1 \mid v_1 \ldots v_p \mid (\neg\varphi) \mid (\varphi \wedge \varphi) \mid \\ & (\varphi \vee \varphi) \mid (\varphi \rightarrow \varphi) \mid SAT(\varphi)\end{aligned}$$

The set of primitive propositions, indicated by v_i in the grammar, can be defined in terms of any globally significant attributes of the ingress router's state that can be expressed as a true/false statement. Link expressions identify the traffic classes allowed to traverse the link, and are denoted by ϵ_{ij} in the algorithms. Path expressions, denoted by ϵ_p in the algorithms, and defined as $\epsilon_p = \epsilon_{x_1 x_2} \wedge \epsilon_{x_2 x_3} \wedge \ldots \wedge \epsilon_{x_{d-1} x_d}$, specify the set of traffic classes allowed to traverse the path. There is a maximum of 2^p unique sets of traffic classes.

The $SAT(\varphi)$ primitive of the traffic algebra is the SATISFIABILITY problem of traditional Boolean algebra. Satisfiability must be tested in two situations by the algorithms presented below that implement traffic-engineering computations. First, an extension to a known route should only be considered if classes of traffic exist that are authorized to use both the path represented by the known route and the link used to extend the path (at line 15 in Figure 2). This is true *iff* the conjunction of these expressions is satisfiable (i.e. $SAT(\epsilon_i \wedge \epsilon_{ij})$). Second, given that classes of traffic exist that are authorized to use a path represented by a new route, the algorithms must determine whether all traffic supported by that route has also been satisfied by other, known shorter routes (not shown

P	$\equiv$	Queue of permanent routes to all nodes.
P_n	$\equiv$	Queue of permanent routes to node n.
T	$\equiv$	Heap of temporary routes.
T_n	$\equiv$	Entry in T for node n.
B_n	$\equiv$	Balanced tree of routes for node n.
$\mathcal{E}_n$	$\equiv$	Summary of traffic expression for all routes in P_n.

Table 1: Notation.

in the algorithms presented in this paper). This is true *iff* the new route's traffic expression implies the disjunction of the traffic expressions for all known better routes (i.e. $(\epsilon_i \rightarrow \epsilon_{i_1}, \epsilon_{i_2}, ..)$ is *valid*, which is denoted by $(\epsilon_i \rightarrow \mathcal{E}_i)$ in the algorithms). Determining if an expression is valid is equivalent to determining if the negation of the expression is unsatisfiable. Therefore the expressions at lines 10 and 13, of the form $\epsilon_1 \rightarrow \epsilon_2$ are equivalent to $\neg SAT(\neg(\epsilon_1 \rightarrow \epsilon_2))$ (or $\neg SAT(\epsilon_1 \wedge \neg\epsilon_2)$).

The satisfiability decision performed by $SAT(\epsilon)$ is the prototypical NP-complete problem [13]. As is typical with NP-complete problems, it has many restricted versions that are computable in polynomial time. An analysis of strategies for defining computationally tractable traffic algebras is beyond the scope of this paper, however we have implemented an efficient, restricted solution to the SAT problem by implementing the traffic algebra as a set algebra with the set operations of intersection, union, and complement on the set of all possible forwarding classes.

The routing algorithms presented here are based on an enhanced version of the path algebra defined by Sobrinho [20], which supports the computation of a set of routes for a given destination containing the "best" set of routes for each destination. Formally, the path algebra $P = < \mathcal{W}, \oplus, \preceq, \sqsubseteq, \bar{0}, \bar{\infty} >$ is defined as a set of weights $\mathcal{W}$, with a binary operator $\oplus$, and two order relations, $\preceq$ and $\sqsubseteq$, defined on $\mathcal{W}$. There are two distinguished weights in $\mathcal{W}$, $\bar{0}$ and $\bar{\infty}$, representing the least and absorptive elements of $\mathcal{W}$, respectively. $\oplus$ is the original path composition operator, and $\preceq$ is the original total ordering from [20]. $\oplus$ is used to compute path weights from link weights. $\preceq$ is used by the routing algorithm to build the forwarding set, starting with the minimal element, and by the forwarding process to select the minimal element of the forwarding set whose parameters satisfy a given QoS request.

A new relation on routes, $\sqsubseteq$, is added to the algebra and used to define classes of comparable routes and select maximal elements of these classes for inclusion in the set of forwarding entries for a given destination. $\sqsubseteq$ is a partial ordering (reflexive, anti-symmetric, and transitive) with the following, additional property:

Property 1 $(\omega_x \sqsubseteq \omega_y) \Rightarrow (\omega_x \succeq \omega_y)$.

A route r_m is a *maximal element* of a set R of routes in a graph if the only element $r \in R$ where $r_m \sqsubseteq r$ is r_m itself. A set R_m of routes is a *maximal subset* of R if, for all $r \in R$ either $r \notin R_m$, or $r \in R_m$ and for all $s \in R - \{r\}$, $r \not\sqsubseteq s$. The maximum size of a maximal subset of routes is the smallest range of the components of the weights (for the two component weights considered here). An example path algebra based on weights composed of delay and cost is as follows:

$$\begin{aligned}
\omega_i &\equiv (d_i, c_i) \\
\overline{0} &\equiv (0,0) \\
\overline{\infty} &\equiv (\infty, \infty) \\
\omega_i \oplus \omega_j &\equiv (d_i + d_j, c_i + c_j) \\
\omega_i \preceq \omega_j &\equiv (d_i < d_j) \vee ((d_i = d_j) \wedge (c_i \leq c_j)) \\
\omega_i \sqsubseteq \omega_j &\equiv (d_j \leq d_i) \wedge (c_j \leq c_i)
\end{aligned}$$

3.2 Basic Algorithms

The notation used in the algorithms presented in this paper is summarized in Table 1. In addition, the maximum number of unique truth assignments is denoted by $A = 2^p$, the maximum number of unique weights by $W = \min(\mathit{range\ of\ weight\ components})$, and the maximum number of adjacent neighbors by $a_{max} = \max\{| A(i) | \mid i \in N\}$. Table 2 defines the primitive operations for queues, heaps, and balanced trees used in the algorithms, and gives their time complexity used in the complexity analysis of the algorithms.

Notation	Description
Queue	
$Push(r,Q)$	Insert record r at tail of queue Q ($O(1)$)
$Head(Q)$	Return record at head of queue Q ($O(1)$)
$Pop(Q)$	Delete record at head of queue Q ($O(1)$)
$PopTail(Q)$	Delete record at tail of queue Q ($O(1)$)
d-Heap	
$Insert(r,H)$	Insert record r in heap H ($O(\log_d(n))$)
$IncreaseKey(r,r_h)$	Replace record r_h in heap with record r having greater key value ($O(d\log_d(n))$)
$DecreaseKey(r,r_h)$	Replace record r_h in heap with record r having lesser key value ($O(\log_d(n))$)
$Min(H)$	Return record in heap H with smallest key value ($O(1)$)
$DeleteMin(H)$	Delete record in heap H with smallest key value ($O(d\log_d(n))$)
$Delete(r_h)$	Delete record r_h from heap ($O(d\log_d(n))$)
Balanced Tree	
$Insert(r,B)$	Insert record r in tree B ($O(\log(n))$)
$Min(B)$	Return record in tree B with smallest key value ($O(\log(n))$)
$DeleteMin(B)$	Delete record in tree B with smallest key value ($O(\log(n))$)

Table 2: Operations on Data Structures [1].

The algorithms presented in this section are based on the data structure model shown in Figure 1. In this structure, a balanced tree (B_i) is maintained for each node in the graph to hold newly discovered, temporary labeled routes for that node. The heap T contains the lightest weight entry from each non-empty B_i (for a maximum of n entries). Lastly, a queue, P_i, is maintained for each node which contains the set of permanently labeled routes discovered by the algorithm, in the order in which they are discovered (which will be in increasing weight). The general flow of these algorithms will be to take the minimum entry from the heap T, compare it with existing routes in the appropriate P_i, if it is incomparable with existing routes in P_i it is pushed onto P_i, and "relaxed" routes for its neighbors are added to the appropriate B_x's.

The correctness of these algorithms is based on the maintenance of the following three invariants: for all routes $I \in P$ and $J \in B_*$, $I \preceq J$, all routes to a given destination i in P are incomparable for some set of satisfying truth assignments, and the maximal subset of routes to a given destination in $P \cup B_*$ represents the maximal subset of all paths to j using nodes with routes in P. Furthermore, these invariants are maintained by the following two constraints on actions performed in each iteration of these algorithms: (1) only known-non-maximal routes are deleted or discarded, and (2) only the smallest known-maximal route is moved to P.

Due to space constraints, only the general algorithm that computes routes in the context of both link predicates and multiple metrics is presented here.

Figure 2 presents a modified Dijkstra algorithm that computes an optimal set of routes to each destination subject to multiple general (additive or concave) path metrics, in the presence of traffic constraints on the links. The time complexity of Policy-Based-Dijkstra is dominated by the loops at lines 4, 11, and 15. The loop at line 4 is executed nWA times, and the loop at line 15 mWA times. The loop at line 11 scans the entries in P_i to verify a new route is best for some truth assignment. For a given destination, this loop is executed at most an incrementally increasing number of times, starting at 0 and growing to $WA - 1$ (the maximum number of unique routes to a given destination) for a total of $\sum_{i=1}^{WA-1} i = \frac{(WA-1)WA}{2}$ times. For completeness, the statements at lines 6 and 21 take time proportional to $\log(a_{max}WA)$ for a total of $nWA\log(a_{max}WA)$ and $mWA\log(a_{max}WA)$, respectively; and those in lines 7–9 and 17–20 proportional to $\log_d(n)$ for a total of $nWA\log_d(n)$ and $mWA\log_d(n)$, respectively. Therefore, the worst case time complexity of Policy-Based-Dijkstra, dominated by the loop at line 11, is $O(nW^2A^2)$.

The loop at line 11, which dominates the cost of Policy-

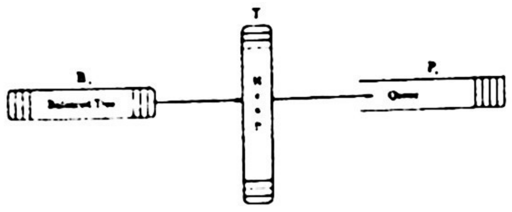

Figure 1: Model of Data structures for Basic Algorithms

```
algorithm Policy-Based-Dijkstra
    begin
1   $Push(<s, s, \bar{0}, 1>, P_s)$;
2   for each $\{(s, j) \in A(s)\}$
3       $Insert(<j, s, \omega_{sj}, c_{sj}>, T)$;
4   while $(|T| = 0)$
        begin
5       $<i, p_i, \omega_i, c_i> \leftarrow Min(T)$;
6       $DeleteMin(B_i)$;
7       if $(|B_i| = 0)$
8           then $DeleteMin(T)$
9           else $IncreaseKey(Min(B_i), T_i)$;
10      $c_{tmp} \leftarrow c_i$; $ptr \leftarrow Tail(P_i)$;
11      while $((c_{tmp} \neq 0) \wedge (ptr \neq \emptyset))$
12          $c_{tmp} \leftarrow c_{tmp} \wedge \neg ptr.c$; $ptr \leftarrow ptr.next$;
13      if $(c_{tmp} \neq 0)$
            then begin
14              $Push(<i, p_i, \omega_i, c_i>, P_i)$;
15              for each $\{(i, j) \in A(i) \mid SAT(c_i \wedge c_{ij})\}$
                    begin
16                  $\omega_j \leftarrow \omega_i \oplus \omega_{ij}$; $c_j \leftarrow c_{ij}$;
17                  if $(T_j = \emptyset)$
18                      then $Insert(<j, i, \omega_j, c_j>, T)$
19                  else if $(\omega_j \prec T_j.\omega)$
20                      then $DecreaseKey(<j, i, \omega_j, c_j>, T)$;
21                  $Insert(<j, i, \omega_j, c_j>, B_j)$;
                    end
            end
        end
    end
```

Figure 2: General-Policy-Based Dijkstra.

Based-Dijkstra, is required because there is no way to summarize the permanent routes for a destination. However, for the traffic engineering and QoS variants of this algorithm, the permanent routes can be summarized by a summary traffic expression (formed by the disjunction of permanent route path expressions) and the weight of the last route, respectively. Using these shortcuts, the complexity of the traffic engineering and QoS algorithms are $O(mA\log(A))$ and $O(mW\log(W))$, respectively.

3.3 Enhanced Algorithms

The $\log(A)$ and $\log(W)$ factors in the complexity of the traffic engineering and QoS variants of the Policy-Based-Dijkstra algorithm (respectively) are the result of the use of a balanced tree for storing the temporarily labeled nodes for a given destination. This section presents enhanced versions of these algorithms which use a queue-based data structure for this purpose, reducing the cost of managing these structures to a lower order term in the time complexity. As a result the runtime cost of the enhanced algorithms becomes dominated by $\log_d(n)$ factors from the manipulation of the T heap.

This enhancement is based on the property that routes to a given node *with the same predecessor* are discovered in strictly increasing (or non-decreasing, depending on the algorithm) order. This property is a result of the fact that routes to a given predecessor will be discovered in strictly increasing (non-decreasing) order, and therefore the order of discovery of routes from a given predecessor to one of its neighbors will have the same property.

Based on this insight, the data structure shown in Figure 3 can be used to improve the performance of the algorithms presented in Section 3.2. In this data structure the balanced trees for each node are replaced with a set of queues for each neighbor of the node, and a summary heap containing the head of each neighbor queue. Exploiting the ordering property of these queues, the algorithms ensure that each node head H_i, and therefore T_i, contain the lightest route in the link queues that is not subsumed by the routes in P_i. Due to space constraints, only the QoS version of these algorithms is presented here.

Figure 11 presents the enhanced version of the TD-QoS-Dijkstra algorithm. Similar to the basic algorithms, the correctness of these algorithms is based on the invariants and constraints presented in Section 3.2. Specifically, as detailed in the comments from that section, constraints 1 and 2 are maintained by the DeleteTMin() and AddCandidate() functions, and, based on this, the Dijkstra iteration over the n^{th} best route in the main body of the algorithm maintains the invariants.

The runtime complexity of the TD-QoS-Dijkstra algorithm (again, ignoring the cost for determining satisfiability) is dominated by the loops at lines 6 and 10. The loop at line 6 is executed at most once for each incomparable path to each node in the graph for a total of nW times. The loop at line 10 is executed at most once for each distinct instance of an edge in the graph, for a total of mW times. The most costly operation in the loop at line 6 is the DeleteTMin() call at line 9. In the DeleteTMin() routine, the loop at line 7 will be executed, in total, at most once per neighbor for each forwarding class for a total of $a_{max}W$, and the cost per call of the heap operations at lines 13 and 14 is $d\log_D(n)$. Therefore, the total worst-case cost of the call at line 8 of the main algorithm is $nW\log_d(n) + a_{max}W$. In the AddCandidate() routine, the runtime complexity is dominated by the heap operations at lines 5, 20, and 23, which cost $\log_d(n)$ each, for a total cost of the call to AddCandidate() at line 12 of the main algorithm of $mW\log_d(n)$. Therefore, the worst-case time complexity of the enhanced TD-QoS-Dijkstra algorithm is $O(mW\log(n))$.

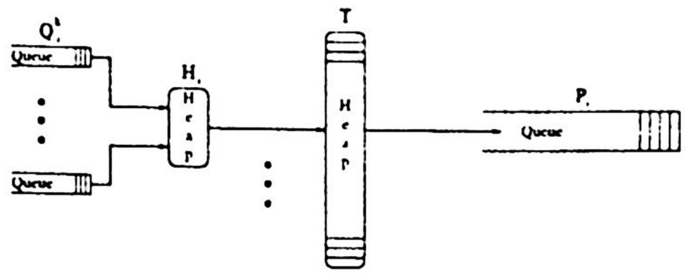

Figure 3: Model of Data Structures for Enhanced Algorithms

Experiments were run on the TD-QoS-Dijkstra algorithm on a 1GHz Intel Pentium 3 based system. The algorithms were implemented using the C++ Standard Template Library (STL) and the Boost Graph Library. Each test involved running the algorithm on ten random weight assignments to ten randomly generated graphs (generated using the GT-ITM package [24]). For each test the worst case measurements are graphed. The metrics were generated using the "Cost 2" scheme from [19] where the delay component is randomly selected in the range $1..MaxMetric$, and the cost component is computed as $cost = \sigma(MaxMetric - delay)$, where σ is a random integer in the range 1..5; this scheme was chosen as it proved to result in the most challenging computations from a number of different schemes considered. The QoS routing problem was used for these tests as it was easiest to generate meaningful random metric assignments for. Space overhead was measured in terms of the maximum number of entries stored in the B_* structures.

Tests were run for performance (both runtime and space) as a function of graph size, average degree of the graph, and the maximum link metric value. Due to space constraints, only the graphs for size are shown here. Also, since the maximum metric was shown to have little impact on performance, only results for tests with a maximum metric of 1000 are presented here. Figures 4 and 5 compare the performance of the basic QoS (not presented in this paper), enhanced QoS, and traditional (single path) Dijkstra algorithms. They show that, while costs increase with both graph size and average degree, both the magnitude of these costs and their growth rate are very manageable. While runtime grows to approximately 2 seconds for the largest problems, for graphs smaller than 500 nodes with an average degree of 8 (well beyond the scale supportable by current Internet routing protocols) the runtime is at most a few hundred milliseconds, and the growth rate is barely beyond linear in this range of parameters. Similarly, the worst-case space utilization stays below 30,000 entries (consuming less than 10MB of memory) with similar growth rates.

In summary, the results showed: excellent runtime performance in the range of parameters expected from routing domains in the Internet (i.e. average degree less than 5, and routing graph size less than 500); the algorithms exhibited very well-behaved growth rates; and they exhibited very reasonable space overhead in all scenarios.

4 Distributed Label-Swap Forwarding

The policy-based routing algorithms presented in Sections 3.2 and 3.3 compute multiple routes to the same destination to satisfy the policy requirements of an internet. Such routes are not supported by current, host-address-based packet forwarding mechanisms that only allow one route per destination. The solution to this problem is to use label-swapping technology (e.g., MPLS [11]) as a generalized forwarding mechanism that replaces IP addresses as the names for network attachment points in the route binding function with arbitrary labels which can be defined by the routing protocol to represent any policy/destination pair for which a route has been computed.

A significant innovation of the policy-based routing architecture presented here is the combination of a table-driven, hop-by-hop routing model with label-swap forwarding mechanisms. Traditionally, label-swap forwarding has only been seen as an appropriate match with an on-demand, source-driven routing model. Indeed, to a large extent, the virtual-circuit nature of these previous solutions has been attributed to their use of label-swap forwarding. Contrary to this view, the position taken here is that host addresses and labels are largely equivalent alternatives for representing forwarding state, and that the virtual-circuit nature of prior architectures derives from their use of a source-driven forwarding model. The primary conceptual difference between address and label-swap forwarding is that label-swap forwarding provides a clean separation of the control and forwarding planes [22] within the network layer, where address-based forwarding ties the two planes together. This separation provides what might be called a *topological anonymity* of

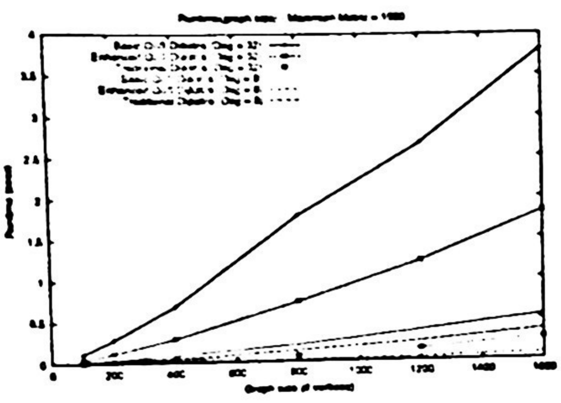
Figure 4: Compare Runtime(Size)

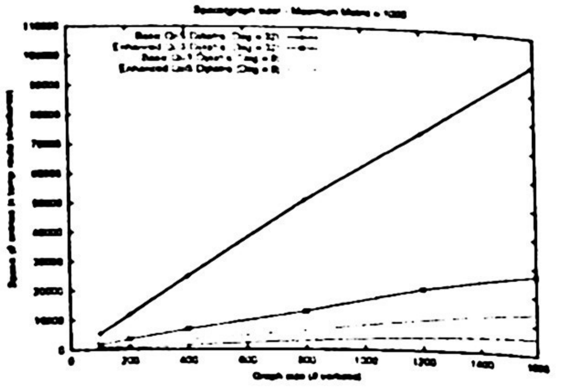
Figure 5: Compare Space(Size)

the forwarding plane that is critical to the implementation of policy-based routes. Chandranmenon and Varghese [9] present a similar notion, which they call *threaded indices*, where neighboring routers share the indexes into their routing tables for specific routes which are then included in forwarded packets to allow rapid forwarding table lookups. In addition they present a modified Bellman-Ford algorithm that exchanges these labels among neighbors. Our solution generalizes the threaded index concept to use generic labels (with no direct forwarding table semantics), uses these labels to represent routing policies computed by the routing protocols, and defines a family of routing protocols to exchange local labels among neighbors.

As illustrated in Figure 6, label-swap forwarding can be used in the context of traditional address-based forwarding. In this example the forwarding table is referenced for both traffic classification (through the "address prefix" field), and for label-swap forwarding (through the "local label" field). The benefit of this mechanism for traffic forwarding is it can be generalized to handle policy-based forwarding. In addition, label-swap forwarding can be used to implement traffic engineering via the assignment of traffic to administrative classes which are used to select different paths for traffic to the same destination depending on the labeling of links in the network with administrative class sets. For example, Figure 7 shows a small network with four nodes, two administrative classes A and B, and the given forwarding state for reaching node 4. The benefits of this architecture are that it is based on forwarding state that is agnostic to the definition of forwarding classes, allowing the data forwarding plane to remain simple yet general; and it concentrates the path computation functions in the routing protocol, which is the least time critical, and most flexible component of the network layer.

The enhancement of traditional unicast routing systems with the policy-based routing technology presented above is straight-forward. The routing protocol must be enhanced to carry the additional link metrics required to implement the desired policies. This requires the use of either a link-state or link-vector routing protocol that exchanges information describing link state. As described earlier, by supporting the use of partial-topology, link-vector protocols this architecture supports much more efficient solutions than the on-demand model.

Forwarding state must be enhanced to include local and next hop label information in addition to the destination and next hop information existing in traditional forwarding tables. Traffic classifiers must be placed at the edge of an internet, where "edge" is defined to be any point from which traffic can be injected into the internet. Since each router represents a potential traffic source (for CLI and network management traffic), this effectively means a traffic classification component must be present in each router. As illustrated in Figure 8, the resulting traffic flow requirements are that all non-labeled traffic (sourced either from a router itself, or from a directly connected host or non-labeling router) must be passed through the traffic classifier first, and all labeled traffic (sourced either from the traffic classifier or a directly connected labeling router) must be passed to the label-swap forwarding process.

Lastly, the routing protocol must be enhanced to exchange information needed to compute the label swap components of its forwarding tables. The output of the routing algorithm is forwarding information described in terms of a destination, traffic expression, and path weight for each computed route. To be used for forwarding, this information must be augmented with local and next hop labels. To determine the next hop label for a given route the routing process requires the forwarding tables of its neighbors. Therefore, the final enhancement required of routing protocols is that they exchange local forwarding tables and use this information to compute the next hop label for their routes. One challenge presented by this re-

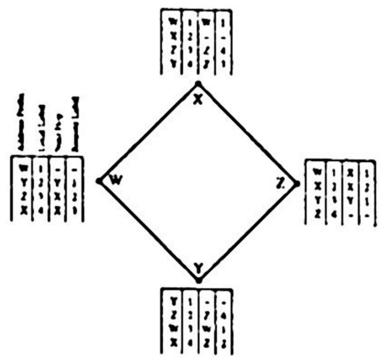
Figure 6: Labels with Address-Based Forwarding

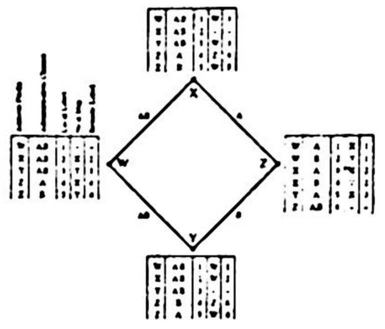
Figure 7: Labels with Policy-Based Forwarding

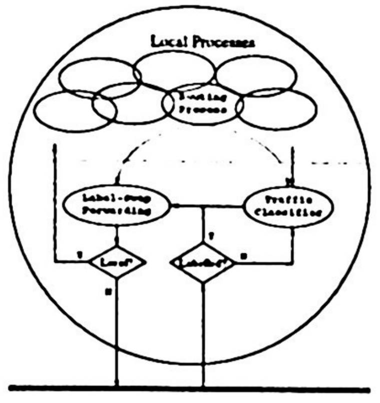

Figure 8: Traffic Flow in Policy-Enabled Router

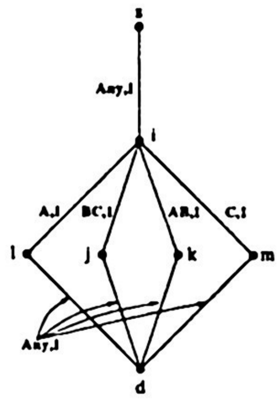

Figure 9: Next Hop Problem with Policy-Based Routing

quirement is that the routes computed by the routing algorithm must be assured of matching an active route in the selected next hop neighbor. As illustrated in Figure 9, this is not guaranteed by the algorithms presented above. Specifically, in this internet there are a number of equally "good" routes from nodes s and i to node d. For example, it is possible that the routing process at node i selects the paths through its neighbors l and j to provide two hop paths for traffic classes A, B, and C, while node s selects the paths that go through nodes k and m. In such a case there is no next hop label that can be chosen at s for routes to d that will satisfy the traffic policies.

To address this problem, Figure 10 presents an enhanced version of the basic traffic engineering algorithm for use in the context of hop-by-hop forwarding. In this algorithm, routes are augmented with two additional fields; n_d is the next hop neighbor for a route to destination d, and l_d is the next hop label for d. As described above, a partial forwarding table is maintained for each neighbor, specified by $F_n[d]$, containing an array of routes for each destination in the internet. Each entry in this array, denoted by $< d, \omega_d, \epsilon_d, l_d >$, gives the weight, traffic expression, and next hop label for each route in the neighbor's forwarding table. In this algorithm, new paths are only considered if they are extensions of paths chosen by the neighbor which is the next hop to the predecessor to the path's destination. For example, from Figure 9, node s will only consider paths to destination d that are extensions of node i's paths to d through nodes l and j. A fringe benefit of this enhancement is the next hop label computation can now be integrated with the routing computation (as shown by the inclusion of the next hop label in the routes computed by the algorithm).

5 Conclusions

This paper presents an enhancement to the Internet architecture called Network Layer Enclaves (NLEs). NLEs are the first proposed solution for providing policy-based management of traffic forwarding in the Internet that maintains its distributed, hop-by-hop routing architec-

```
algorithm Hop-by-Hop-TD-TE-Dijkstra
  begin
1   $Push(<s, s, 0, 1, s, \emptyset>, P_s)$;
2   for each $\{(s, j) \in A(s)\}$
3     $Insert(<j, s, \omega_{sj}, c_{sj}, j, \emptyset>, T)$;
4   while $(|T| = 0)$
      begin
5     $<i, p_i, \omega_i, c_i, n_i, l_i> \leftarrow Min(T)$;
6     $DeleteMin(B_i)$;
7     if $(|B_i| = 0)$
8       then $DeleteMin(T)$
9       else $IncreaseKey(Min(B_i), T_i)$;
10    if $(\neg(c_i \rightarrow \mathcal{E}_i))$
        then begin
11        $Push(<i, p_i, \omega_i, c_i>, P_i)$;
12        $\mathcal{E}_i \leftarrow \mathcal{E}_i \vee c_i$;
13        for each $\{(i, j) \in A(i) \mid$
              $(\exists <j, \omega'_j, c'_j, l'_j> \in F_{n_i}[j] \mid$
                $(c_{on_i} \wedge c'_j = c_i \wedge c_{ij}) \wedge$
                $(\omega_{on_i} + \omega'_j = \omega_i + \omega_{ij})) \wedge$
              $SAT(c_i \wedge c_{ij}) \wedge \neg((c_i \wedge c_{ij}) \rightarrow \mathcal{E}_j)\}$
          begin
14          $\omega_j \leftarrow \omega_i + \omega_{ij}$; $c_j \leftarrow c_i \wedge c_{ij}$;
15          if $(T_j = \emptyset)$
16            then $Insert(<j, i, \omega_j, c_j, n_i, l'_j>, T)$
17          else if $(\omega_j < T_j.\omega)$
18            then $DecreaseKey(<j, i, \omega_j, c_j, n_i, l'_j>, T)$;
19          $Insert(<j, i, \omega_j, c_j, n_i, l'_j>, B_j)$;
          end
        end
      end
  end
```

Figure 10: Hop-by-Hop TD-TE-Dijkstra.

ture. NLEs work by allowing network resources to be assigned to enclaves, and policies to be defined specifying what traffic is allowed in each enclave. A family of new, efficient policy-based path-selection algorithms, and a new forwarding model called Distributed Label-Swap Forwarding (DLSF) are presented which are needed to implement NLEs. The enhanced TD-TE-Dijkstra algorithm is the most efficient algorithm available for computing routes that satisfy traffic engineering requirements, and Policy-Based-Dijkstra is the first algorithm for computing routes that simultaneously satisfy traffic engineering and quality-of-service requirements. A traffic algebra is defined to formalize the notion of traffic constraints, and a set-based model is identified for efficiently implementing restricted but useful traffic engineering policies. The DLSF architecture efficiently implements multiple paths per destination required for hop-by-hop policy-based routing using label-swap-based forwarding, and a hop-by-hop version of the TD-TE-Dijkstra algorithm is presented for computing routes that can be implemented in this architecture.

The resulting system implements a fully distributed routing computation that pre-computes routes for every destination, administrative policy, and, optionally, unique set of performance characteristics available in an internet. Forwarding over these routes is then performed by a simple and efficient label-swap data plane where control of the forwarding state is fully distributed.

References

[1] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows – Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[2] Christopher Allen and Tim Dierks. The TLS Protocol Version 1.0. RFC 2246, January 1999.

[3] Marjory S. Blumenthal and David D. Clark. Rethinking the design of the Internet: The end to end arguments vs. the brave new world. *ACM Transactions on Internet Technology*, 1(1):70–109, August 2001.

[4] Brian E. Carpenter. Architectural Principles of the Internet. RFC 1958, June 1996.

[5] D. Cavendish and M. Gerla. Internet QoS Routing using the Bellman-Ford Algorithm. In *Proceedings IFIP Conference on High Performance Networking*. IFIP, 1998.

[6] Vinton G. Cerf. The Catenet Model for Internetworking. IEN 48, July 1978.

[7] Vinton G. Cerf and Edward Cain. The DoD Internet Architecture Model. *Computer Networks*, 7:307–318, 1983.

[8] Vinton G. Cerf and Robert E. Kahn. A Protocol for Packet Network Intercommunication. *IEEE Transactions on Communications*, COM-22(5):637–648, May 1974.

[9] Girish P. Chandranmenon and George Varghese. Trading Packet Headers for Packet Processing. *IEEE ACM Transactions on Networking*, 4(2):141–152, October 1995. 1995.

[10] Shigang Chen and Klara Nahrstedt. An Overview of Quality of Service Routing for Next-Generation High-Speed Networks: Problems and Solutions. *IEEE Network*, pages 64–79, November 1998.

[11] Bruce Davie and Yakov Rekhter. *MPLS: Technology and Applications*. Morgan Kaufmann, 2000.

[12] J.J. Garcia-Luna-Aceves and Jochen Behrens. Distributed, Scalable Routing Based on Vectors of Link States. *IEEE Journal on Selected Areas in Communications*, October 1995.

[13] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman & Co., 1979.

[14] Jeffrey M. Jaffe. Algorithms for Finding Paths with Multiple Constraints. *Networks*, 14(1):95–116, 1984.

[15] Whay C. Lee, Michael G. Hluchyi, and Pierre A. Humblet. Routing Subject to Quality of Service Constraints in Integrated Communication Networks. *IEEE Network*, 9(4):46–55, August 1995.

[16] Qingming Ma and Peter Steenkiste. Quality-of-Service Routing for Traffic with Performance Guarantees. In *Proceedings 4th International IFIP Workshop on QoS*. IFIP, May 1997.

[17] Piet Van Mieghem, Hans De Neve, and Fernando Kuipers. Hop-by-hop quality of service routing. *Computer Networks*, 37:407–423, November 2001.

[18] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-End Arguments in System Design. *ACM Trans. on Computer Systems*, 2(4):277–288, November 1984.

[19] Stavroula Siachalou and Leonidas Georgiadis. Efficient QoS Routing. In *Proceedigns of INFOCOM'03*. IEEE, April 2003.

[20] João Luís Sobrinho. Algebra and Algorithms for QoS Path Computation and Hop-by-Hop Routing in the Internet. *IEEE/ACM Transactions on Networking*, 10(4):541–550, August 2002.

[21] SSH Communications Security. http://www.ssh.com/.

[22] George Swallow. MPLS Advantages for Traffic Engineering. *IEEE Communications Magazine*, 37(12):54–57, December 1999.

[23] Zheng Wang and Jon Crowcroft. Quality-of-Service Routing for Supporting Multimedia Applications. *IEEE Journal on Selected Areas in Communications*, pages 1228–1234, September 1996.

[24] Ellen W. Zegura, Ken Calvert, and S. Bhattacharjee. How to Model an Internetwork. In *Proceedings INFOCOM '96*. IEEE, 1996.

```
algorithm TD-QoS-Dijkstra
    begin
1   $Push(<s, s, \bar{0}>,\ P_s)$;
2   for each $\{(s, j)\ \in\ A(s)\}$
      begin
3     $Push(<j, s, \omega_{sj}>,\ Q_j^s)$;
4     $Insert(<j, s, \omega_{sj}>,\ H_j)$;
5     $Insert(<j, s, \omega_{sj}>,\ T)$;
      end;
6   while $(|T| > 0)$
      begin
7     $<i, p, \omega> \leftarrow Min(T)$;
8     $Push(<i, p, \omega>,\ P_i)$;
9     DeleteTMin();
10    for each $\{(i, j)\ \in\ A(i)\}$
        begin
11      $\omega_i\ \leftarrow\ \omega \oplus \omega_{ij}$;
12      AddCandidate($<j, i, \omega_i>$);
        end
      end
    end
```

```
function QoS-DeleteTMin()
    // Delete minimum entry from $T$ and restore invariants:
    //   Constraint 1 - only deletes routes (line 9) that are
    //      $\sqsubseteq$ another route.
    //   Constraint 2 - loop at line 7 ensures new $T_i\ \not\sqsubseteq$
    //      new $Tail(P_i)$.
    begin
1   $<i, p, \omega> \leftarrow Min(T)$;
2   $Pop(Q_i^p)$;
3   if $(|Q_i^p| > 0)$
4     then $IncreaseKey(Head(Q_i^p),\ H_i^p)$
5     else $DeleteMin(H_i)$;
6   if $(|H_i| > 0)$
      then begin
        // Find smallest route in link queues that is not
        // $\sqsubseteq$ the deleted route.
7       for each $\{(i, k)\ \in\ A(i)\ |\ (|Q_i^k| > 0) \wedge$
                 $(Head(Q_i^k).\omega\ \sqsubseteq\ \omega)\}$
          begin
8         while $((|Q_i^k| > 0)\ \wedge\ (Head(Q_i^k).\omega\ \sqsubseteq\ \omega))$
9           $Pop(Q_i^k)$;
10        if $(|Q_i^k| > 0)$
11          then $IncreaseKey(Head(Q_i^k),\ H_i^k)$
12          else $Delete(H_i^k)$;
          end
13      if $(|H_i| > 0)$
          then $IncreaseKey(Min(H_i),\ T_i)$; return;
      end
14  $DeleteMin(T)$;
    end
```

```
function QoS-AddCandidate($<i, p, \omega_i>$)
    // Add new route to appropriate $Q$ and restore invariants:
    //   Constraint 1 - only drops known comparable routes
    //      (lines 1, 10, 15, and 24).
    //   Constraint 2 - ensures $Min(H_i)\ \preceq$ (and therefore $\not\sqsubseteq$)
    //      all routes in $Q_i^*$ queues.
    begin
1   if $(\omega_i\ \sqsubseteq\ Tail(P_i).\omega)$ then return;
2   if $(|H_i| =\ 0)$
      then begin
3       $Push(<i, p, \omega_i>,\ Q_i^p)$;
4       $Insert(<i, p, \omega_i>,\ H_i)$;
5       $Insert(<i, p, \omega_i>,\ T)$;
6       return;
        end
7   $<i, k, \omega_m> \leftarrow Min(H_i)$;
8   if $(\omega_m\ \preceq\ \omega_i)$
      then
9       if $(\omega_i\ \sqsubseteq\ \omega_m)$
10        then return;
11        else begin // $((\omega_i\ \not\sqsubseteq\ \omega_m)\ \wedge\ (\omega_m\ \preceq\ \omega_i))$
12          if $(|Q_i^p| =\ 0)$
13            then $Insert(<i, p, \omega_i>,\ H_i)$
14            else if $(\omega_i\ \sqsubseteq\ Tail(Q_i^p).\omega)$
15              then return;
16          $Push(<i, p, \omega_i>,\ Q_i^p)$;
            end
      else // $\omega_i \prec \omega_m$: since $\omega_i\ \succ\ Min(H_i^p)$, it must be
          // true that $|Q_i^p| =\ 0$.
17      if $(\omega_i\ \not\sqsupseteq\ \omega_m)$
          then begin
18          $Push(<i, p, \omega_i>,\ Q_i^p)$;
19          $Insert(<i, p, \omega_i>,\ H_i)$;
            // Following replaces $<i, k, \omega_m>$.
20          $DecreaseKey(<i, p, \omega_i>,\ T_i)$;
            end
          else begin // $(\omega_i\ \sqsupseteq\ \omega_m)$
21          $Push(<i, p, \omega_i>,\ Q_i^p)$;
            // Following replaces $<i, k, \omega_m>$.
22          $DecreaseKey(<i, p, \omega_i>,\ H_i^k)$;
23          $DecreaseKey(<i, p, \omega_i>,\ T_i)$;
24          $Pop(Q_i^k)$;
25          if $(|Q_i^k| > 0)$
26            then $Insert(Head(Q_i^k),\ H_i)$;
            end
    end
```

Figure 11: Enhanced QoS Dijkstra.