Smart Forwarding Technique for Routing with Multiple QoS Constraints

Aiguo Fei, Mario Gerla
Computer Science Department
University of California
Los Angeles, CA 90095

Abstract—QoS-constrained routing is considered as one of the key components to support quality of service in next-generation data networks. However, optimal routing problem subject to multiple constraints is NP-hard in general. In this paper we propose a technique called “smart forwarding” which can be used in both distributed hop-by-hop QoS routing and centralized source-based routing. It enables fast on-demand routing by utilizing a table pre-computed with link-state information or distributed Bellman-Ford algorithm. It can greatly reduce routing overhead in both flooding-based and crank-back routing protocols by only forwarding routing request to a neighbor that is known to be able or potentially be able to meet the QoS requirement. We also describe how we can adjust routing overhead by bounding the number of flooding or crank-back trials with this technique. More detailed analysis of this technique with simulation results which demonstrate that smart forwarding technique is effective in finding low-cost path while it has the property of being able to find a feasible solution if there is one.

I. INTRODUCTION

In recent years, there has been a major effort to enable QoS support in data networks: from ATM networks to more recent Integrated Services[3] and Differentiated Services[2] architectures for the Internet. QoS support is a multi-facet problem, it involves traffic classification and/or prioritizing, buffering and scheduling, resource reservation, signaling, routing, among others. QoS routing consists of finding paths subject to different often multiple QoS constraints to satisfy application requirements. Unfortunately optimal routing problem (for a given optimization objective such as minimizing administrative cost) with multiple constraints is often not solvable in polynomial time [12].

To avoid the complexity involved in seeking optimal exact solutions, a variety of heuristic strategies have been proposed and studied. Some heuristic algorithms attempt to find an approximate solution by considering one metric at a time for a generic multiple-metric routing problem[7]. One interesting approximate algorithm can obtain coarse and approximate solutions by solving a polynomial-time problem reduced from the original problem through mapping unbounded real number metrics of a link to bounded integers[5]. By assuming a particular scheduling discipline and traffic shaping policy at each router, sometimes the problem can also be reduced to be solvable in polynomial time[10, 14]. A survey of such algorithms can be found in [4]. Though most research work in this area has been in a theoretical setting, there is an effort to extend the current OSPF/Open Shortest Path First) routing protocol to support QoS[1].

Most routing approaches can be classified into two categories: centralized source-based and distributed hop-by-hop. In a source-based routing scheme, a source node computes the path based on available network state information and then may setup the path and reserve resource using a signaling protocol. In a distributed hop-by-hop routing scheme, a routing request is forwarded hop-by-hop until it reaches the destination or is dropped somewhere inside the network. From one viewpoint, any QoS routing scheme is essentially a technique to enumerate some or all possible routes from the source to the destination and find the one which satisfies all QoS constraints while minimizing path cost or maximizing network throughput. Flooding is the most effective routing scheme in the sense that it can probe all possible routes in parallel, but it is also the least efficient scheme since it generates significant overhead. Indeed recently several fine-tuned flooding-based algorithms were reported [6], [9], [11]. Among them, [6] does simple feasibility check at each hop while [9], [11] use hop-count to bound flooding scope.

In this paper we present a new technique which we call “smart forwarding”. This technique can be applied in a centralized source-based routing algorithm and also in a distributed hop-by-hop approach. The key idea of smart forwarding is that when constructing an end-to-end path, one can make some “smart” decision on which next hop should be in the path to satisfy the constraints and minimize the cost. It can be used as a crank-back approach in which it tries another path if one doesn’t work out. However it tries a path only if it knows there is a chance that given path might be feasible. The algorithm’s performance and processing overhead can be adjusted by setting limit on how many crank-back trials or forwarding branches it can have. It utilizes a pre-computed table (updated periodically) which can be constructed with low computation complexity and each on-demand routing request can then be answered with low processing overhead. When used as a flooding scheme without limiting crank-back trials, it can greatly reduce number of routing messages since routing request will only be sent to promising neighbors based on “smart” decisions.

The rest of this paper is organized as follows. Section 2 introduces classification of QoS metrics and definitions that provide the mathematical background for our smart-forwarding technique. Section 3 describes algorithms based on this technique, followed by simulation results in section 4. We briefly conclude our paper in section 5.

II. QoS METRICS CLASSIFICATION AND DEFINITIONS

Let $R_i$ denote a network node as in Fig. 1. Consider the concatenation of a path from $R_1$ to $R_2$ and a path from $R_2$ to $R_3$ to make up a path from $R_1$ to $R_3$. Consider a QoS metric,
say, $QM$, which is a function of the path: $QM(R_1 \rightarrow R_2)$, $QM(R_2 \rightarrow R_3)$ and $QM(R_1 \rightarrow R_3)$, where $R_i \rightarrow R_j$ denotes a specific path from $R_i$ to $R_j$, and $R_1 \rightarrow R_3$ refers to the path concatenated by paths $R_1 \rightarrow R_2$ and $R_2 \rightarrow R_3$. In general, $QM(R_i \rightarrow R_j)$ is a function of the complete path from $R_i$ to $R_j$. However, for some metric $QM$, $QM(R_i \rightarrow R_j)$ only depends on $QM(R_1 \rightarrow R_2)$ and $QM(R_2 \rightarrow R_3)$. For example, delay $D(R_1 \rightarrow R_2)$ is such a metric, 

$$D(R_1 \rightarrow R_2 \rightarrow R_3) = D(R_1 \rightarrow R_2) + D(R_2 \rightarrow R_3). \quad (1)$$

To be general, we can define an operator $\oplus$ on $QM$ such that,

$$QM(R_1 \rightarrow R_2) = QM(R_1 \rightarrow R_2) \oplus QM(R_2 \rightarrow R_3). \quad (2)$$

Delay is a metric that obeys the regular addition$(+)$ operation, we call such a metric as an additive QoS metric. Number of hops is another example of additive metrics. Generally delay jitter is also considered to be additive.

We call a metric a transitive QoS metric if it has the operator $\oplus$ defined as:

$$t_1 \oplus t_2 = \min\{t_1, t_2\} \quad (3)$$

or,

$$t_1 \oplus t_2 = \max\{t_1, t_2\} \quad (4)$$

To simplify our discussion, from now on, we consider only transitive metric defined by the $\min$ operator, which is often called concave metric in literature. Available bandwidth of a path is a transitive metric, e.g., $BW(R_1 \rightarrow R_2) = \min\{BW(R_1 \rightarrow R_2), BW(R_2 \rightarrow R_3)\}$.

We call a metric a multiplicative QoS metric if it has the operator $\oplus$ defined as:

$$m_1 \oplus m_2 = m_1 \times m_2 \quad (5)$$

If one defines reliability $r$ as $r = 1 - \text{loss rate}$, then $r$ is a multiplicative metric. On the other hand, if loss rate $L$ on all link is considered small enough such that $L(R_1 \rightarrow R_2) = L(R_1 \rightarrow R_3) + L(R_2 \rightarrow R_3)$, then loss rate $L$ can be considered as an additive metric.

In a multiple-constraint routing problem, there are multiple QoS metrics that have to be bounded; meanwhile, there is one additional cost metric that is intended to be minimized. Normally the cost metric is additive, but theoretically it can be of any type defined above; to simplify our discussion, we assume it is additive in this paper. Consider a path from $S$ to $D$ that has a QoS descriptor denoted as $QD =$ \langle $a_1$, $a_n$, $t_1$, $\ldots$, $t_m$, $m_1$, $\ldots$, $m_n$ \rangle, where $a_i$ is an additive QoS metric for the path, $t_i$ is a transitive metric and $m_k$ is a multiplicative metric, $c$ is a cost metric which can be a metric from $a_i$, $t_i$ or $m_k$; $n_a$, $n_t$, $n_m$ are number of additive, transitive and multiplicative metrics, respectively. To make a meaningful problem, we assume $a_i \geq 0$ for any additive metric $a_i$, and $0 < m_i \leq 1$ for any multiplicative metric $m_i$.

Consider a QoS requirement $QR$, which can be denoted in the same way as a QoS descriptor, $QR = \langle a_1', a_n'$, $t_1'$, $\ldots$, $t_m'$, $m_1'$, $\ldots$, $m_n'$ \rangle. We define the $\leq$ operator between $QR$ and $QD$ (and between two $QD$'s as

$$QR \leq QD = \left\{ \begin{array}{ll}
true & \text{if } a_i' \geq a_i, t_i' \leq t_i, m_i' \leq m_i \\
false & \text{otherwise.}
\end{array} \right. \quad (6)$$

We say a QoS requirement $QR$ is satisfied by the path $S \rightarrow D$ with QoS descriptor $QD$ if $QR \leq QD$. Intuitively if two QoS requirements $QR_1 \leq QR_2$, then $QR_1$ is easier to be satisfied than $QR_2$.

We can also define the $\oplus$ operator between two $QD$'s as

$$QD_1 \oplus QD_2 = \langle \ldots, a_i \oplus a'_i, \ldots, t_j \oplus t'_j, \ldots, m_k \oplus m'_k, \ldots \rangle \quad (7)$$

where QoS descriptor $QD =$ \langle $a_1$, $a_n$, $t_1$, $\ldots$, $t_m$ \rangle and $QD' =$ \langle $a_1'$, $a_n'$, $t_1'$, $\ldots$, $t_m'$ \rangle. For a path $R_1 \rightarrow R_2 \rightarrow R_3$, $QD(R_1 \rightarrow R_2 \rightarrow R_3) = QD(R_2 \rightarrow R_3) \oplus QD(R_1 \rightarrow R_2)$. From now on, we use the index $i$ of a QoS metric in the QoS descriptor to refer to that specific QoS metric.

Consider network nodes $S$ and $D$ and an intermediate node $I$, a path $S \rightarrow I$ with QoS descriptor $QD(S \rightarrow I)$ and a path $I \rightarrow D$ with QoS descriptor $QD(I \rightarrow D)$, we have:

**Lemma 1.** A QoS requirement $QR$ is satisfied by the path $S \rightarrow I \rightarrow D$ if $QR \leq QD(S \rightarrow I)$ and $QR \leq QD(I \rightarrow D).

If $QR \leq QD$, an $\oplus$ operator between them can be defined as

$$QR \oplus QD = \langle \ldots, a_i - a_i', \ldots, t_j, \ldots, m_k/m_k, \ldots \rangle \quad (8)$$

where $QR =$ \langle $a_1$, $a_n$, $t_1$, $\ldots$, $t_m$ \rangle and $QD =$ \langle $a_1'$, $a_n'$, $t_1'$, $\ldots$, $t_m'$ \rangle. The $\oplus$ operator is only defined between a $QR$ and a $QD$, and the result is a new QoS requirement. It is not defined if $IQR \leq QD$.

These operators defined above obey similar rules as regular arithmetic operators, e.g., if $QR \oplus QD_1 \leq QD_2$ then $QR \leq QD_1 \oplus QD_2$, and vice versa. Thus we have:

**Lemma 2.** Given QoS requirement $QR$, a path $S \rightarrow I$ with $QD(S \rightarrow I)$ and a path $I \rightarrow D$ with $QD(I \rightarrow D)$, $QR$ is satisfied by the path $S \rightarrow I \rightarrow D$ if $QR \leq QD(S \rightarrow I)$ and $QR \oplus QD(D \rightarrow I) \leq QD(I \rightarrow D)$.

Now we consider all possible paths from $I$ to destination $D$ through $I$'s neighbor $K$. Let $p_K$ be the path which is the "best" regarding QoS metric $j$: metric $j$ of $QD(p_K)$ is the minimal among all possible paths if metric $j$ is an additive metric, or metric $j$ of $QD(p_K)$ is the maximal among all possible paths if metric $j$ is a transitive or multiplicative metric; $1 \leq j \leq n$, where $n$ is the total number of QoS metrics under consideration plus cost metric. For example, assume bandwidth and delay are QoS constraints under consideration here (with one additional cost metric), then $p_K$ is the path with "most" available bandwidth and $p_K$ is the path with "shortest" delay, and $p_K$ is the path of "minimum" cost. Correspondingly each of these paths has its QoS descriptor $QR(p_K) =$ \langle $a_1$, $a_n$, $t_1$, $\ldots$, $t_m$ \rangle, where $QR =$ \langle $a_1$, $a_n$, $t_1$, $\ldots$, $t_m$ \rangle. Consider metric $i$, if it is an additive metric, then $x_{i,K} = \min\{x_{i,K}, 1 \leq j \leq n\}$; if it is a transit or multiplicative metric, then $x_{i,K} = \max\{x_{i,K}, 1 \leq j \leq n\}$. We define $QD_{x_i}(K, D) =$ \langle $x_{i,1}$, $\ldots$, $x_{i,n}$ \rangle.
min\{QD^j(K, D)\}, 1 \leq j \leq n\} and QDM(K, D) =< ...x_i,... \geq max\{QD^j(K, D)\}, 1 \leq j \leq n\}, where x_i = max\{x_i^j, 1 \leq j \leq n\} and x_i' = min\{x_i^j, 1 \leq j \leq n\} if i is an additive metric, and x_i = min\{x_i^j, 1 \leq j \leq n\} and x_i' = max\{x_i^j, 1 \leq j \leq n\} if i is a transit or multiplicative metric. Intuitively, any path from I to D passing K cannot have a QoS descriptor "better" than QDM(K, D), while any QD^j(K, D) is "better" than QDM(K, D). We have,

Lemma 3. A QoS requirement QR can be satisfied by a path (S \rightarrow I) \rightarrow K \rightarrow D if QR \leq QD(S \rightarrow I) \oplus QD_m(K, D); QR cannot be satisfied by any path (S \rightarrow I) \rightarrow K \rightarrow D if QD(S \rightarrow I) \oplus QD_m(K, D) > QR.

Lemma 4. A QoS requirement QR may or may not be satisfied by a or any path (S \rightarrow I) \rightarrow K \rightarrow D if QD(S \rightarrow I) \oplus QD_m(K, D) = QR and QR \leq QD(S \rightarrow I) \oplus QD_m(K, D). (S \rightarrow I) represents a fixed path from S to I which has QoS descriptor QD(S \rightarrow I).

Lemma 5. If QR \leq QD(S \rightarrow I) \oplus QD_m(K, D), then \exists j s.t. QR \leq QD(S \rightarrow I) \oplus QD^j(K, D).

III. ROUTING ALGORITHMS BASED ON SMART FORWARDING

A. Problem Statement

After introducing necessary notations and definitions in last section, the unicast routing problem with multiple QoS constraints can be formulated as follows:

Problem Statement: Given a directed graph G(V, E) with weight functions which maps edge e \in E to a QoS descriptor QD(e) with n QoS metrics (any metric is of a type of those defined above) including cost function c(e) \geq 0. The cost of a path p is defined as C(p) = \sum_{e \in p} c(e). The optimal routing problem is to find a path p from source node S to destination node D such that: (1) QR \leq QD(p); (2) C(p) \leq C(p') for any path p' with QR \leq QD(p').

B. Assumptions and Pre-computation

With the notations introduced in Section II, it is very straightforward to present our technique. Here first we will show how this scheme can be used in a distributed setting, it is straightforward that it can also be applied as a source-based routing algorithm.

Each node I maintains a table of the following information for every other network node D: for each neighbor K, QD^j(K, D) for the path "best" on metric j (including cost) going to D through K from I. In the rest of this paper, we refer to this table as QD table. To facilitate routing decision, the following additional information can also be computed: QD_m(K, D) = min\{QD^j(K, D)\}, 1 \leq j \leq n\} and QDM(K, D) = max\{QD^j(K, D)\}, 1 \leq j \leq n\} as defined earlier, where n is the number of QoS metrics under consideration. The above information can be pre-computed periodically either by link-state information flooding then utilizing Dijkstra or Bellman-Ford's algorithm, or by a distributed Bellman-Ford algorithm. It is well-known that both algorithms can compute single-metric shortest-path in polynomial time.

When a source S wants to setup a connection to destination D with QoS requirement QR, it sends a routing message with QR specified destined to D. The routing message carries a unique connection id assigned by the source to identify the specific connection request. The routing message will also carry the complete path information with QoS descriptor for that path. If, say, intermediate node I receives a routing message from S destined to D, I knows the path from S to I and QD(S \rightarrow I); delay so far, available bandwidth, etc. on that path. An intermediate node can detect routing loop by checking if it is already in the path.

C. "Smart" Flooding

A node I (including S) makes decision on which neighbor to forward the routing message to as follows. Node I checks QD table for D for all neighbors (indexed by K) except the one from which the routing message is received (assumed to be node K_m), and it makes routing decision based on the following rules:

1) If \{QR \leq QD(S \rightarrow I) \oplus QD_m(K, D)\} for all K \neq K_m, then there is no path from I that will satisfy the QoS requirements (Lemma 3), routing message will not be forwarded further.
2) If QR \leq QD(S \rightarrow I) \oplus QD_m(K, D) or QR \leq QD(S \rightarrow I) \oplus QD^j(K, D) for some neighbor K and metric index j, then forward the routing message to that neighbor. By Lemma 1 and 3, there is a path going through K which will satisfy QoS requirement QR. If more than one neighbor satisfies that condition, choose the one to minimize the total cost C(S \rightarrow I) \oplus c^j(K), where j is the index for which QR \leq QD(S \rightarrow I) \oplus QD^j(K, D).
3) If condition in (2) cannot be satisfied with any neighbor, forward routing message to all neighbors which can satisfy QR \leq QD(S \rightarrow I) \oplus QD_m(K, D). (Lemma 4)

When destination D receives a routing message which has QR \leq QD(S \rightarrow D) satisfied, it can send back a reply along the path recorded to source S. That reply message can work with other protocols such as RSVP to setup the connection and reserve all required resources. If condition (2) is satisfied all the way from S to D, then D will only receive one routing message. If along the way condition (3) is satisfied at some node and routing message is forwarded to more than one neighbor, then D may receive multiple routing messages for the same connection request (identified by connection id). It should choose one to reply to and neglect all others. For this purpose, a destination node can wait for a short period after it receives a successful routing message so it can collect all successful routing messages and choose the "best" one. No routing message may reach D if all failed routing messages are dropped. To resolve this, an intermediate node can send a failure notification back to the source.

From the above description, one can see that it is possible for a node to forward multiple routing messages that arrive by traveling on different paths for the same connection request. Sometimes it is necessary since different routing messages that have traveled different paths may be all potentially feasible while none is known to be definitely possible at the moment. However, sometimes multiple routing messages from a same node may be unnecessary. Consider the situation that, one routing message is received at node I which traveled path p_1 with QD_1, and then another message that traveled path p_2 with QD_2, and both are for a connection request with QR. If QR \oplus QD_1 \leq QR \oplus QD_2, then there is no need for I to for-
ward the second routing message: $QR \oplus QD_1$ is easier to be satisfied than $QR \oplus QD_2$. To prevent this type of unnecessary routing messages being forwarded, network nodes must maintain state information for routing messages forwarded. This can eliminate some unnecessary network traffic but introduces additional complexity.

C.1 Bounded “Smart” Flooding

The above scheme guarantees it can find a feasible solution (though may not be optimal) if there is one. It can greatly reduce the number of routing messages flooded in the network by rule (1) and (2). However, if it is desired to further reduce the number of routing messages, the following technique can be applied as well.

The original routing message from the source can carry an integer number (called a token here) as the number of routing messages that can be re-generated in the network for a given connection request. Inside the network, when situation (1) happens, simply a routing message is dropped. When situation (2) happens, no additional routing message is regenerated. When situation (3) happens, the forwarding node can split the token into smaller numbers (while keep the summation as original) as tokens for regenerated message. If the token value is already one, then the routing message can only be forwarded to one neighbor. The token value can be distributed among new messages evenly or weighted based on some pre-defined criteria (say, cost of potentially feasible path). This way, the total number of routing messages forwarded can only be as many as specified by the original token value.

Bounded flooding scheme can reduce control overhead and network traffic. However, it does so by limiting choices of potential feasible paths and thus may not always be able to find feasible solutions even if there are. This is also true with bounded crank-back to be discussed.

D. Crank-Back

Above we have shown how our technique can be used as a “smart” flooding routing scheme. Apparently it can be used in a crank-back style of routing scheme too.

As in Section III-C, each node computes and maintains the $QD$ table. The source and an intermediate node check and make forwarding decision on routing message as described there. Changes to the rules are the following. When condition (2) happens, a routing message is forwarded to the corresponding neighbor the same way. However, when situation (3) happens, the node (say $I$) will only forward the routing message to one neighbor. If there are more than one neighbor that satisfies the condition, $I$ maintains a list of these neighbors for the connection request (marks those that have been tried); at the same time, it marks itself as a “fork” point in the path in the forward routing message. When situation (1) happens, if there is no “fork” point in the path then the routing message is dropped (a failure report is to be sent back to the source); if there is one or more than one “fork” point in the path, then the message is sent back to the most recent (nearest) “fork” point. When a node receives a “turned back” routing message, it will try the next candidate in the list it maintains; if there is one candidate left, then it removes the “fork” mark since it runs out of option and there is no need to send back the routing message to it.

This crank-back process can be stopped anytime when a routing message hits the destination and is a successful one. Otherwise it terminates after all potentially feasible routes are tried. Once again, the use of our smart forwarding technique can help reduce the number of trials that will be performed. However, if it is desired to limit the number of trials, a similar technique as in Section III-C.1 can be applied: a token as total number of trials allowed is stored in the routing message and its value is decreased by one every time a crank-back happens. Crank-back process is terminated when the token reaches zero.

It is interesting to observe that flooding and crank-back are actually very similar. The difference is that flooding algorithm always tries to “fork” routing message at nodes close-to-source-first, while the crank-back algorithm “forks” routing message at nodes close-to-destination-first.

E. Source-Based Routing

Smart forwarding technique can also be applied in a centralized source-based routing scheme. As described in Section III-B, every node will need to compute and maintain a $QD$ table for every other node in the network, not just itself. With this table, a node can locally run a routing algorithm similar to the distributed flooding and crank-back algorithms described above. This can be implemented in many different ways, here we are not going into algorithm details. One way is to use a queue or stack to store intermediate nodes (along with neighbor information) that are “fork” points (which means situation (3) in Section III-C) when constructing an end-to-end path. When a queue is used, “fork” points will be tried in the order that they appear in the process of constructing path from the source to the destination, it is a centralized implementation of “flooding”. When a stack is used, most recent “fork” point (thus closer to the destination) is tried first, thus it is like “crank-back”. When there is no limit on queue or stack size, it is unlimited flooding or crank-back. If we limit the size of the queue(stack) and the number of push/pop operations, then it is bounded flooding(crank-back).

F. Delay-Constrained Routing

When QoS constraints are on bandwidth and delay only (or any one transitive metric plus one additive or multiplicative metric), it is sufficient to solve a single constraint problem in some cases: (1) bandwidth requirement is small and can be easily met while end-to-end delay bound is tight; (2) in a source-based routing setting, the source computes the route on-demand and it can exclude links without enough bandwidth before it computes the $QD$ table. Under this situation, smart-forwarding based algorithms demonstrates some interesting properties.

**Property 1.** If there is only one additive or multiplicative QoS constraint (assumed to be delay constraint here) in addition to cost minimization objective, smart-forwarding based algorithms (flooding and crank-back) can always find a feasible path if there is one, by sending only one routing message over the network in distributed hop-by-hop setting assuming consistent and stable network state. A one-pass trial can also do so in a centralized source-based routing.

**Proof.** Examine the rules specified in Section III-C, one can see that at each hop, the minimum-cost path from that node to the destination is tried first; if it can’t meet the delay constraint, the minimum-delay path will be tried. If a connection
request is not rejected at the source, then the routing message will be either forwarded along the minimum-cost path or the minimum-delay path initially. A routing message that is forwarded along the minimum-cost path initially will stay on that path until it reaches the destination. This is because at each intermediate node the minimum-cost path is tried first and it should always meet delay constraint since consistent and stable network state is assumed. On the other hand, a routing message that is forwarded along the minimum-delay path initially will stay on the minimum-delay path until it reaches an intermediate node (or the destination otherwise) at which point the minimum-cost path becomes feasible and from that point the routing message will be forwarded along the minimum-cost path. The same argument applies when a path is constructed hop-by-hop in a centralized setting.

**Property 2.** Under the same conditions as in Property 1, if the minimum-cost path is feasible, smart-forwarding based algorithms will find the minimum-cost path; on the other hand if the only feasible solution is the minimum-delay path, then smart-forwarding based algorithms will find the minimum-delay path.

**G. Computation and Processing Complexity**

1) QD table update: Let \( N = |V| \) and \( M = |E| \). First we consider distributed routing. If QD table is computed using link-state information flooding, a node can compute the "best" path of a single metric to all destinations with a specific neighbor as the next hop using Dijkstra's algorithm which has a time complexity \( O(M \log(N)) \). If node \( j \) has \( n_j \) neighbors, it has to run the algorithm for all of them, then the total complexity is \( O(n_j \times n \times M \times \log(N)) \) for a total of \( n \) metrics. All network nodes have to do this, a complete QD table update for all nodes will run Dijkstra's algorithm \( 2 \times |E| \times n \) times (\( \sum n_j = 2|E| \)) which gives a total time complexity \( O(M^2 \log(N)) \). If QD table is computed using distributed Bellman-Ford algorithm, one pass of QD table update costs \( O(n_j \times N) \) for a single node, or \( O(M \times N) \) for all nodes. In a source-based routing setting, link-state information flooding has to be used, and every node has to compute and maintain QD table for itself and all other nodes. One pass table update for a single node will cost \( O(M \log(N)) \).

2) Routing message forwarding: At each hop, a total of \( O(n_j) \) comparisons are necessary to make a routing message forwarding decision. The worst-case bound for a routing message forwarded end-to-end is \( O(M) \). Theoretically there is no polynomial bound on the total number of comparisons if there is no limit on number of flooding or crank-back trials since it is an NP problem. If we limit the number of trials to be \( T \) using techniques described earlier, then total number of comparisons is bounded by \( O(T \times M) \).

**IV. SIMULATION RESULTS**

Simulation has been used to study the behavior of our algorithm and to compare its performance with other schemes. Because of space limitation, only the results for delay-constrained routing (Section III-F) are presented here. Three other routing strategies are compared here. One is min-delay path algorithm which guarantees to find a path if there is a feasible one. One is min-cost path algorithm: it only tries the min-cost path; however it will be the optimal solution if it satisfies the delay bound. The third one tries min-cost path first then tries min-delay path if min-cost path doesn't work out. It is similar to the strategy used in some vendors' ATM PNNI implementation [8].

We use randomly generated networks using the approach given in [13]. In simulation presented here, networks have a fixed size of 200 nodes chosen over a 40 x 40 grid. Parameters are chosen such that in average each node has degree about 5. Geometric distance is used as delay on a link. To be as general as possible, a random cost between 0 to 1 is generated for each link. For simplicity, links are assumed to be bi-directional and symmetric. Here we only consider delay bound, so we are assuming all links have enough bandwidth or links without enough bandwidth are eliminated before running the algorithm. Connection requests are generated between randomly chosen (source, destination) pairs with a given delay bound. Every point in figures to be shown is the average of 1000 instances.

In our simulation we vary delay bound and compare path cost and connection request success rates of different approaches. Fig. 2 shows the path cost vs. delay bound of the three algorithms which are always able to find a feasible path (though may not be optimal) if one exists: min-delay, min-cost then min-delay and smart-forwarding. From Fig. 2 one can see that smart-forwarding performs better than the other two. Since min-delay path algorithm doesn't consider path cost at all, it always has the highest cost. One may expect its cost stays flat with delay bound, however here one can see its average cost is lower with tighter delay bound. Average path cost is not flat because it is the average of "feasible" paths, which are a subset of all min-delay paths. When delay-bound is tight, many min-delay paths are not feasible and feasible ones tend to have smaller number of hops, thus smaller delay and cost. When delay bound is loosened, then almost all min-delay are feasible, thus the average path cost is getting close to be con-
stant as shown. Smart-forwarding with delay-constraint is essentially "min-cost then min-delay" at each hop. It is not surprising that it performs better than "min-cost then min-delay" which only tries that strategy at the first hop.

Fig. 3 shows the number of successful instances out of total 1000 runs for each delay bound. Smart-forwarding, min-delay and min-cost then min-delay all are capable of finding feasible path if there is one, so they all have the same number of successful instances. While min-cost-only has much lower success rate as shown.

To see how close to optimal solutions smart forwarding approach can get, another greedy algorithm based on Bellman-Ford algorithm is simulated. Theoretically greedy algorithm doesn’t guarantee to find a feasible solution even if there is one, but it guarantees the solution it finds is optimal. Pseudo code is attached as an appendix. Fig. 4 shows the results for network size of 100 nodes, each point as an average of 1000 instances. One can see that smart-forwarding gets very close to optimal solutions, the difference is between 3% and 7%. The difference between "min-cost then min-delay" and optimal is between 16% and 27%. This demonstrates that while smart-forwarding can always find a feasible solution if there is one, it also finds low-cost solutions with reasonable computation complexity. In this simulation, greedy algorithm actually also find feasible solutions in all cases when there is feasible one. However, greedy algorithm has higher computation cost than smart-forwarding and can only be used as a centralized algorithm.

V. CONCLUSION

In this paper we first introduced an algebraic notation for QoS metrics and then presented a "smart forwarding" technique for QoS routing with multiple constraints based on our algebraic notation. It is discussed in both flooding-based and crank-back based distributed routing environments, and also in centralized source-based routing. This technique can reduce routing overhead by forwarding routing request only to a neighbor that is known to be able or potentially be able to meet a connection’s QoS requirement. Such type of "smart" decision is made based a pre-computed table which can be constructed from periodic link-state information exchange or distributed Bellman-Ford algorithm. A routing request can be processed with little overhead after this table is constructed. When there is only delay constraint, smart-forwarding base algorithms only need one routing request being forwarded end-to-end to find the path, and it can always find a feasible path if there is one. Brief simulation results show that indeed it is effective in finding low-cost paths.

REFERENCES


APPENDIX. A GREEDY DELAY-BOUNDED MIN-COST PATH HEURISTIC ALGORITHM

Input: \( G = (V, E) \), cost \( c(e) \) and delay \( d(e) \) for \( e \in E \), source \( s \), destination \( t \) and delay bound \( D \).
Output: a path from \( s \) to \( t \) satisfying delay bound.

begin

for all vertices \( w \) do

\( w.cost = \infty \);
\( w.delay = \infty \);
\( s.cost = 0 \), \( s.delay = 0 \);

for \( i = 1 \) to \( |V| \) do

for edge \( e(u, v) \) in \( E \) do

update \( v.cost \), \( v.delay \), and \( v.pred \) if

\( c(e) + u.cost < v.cost AND \)
\( d(e) + u.delay < v.delay \), or

\( c(e) + u.cost < v.cost AND \)
\( d(e) + u.delay \leq D \) AND

\( v.delay > D AND \)
\( d(e) + u.delay < v.delay \);

output path for \( t \) if \( t.delay \leq D \);

end

Note: \( v.cost \) and \( v.delay \) represent the cost and delay of the current path from \( s \) to \( v \), respectively. \( v.pred \) represents the preceding node of node \( v \).

Lemma A.1. The above greedy algorithm guarantees the solution it finds is optimal if it finds a feasible solution; however it doesn’t guarantee it always be able to find a feasible solution even there is one.