

A Scalable Approach to the Partition of QoS Requirements in Unicast and Multicast

Ariel Orda and Alexander Sprintson

Abstract—

Supporting Quality of Service (QoS) in large-scale broadband networks poses major challenges, due to the intrinsic complexity of the corresponding resource allocation problems. An important problem in this context is how to partition QoS requirements along a selected topology (path for unicast, tree for multicast). As networks grow in size, the scalability of the solution becomes increasingly important. This requires to devise efficient algorithms, whose computational complexity is less dependent on the network size. In addition, recently proposed *precomputation*-based methods can be employed to facilitate scalability by significantly reducing the time needed for handling incoming requests.

We present a novel solution technique to the QoS partition problem(s), based on a “divide and conquer” scheme. As opposed to previous solutions, our technique considerably reduces the computational complexity in terms of dependence on network size; moreover, it enables the development of precomputation schemes. Hence, our technique provides a scalable approach to the QoS partition problem, for both unicast and multicast. In addition, our algorithms readily generalize to support QoS routing in typical settings of large-scale networks.

Keywords— **QoS partition, Performance-dependent costs, Multicast, Routing, Resource allocation.**

I. INTRODUCTION

Future communication networks are expected to support applications with quality of service (QoS) requirements. Supporting QoS poses major challenges due to the large size and complex structure of networks. A key issue in the design of broadband architectures is how to allocate network resources in order to meet end-to-end QoS requirements in a way that maximizes the overall network performance.

Several network mechanisms need to be introduced to support QoS. One is a QoS routing mechanism, whose purpose is to find a suitable topology (path for unicast, tree for multicast) that can support the connection(s) QoS requirements. Then, a second mechanism is required, in order to optimally allocate re-

sources (e.g., bandwidth, buffer space) along the selected topology such that the required QoS can be guaranteed at minimal cost.

A network link (or element) can offer several levels of QoS guarantees, each associated with a certain cost. The link’s cost represents the consumption of local resources that must be reserved on the link in order to support the QoS guarantee. For example, in the DiffServ architecture [1] a service provider can offer several types of service at different prices. Moreover, links may aggregate subnetworks (e.g., accordingly to the ATM PNNI recommendations [10]), in which case each link represents several paths that support different QoS requirements at different cost values. Accordingly, we consider a network model, in which each link is associated with a *performance-dependent cost function*.

The problem of optimal partition of QoS requirements was formulated by Lorenz and Orda [6] and has been the subject of several studies [3], [11], [7], [2], [5]. Efficient optimal solutions for the special case of *convex* cost functions for both unicast and multicast were established in [6]. However, the convexity assumption is not valid in many cases of practical interest. Since in the general case the problem of optimal partition is intractable (i.e., NP-hard [6]), suitable approximation schemes were presented in [7], [2], [11]. While the computational complexity of those proposed approximations is polynomial, it depends heavily on the size of the topology, which renders these solutions *unscalable*. The high complexity, in turn, results in a high response time to each connection request, which adversely affects the service to network users.

Accordingly, the purpose of this study is to provide scalable solution schemes to the problem. This is achieved in two ways. First, we establish algorithmic solutions that are considerably less dependent on the size of the routing topology than previous proposals. Second (and independently), we employ a *precomputation* approach, in order to further enhance scalability. We proceed to discuss each of these two contributions.

The major contribution of this study is a novel solution technique that allows to better exploit the specific structure of routing topologies (paths and trees). More specifically, we employ a divide-and-conquer scheme, which first computes the costs of supporting various QoS requirements through smaller components (subpaths and subtrees), and then combines the results in order to obtain solutions for larger components. Our technique can be easily distributed. Furthermore, it can be generalized to handle the combined problem of routing and partition of QoS in typical settings of large-scale networks.

Precomputation-based methods have recently been proposed [4], [8] (in the context of QoS routing) as an instrument to facilitate scalability, improve response time and reduce the computational load on network elements. The key idea is to effectively reduce the time needed to handle a request, by performing a certain amount of computations in *advance*, i.e., prior to the request's arrival. Such advance computations are performed as background processes, i.e., when a network element is idle or underutilized, thus resulting in better utilization of the computational capabilities of network elements. In addition, when the rate of incoming requests is high, a considerable reduction in overall computational load is achieved. Accordingly, we employ the precomputation approach in order to improve the scalability of our solutions.

The rest of this paper is organized as follows. In Section II, we formulate the network model and formally state the considered problems. In Section III, we describe our solution methodology. Section IV deals with unicast topologies and presents solutions both for performing on-demand computations as well as precomputations. Section V presents similar solutions for the much more complex setting of multicast. In Section VI we present some conclusions, and, in particular we discuss the applicability of our methods to QoS routing.

Due to space limits some proofs are omitted and appear in [9].

II. MODEL AND PROBLEM FORMULATION

This section formulates the general model and main problems addressed in this paper. A *network* is represented by a directed graph $G(V, E)$, where V is the set of nodes and E is the set of links. Let $N = |V|$ and $M = |E|$. A *path* is a finite sequence of nodes $\mathbf{p} = (v_0, v_1, \dots, v_n)$, such that, for $0 \leq i \leq n - 1$, $(v_i, v_{i+1}) \in E$; $n = |\mathbf{p}|$ is then said to be the *number of hops (or hop count)* of \mathbf{p} . A *tree* is a connected subgraph $\mathbf{T}(V, E)$ of $G(V, E)$ whose

undirected version $\mathbf{T}'(V, E')$ is acyclic.

We assume that the connection's topology is given, i.e., a path, \mathbf{p} , for unicast, or a tree, \mathbf{T} , for multicast. For clarity of presentation, we focus here on unicast; the definitions and terminology for multicast are presented in Section V.

Each link $l \in E$ offers different (integer) QoS guarantees $\{d_l\}$, whose significance depends on the type of considered QoS requirement. For example, when the QoS requirement is an upper bound on the end-to-end delay, the values $\{d_l\}$ are delay guarantees supported by link l . A QoS partition P_D of an end-to-end QoS requirement D , on a path, \mathbf{p} , is a set $\{d_l\}_{l \in \mathbf{p}}$ of local QoS requirements, which satisfies the end-to-end QoS requirement D .

QoS requirements may be *additive*, such as delay and jitter, or *bottleneck*, such as bandwidth. As is easy to verify, the QoS partition problem is straightforward for bottleneck metrics, hence we focus on additive QoS requirements. In other words, a partition of a QoS requirement D is a set $P_D = \{d_l\}_{l \in \mathbf{p}}$ on a path \mathbf{p} such that $\sum_{l \in \mathbf{p}} d_l \leq D$. For clarity of presentation and without loss of generality, we describe our model and problems in terms of end-to-end delay requirements.

For each link $l \in E$, there is a *link cost function*, $c_l(d_l)$, which assigns a cost to each delay guarantee d_l that the link offers. We assume the $c_l(d_l)$ is higher for tighter delay constraints, i.e., the function $c_l(d_l)$ is monotonically decreasing. The link cost function estimates the quality of the link in terms of resource utilization; it may depend on various factors, e.g., the link's available bandwidth, its location, etc. The link cost function can be specified by either an algebraic expression (e.g., $c_l(d_l) = \lfloor 1/d_l \rfloor$) or by a table that specifies costs for supporting various delay guaranties. In the latter case, we say it is a *discrete cost function*. We shall assume that all parameters (both delay guaranties and costs) are (positive) integers. The overall cost of a partition $c(P_D)$ is the sum of the local costs, i.e., $c(P_D) = \sum_{l \in \mathbf{p}} c_l(d_l)$.

The optimal QoS partition problem is then defined as follows.

Problem OPQ (Optimal Partition of QoS) Given a path $\mathbf{p} = \{s, \dots, t\}$ and a QoS requirement D , find a QoS partition $\{d_l\}_{l \in \mathbf{p}}$ such that $\sum_{l \in \mathbf{p}} d_l \leq D$ and $\sum_{l \in \mathbf{p}} c_l(d_l)$ is minimized.

The solution $\hat{P}_D = \{\hat{d}_l\}_{l \in \mathbf{p}}$ of Problem OPQ is referred to as an *optimal partition of a QoS requirement D along \mathbf{p}* .

In general, Problem OPQ is intractable, i.e., NP-

hard [6]. Accordingly, in this work we resort to scalable ε -approximate solutions, *e.g.*, solutions of (low) polynomial complexity, whose cost is at most $(1 + \varepsilon)$ times larger than the cost of the optimal solution.

III. LAYERING APPROACH

For clarity of exposition, we focus in this section on unicast connections, and defer the discussion on multicast to Section V. Our approach is based on the following divide-and-conquer scheme. We recursively split the given unicast path into some disjoint set of subpaths. Considering each such subpath and each delay value d , $1 \leq d \leq D$, we attempt to compute the optimal cost of partitioning d along that subpath. We then obtain a solution to the original problem, *i.e.*, on path \mathbf{p} and for the end to end constraint D , by recursively combining the solutions obtained for the subpaths.

More specifically, consider a unicast path $\mathbf{p} = \{s = v_0, v_1, \dots, v_n = t\}$, which is referred to as a *layer-0* path. We split \mathbf{p} into two *layer-1* subpaths $\{v_0, \dots, v_b\}$ and $\{v_b, \dots, v_n\}$, where $b = \lfloor n/2 \rfloor$. Then, for each value k , $k = 1, 2, \dots, K$, each layer- k subpath $\{v_i, \dots, v_j\}$ is split into two layer- $(k+1)$ subpaths $\{v_i, \dots, v_b\}$ and $\{v_b, \dots, v_j\}$, where $b = \lfloor (i+j)/2 \rfloor$. The value of K is set such that layer- K subpaths comprise of just a single link, *i.e.*, $K = \lceil \log n \rceil$. Clearly, the number of subpaths n_k of a layer k is $O(2^k)$.

As mentioned, our purpose is to calculate, for each subpath of layer- k , $1 \leq k \leq K$, and each delay value d , $1 \leq d \leq D$, the costs of supporting the delay d along the subpath. To that end, we define the following subpath cost functions.

Definition 1: The *optimal subpath cost function* $C_{(v,u)}^{opt}(d)$ of a subpath $\mathbf{p}' = \{v, \dots, u\}$ of \mathbf{p} is defined as the cost of the optimal partition of the QoS requirement d along the subpath \mathbf{p}' , *i.e.*,

$$C_{(v,u)}^{opt}(d) = \min_{\{d_l\}_{l \in \mathbf{p}'}} \left\{ \sum_{l \in \mathbf{p}'} c_l(d_l) \mid \sum_{l \in \mathbf{p}'} d_l \leq d \right\}$$

Note that, if the subpath between v and u comprises of a single link (v, u) , then $C_{(v,u)}^{opt}(d)$ is identical to the cost function of that link, *i.e.*, $C_{(v,u)}^{opt}(d) = c_{(v,u)}(d)$.

While optimal subpath cost functions accurately capture the costs of supporting QoS requirements, they are impractical, since their computation is intractable, and moreover, their storage requirements are prohibitively large. Accordingly, we resort to ε -approximate cost functions, whose computation and storage requirements are feasible.

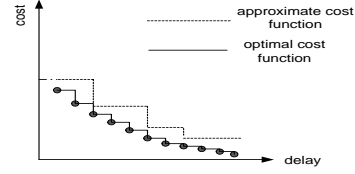


Fig. 1. A cost function and its approximation

Definition 2: An ε -approximate (subpath) cost function $C_{(v,u)}^\varepsilon(d)$ of a subpath $\mathbf{p}' = \{v, \dots, u\}$ of \mathbf{p} is an (integer valued) function that satisfies, for each $d \geq 0$, $C_{(v,u)}^\varepsilon(d) \leq (1 + \varepsilon) \cdot C_{(v,u)}^{opt}(d)$.

For the sake of clarity, and when no ambiguity exists, ε -approximate cost functions shall be referred to as just *cost functions*.

We shall construct cost functions by employing a *logarithmic scaling* method. The idea is to sample the optimal cost function at cost values $\{1, \delta, \delta^2, \dots, \delta^{\lceil \log_\delta U \rceil}\}$, where δ is a scaling factor and U is an upper bound on the cost of the optimal partition. For each sampled cost value c , the sampling yields an interval of delay values, such that for each of which the corresponding cost is at least c , but lower than the next sampled value (see Fig. 1).

The storage requirements of a cost function depend on the number of its segments, *i.e.*, the number of sampled values. Indeed, for each segment, it is sufficient to keep only the minimum delay of its leftmost point. For example, the approximate cost function depicted on Fig. 1 can be represented by the following delay values: $\{d_1, d_2, d_3, d_4\}$. In the algorithms presented in the next sections, the function $C_{(v_i, v_j)}^\varepsilon(d)$ is stored in a table $D[v_i, v_j, c]$ that keeps the delay of the leftmost point of the segment of $C_{(v_i, v_j)}^\varepsilon(d)$ that corresponds to cost c .

The computation of a cost function introduces some inaccuracy ε_k at each layer k , which accumulates as we proceed to upper layers. In particular, we obtain: for layer K , $\varepsilon^{(K)}$ -approximate cost functions, where $\varepsilon^{(K)} = \varepsilon_1$; for layer $K - 1$, $\varepsilon^{(K-1)}$ -approximate cost functions, where $\varepsilon^{(K-1)} = (1 + \varepsilon_K)(1 + \varepsilon_{K-1}) - 1$; and, in general, for layer k , $\varepsilon^{(k)}$ -approximate cost functions, where $\varepsilon^{(k)} = \prod_{l=K, \dots, k} (1 + \varepsilon_l) - 1$. For each layer k we use the scal-

ing factor $\delta_k = 1 + \varepsilon_k$. The values $\{\varepsilon_k\}$ should be carefully chosen in order to obtain a solution of low complexity. Accordingly, we assign $\{\varepsilon_k\}$ such that the amount of computations required for processing

each layer is the same at all layers. Specifically, we use the following assignment of ε_k :

$$\varepsilon_k = \begin{cases} \varepsilon \cdot \frac{\sqrt{2}-1}{8\sqrt{2}} & \text{if } k = K, \\ \frac{\varepsilon_{k+1}}{\sqrt{2}} & \text{otherwise.} \end{cases} \quad (1)$$

The efficiency of this choice shall be established in the following.

IV. UNICAST: PROBLEM OPQ

In this section we deal with the partition of QoS requirements along unicast paths. As explained in the Introduction, we establish scalable solutions to the problem by employing two independent approaches. The first, "divide and conquer" approach, significantly reduces the solution's dependence on the topology size. The second approach employs a pre-computation scheme. We note that the divide and conquer approach is applicable to both the (standard) on-demand problem, as well as to the precomputation scheme. Although the on-demand setting is more basic, it is simpler to present the results by considering first the precomputation setting.

A. Precomputation scheme

Precomputation is performed by means of a two-phase procedure, which we refer to as a *precomputation scheme*. The first phase is executed in advance and its purpose is to precompute solutions *a priori*, for a wide set of parameters. The computations performed at this stage are then summarized into a data structure for later usage. The purpose of the second phase is to provide an adequate solution on demand, *i.e.*, upon an incoming request. The second phase either selects one of the solutions precomputed at the first phase, or, if necessary, performs additional computations.

We apply a precomputation scheme to Problem OPQ, *i.e.*, given a source-destination pair (s, t) and an interconnecting path $\mathbf{p} = \{s = v_0, \dots, v_n = t\}$, we precompute optimal QoS partitions for a *wide range* of QoS requirements.

For clarity of exposition, we assume that $n = 2^K$; dropping this assumption requires a mild and straightforward modification of our results, with no penalty in terms of computational complexity.

A.1 Scheme Description

The goal of the precomputation scheme is to identify the cost function $C_{(s,d)}^\varepsilon$ for \mathbf{p} and the corresponding partitions. First, we compute the cost functions

for layer- K subpaths, then for layer- $(K-1)$ subpaths, etc., up to layer 0. While processing subpaths of a layer k , we use the previously computed cost functions for the subpaths of layer $(k+1)$.

Since a subpath of layer- K comprises of a single link (v_i, v_{i+1}) , its cost function can be obtained by applying logarithmic scaling on the cost function $c_l(d_l)$, *i.e.*, $D[v_i, v_{i+1}, \delta_1^t] = \min\{d | c_{(v_i, v_{i+1})}(d) \leq \delta_1^t\}$, for each $t = 0, 1, \dots, \lceil \log_{\delta_1} U \rceil$, where U is an upper bound on the cost of an optimal partition (a trivial choice being the cost of the supporting the most stringent delay requirement, *i.e.*, $U = \sum_{l \in \mathbf{p}} c_l(1)$).

A key building block in our scheme is *merging* $C_{(v_i, v_b)}^\varepsilon(d)$ and $C_{(v_b, v_j)}^\varepsilon(d)$ for subpaths $\{v_i, \dots, v_b\}$ and $\{v_b, \dots, v_j\}$ into a cost function for subpath $\{v_i, \dots, v_j\}$. The merge operation essentially amounts to finding, for each budget $c_t \in \{1, \delta_k, \delta_k^2, \dots\}$, the partition (c_t^1, c_t^2) of a budget c_t between the subpaths, which minimizes the weight of the subpath $\{v_i, \dots, v_j\}$.

A straightforward solution would be to examine all possible partitions (c_t^1, c_t^2) of the budget c_t . Since the choice of c_t^1 determines c_t^2 , it is sufficient to consider each $c_t^1 \leq c_t$. Moreover, since the cost function $C_{(v_i, v_b)}^\varepsilon(d)$ comprises of several segments, only values of c_t^1 that correspond to segments of $C_{(v_i, v_b)}^\varepsilon(d)$ should be considered.

The merging can be performed more efficiently by the following procedure. We divide the set $\{(c_t^1, c_t^2) | c_t^1 + c_t^2 \leq c_t\}$ of feasible partitions into two subsets, the first includes the partitions for which $c_t^1 \leq c_t/2$ and the second the partitions for which $c_t^1 > c_t/2$. Then we identify, for each subset, the partition that minimizes the weight of the subpath $\{v_i, \dots, v_j\}$. For the first subset, we note that it is sufficient to examine partitions for which values of c_t^2 correspond to segments of $C_{(v_b, v_j)}^\varepsilon(d)$. Thus, and since $c_t^2 > c_t/2$, we need to consider only $O(1/\varepsilon_{k+1})$ values of $c_t^2 \in \{\delta_{k+1}^t | c_t/2 < \delta_{k+1}^t \leq c_t\}$. Similarly, for the second subset, it is sufficient to consider only $O(1/\varepsilon_{k+1})$ values of $c_t^1 \in \{\delta_{k+1}^t | c_t/2 < \delta_{k+1}^t \leq c_t\}$. We conclude that the optimal partition of the budget c_t requires $O(1/\varepsilon_{k+1})$ time. Since we need to find an optimal partition for $O(\log U/\varepsilon_k)$ budget values, the total complexity incurred is $O(\log U/\varepsilon_k^2)$, where U is an upper bound on the cost of an optimal partition. This procedure is referred to as Procedure MERGE and its formal specification is presented in Fig. 2.

The first phase of the precomputation scheme is implemented by Algorithm POPQ (Figure 3). The al-

```

Procedure MERGE ( $v_i, v_b, v_j, \delta_k, \delta_{k+1}, U$ )
1  for  $t = 0$  to  $\lceil \log_{\delta_k} U \rceil$  do
2     $c_t \leftarrow \delta_k^t$ 
3    if  $t = 0$  then  $D[v_i, v_j, c_t] \leftarrow \infty$ 
4    else  $D[v_i, v_j, c_t] \leftarrow D[v_i, v_j, c_{t-1}]$ 
5     $c_t^2 \leftarrow \text{ROUND}(\delta_{k+1}, c_{t-1}/2)$ 
6    while  $c_t^2 \leq c_t$  do
7       $c_t^1 \leftarrow \text{ROUND}(\delta_{k+1}, c_t - c_t^2)$ 
8       $D[v_i, v_j, c_t] \leftarrow \min\{D[v_i, v_j, c_t], D[v_i, v_b, c_t^1] +$ 
9         $D[v_b, v_j, c_t^2]\}$ 
10      $c_t^2 \leftarrow c_t^2 \cdot \delta_{k+1}$ 
11      $c_t^1 \leftarrow \text{ROUND}(\delta_{k+1}, c_{t-1}/2)$ 
12     while  $c_t^1 \leq c_t$  do
13        $c_t^2 \leftarrow \text{ROUND}(\delta_{k+1}, c_t - c_t^1)$ 
14        $D[v_i, v_j, c_t] \leftarrow \min\{D[v_i, v_j, c_t], D[v_i, v_b, c_t^1] +$ 
15          $D[v_b, v_j, c_t^2]\}$ 
16        $c_t^1 \leftarrow c_t^1 \cdot \delta_{k+1}$ 
Procedure ROUND ( $\delta, x$ )
1  return  $\delta^{\lceil \log_{\delta} x \rceil}$ 

```

Fig. 2. Procedure MERGE

gorithm begins by computing cost functions for subpaths of layer- K . Then, the cost function for subpaths of layer- $(K-1)$ are computed by merging cost functions for subpaths of layer- K (Procedure MERGE). In a similar way, we compute the cost function of layer- $(K-2)$, etc., up to the last, 0th layer. The cost function for a subpath $\{v_i, \dots, v_j\}$ is stored in the table $D[v_i, v_j, c]$.

```

Algorithm POPQ ( $G(v, E), \mathbf{p}, \varepsilon, U$ )
parameters
   $G(V, E)$ -the network;
   $\mathbf{p} = \{v_0 = s, \dots, v_n = t\}$ -a QoS path;
   $\varepsilon$ -approximation ratio;
   $U$ -the upper bound on the cost of an optimal partition.
1   $\varepsilon_K \leftarrow \varepsilon \cdot \frac{\sqrt{2}-1}{\sqrt{2}}$ ,  $\delta_K = 1 + \varepsilon_K$ 
2  for each  $l = (v_i, v_{i+1}) \in \mathbf{p}$  do
3    for each  $c_l \in \{\delta_K^l | 0 \leq \delta_K^l \leq U\}$  do
4       $D[v_i, v_{i+1}, c_l] = \min\{d | c_l(d) \leq c_l\}$ 
5  for  $k \leftarrow K-1$  to 0 do
6     $\varepsilon_k \leftarrow \frac{\varepsilon_{k+1}}{\sqrt{2}}$ ,  $\delta_k \leftarrow 1 + \varepsilon_k$ 
7    for each subpath  $\{v_i, \dots, v_j\}$  of layer  $k$  do
8       $b = \frac{i+j}{2}$ 
9      invoke Procedure MERGE for  $(v_i, v_b, v_j, \delta_k, \delta_{k+1}, U)$ .

```

Fig. 3. Algorithm POPQ

Upon a request with some QoS requirement D , the optimal partition is promptly identified by examining the output of Algorithm POPQ. Specifically, we identify, through binary search, the cost $c(D)$ of a suitable partition, $c(D) = \min\{c_t = \delta_0^t | D[s, d, c_t] \leq D\}$, and return the corresponding partition. The computational complexity of this procedure is $O(\log \log U + \log(1/\varepsilon) + n)$. The term n in the complexity expression is due to the need to de-

scribe the partition.

A.2 Analysis of the precomputation scheme

We proceed to analyze our precomputation scheme. First, we assess the inaccuracy introduced by a single invocation of Procedure MERGE, which is needed for the correctness proof of Algorithm POPQ.

Lemma 1: Given are a layer- k path $\mathbf{p} = \{v_0, \dots, v_n\}$, layer- $(k+1)$ subpaths $\mathbf{p}_1 = \{v_i, \dots, v_b\}$ and $\mathbf{p}_2 = \{v_b, \dots, v_j\}$ of \mathbf{p} , and an approximation ratio ε_k . The corresponding $\tilde{\varepsilon}$ -approximate cost functions $C_{(v_i, v_b)}^{\varepsilon}$ and $C_{(v_b, v_j)}^{\varepsilon}$ are stored in tables $D[v_i, v_b, c]$ and $D[v_b, v_j, c]$. Then, the execution of Procedure MERGE for $v_i, v_b, v_j, \delta_k, \delta_{k+1}$ and U yields a $\tilde{\varepsilon}$ -approximate cost function $C_{(v_i, v_j)}^{\tilde{\varepsilon}}(d)$, and stores it in table $D[v_i, v_j, c]$, where $\tilde{\varepsilon} = (1 + \varepsilon_k)(1 + \varepsilon) - 1$.

Proof: See [9]. ■

The following lemma establishes the computational complexity of Procedure MERGE.

Lemma 2: The computational complexity of Procedure MERGE is $O((\log U)/\varepsilon_k^2)$.

Proof: See [9]. ■

We proceed with the correctness proof of Algorithm POPQ.

Lemma 3: Algorithm POPQ identifies, for each subpath $\{v_i, \dots, v_j\}$ of layer k , $1 \leq k \leq K$, an $\varepsilon^{(k)}$ -approximate cost function $C_{(v_i, v_j)}^{\varepsilon}$ and stores it in the table $D[v_i, v_j, c]$, where $\varepsilon^{(k)} = \prod_{t=k}^K (1 + \varepsilon_t) - 1$.

Proof: By induction on the layer number k . Consider a layer- K subpath $\{v_i, v_{i+1}\}$. It is immediate that lines 2 and 3 compute an ε_K -approximate cost function and store it in the table $D[v_i, v_{i+1}, c]$. Assume inductively that the lemma holds for subpaths of layer $(k+1)$, and consider a layer- k subpath $\{v_i, \dots, v_j\}$. Since the lemma holds for the subpaths $\{v_i, \dots, v_b\}$ and $\{v_b, \dots, v_j\}$, $b = (j-i)/2$, the condition of Lemma 1 is satisfied for $\varepsilon^{(k+1)} = \prod_{t=k+1}^K (1 + \varepsilon_t) - 1$. Lemma 1 implies, in turn, that the algorithm identifies an $\varepsilon^{(k)}$ -approximate cost function $C_{(v_i, v_j)}^{\varepsilon}(d)$, for $\varepsilon^{(k)} = \prod_{t=k}^K (1 + \varepsilon_t) - 1$. ■

Corollary 1: Algorithm POPQ identifies, for each subpath $\{v_i, \dots, v_j\}$ of layer k , $1 \leq k \leq K$, an $\varepsilon/4$ -approximate cost function $C_{(v_i, v_j)}^{\varepsilon}$ and stores it in the table $D[v_i, v_j, c]$.

Proof: See [9]. ■

In the next lemma, we analyze the computational complexity of Algorithm POPQ.

Lemma 4: The computational complexity of Algorithm POPQ is $O((1/\varepsilon^2)n \cdot \log n \cdot \log U + n)$.

$\log U \log D$).

Proof: The computational complexity of the loop that begins on line 2 is $O(n \cdot \log U \log D)$. In order to determine the execution time of the loop that begins on line 5 we shall count the time needed for processing all subpaths of all layers. Lemma 2 implies that computing the cost function for a subpath of layer k requires $O((\log U)/\varepsilon_k^2) = O(2^{K-k} \cdot \log U/\varepsilon^2)$ time. As there are $n/2^k$ paths at layer k , we conclude that the time needed for processing all layer- k paths is $O((1/\varepsilon^2)n \cdot \log U)$. Since there are $K = O(\log N)$ layers, the total time for processing all subpaths of all layers is $O((1/\varepsilon^2)n \cdot \log n \cdot \log U)$. We conclude the computational complexity of the algorithm is $O((1/\varepsilon^2)n \cdot \log n \cdot \log U + n \cdot \log U \log D)$. ■

The above results are summarized by the following theorem.

Theorem 1: Algorithm POPQ precomputes, for each possible delay constraint $d, 1 \leq d \leq D$, an $(\varepsilon/4)$ -approximate solution to Problem OPQ within a computational complexity of $O((1/\varepsilon^2)n \cdot \log n \cdot \log U + n \cdot \log U \log D)$.

B. On-demand computation

In this section we present an algorithm for (fully) computing a suitable QoS partition on demand, *i.e.*, upon an incoming request. The algorithm comprises of the following steps. First, we obtain upper and lower U, L bounds on the solution to Problem OPQ such that $U/L \leq 2 \cdot n$. Then, we perform *linear scaling* and *rounding* of link cost functions, using the factor $L \cdot \varepsilon/n$. Finally, we apply Algorithm POPQ on the resulted graph.

As a first step, we compute sufficiently tight lower and upper bounds (L and U , respectively), by employing the algorithmic technique presented in [7]. Specifically, we seek a threshold c , such that allocating a budget c on each link results in a feasible partition, while the budget $c - 1$ is insufficient to support the end-to-end QoS requirement D . Note that such a threshold c constitutes a lower bound on the solution, while $c \cdot n$ constitutes an upper bound, thus we may set $L = c$ and $U = c \cdot n$. In order to satisfy $U/L \leq 2 \cdot n$ it is sufficient to obtain lower and upper bounds L_1, U_1 on c such that $U_1/L_1 \leq 2$.

The procedure begins by computing trivial lower and upper bounds on c (L_1 and U_1 , respectively), which are then improved in order to be sufficiently tight. Since, for a QoS requirement d_l of a link l under a feasible partition, it holds that $d_l \leq D$, and since $c_l(d)$ is a decreasing function of d , $L_1 = \sum_{l \in \mathbf{p}} c_l(D)$

is an obvious lower bound on c . Also, given a partition $\{d_l = D/n\}$, $U_1 = \max_{l \in \mathbf{p}} c_l(d_l)$ constitutes an obvious upper bound. The complexity of the bounding procedure depends on the ratio U_1/L_1 of the initial lower and upper bounds, which we denote by β .

Reduction of the ratio U_1/L_1 is achieved by performing binary search on the interval (L_1, U_1) in a logarithmic scale. At each iteration we test whether $c > t$, where $t = (U \cdot L)^{1/2}$. The testing procedure computes, for each $l \in \mathbf{p}$, the minimum value of the QoS requirement d_l that can be supported by allocating a budget t to l , and checks whether the resulted partition $\{d_l\}$ is feasible. The computational complexity of the test procedure is $O(n \log D)$. Since at each iteration, the ratio $x = U_1/L_1$ is reduced to \sqrt{x} , the number of tests required to reduce the ratio below 2 is $O(\log \log \beta)$. Thus, we need time $O(n \log \log \beta \log D)$ for finding suitable bounds L and U .

Having computed suitable bounds U and L , for which $U/L \leq 2 \cdot n$, we apply a scaling and rounding procedure on the link cost functions. To that end, a new cost function is defined for each link l , as follows:

$$c_l^*(d_l) = \left\lfloor \frac{c_l(d_l) \cdot n}{(\varepsilon/2) \cdot L} \right\rfloor. \quad (2)$$

After the rounding procedure, the new cost c^* of a partition with original cost c is bounded by

$$\frac{c \cdot n}{(\varepsilon/2) \cdot L} - n \leq c^* \leq \frac{c \cdot n}{(\varepsilon/2) \cdot L}. \quad (3)$$

Thus, the upper bound on the solution with respect to the new cost function is $U^* = \frac{n^2}{(\varepsilon/2)}$. Finally, the problem is solved by applying Algorithm POPQ to a path with the scaled cost functions $c_l^*(d_l)$. The algorithm is invoked with the upper bound $U^* = \frac{n^2}{(\varepsilon/2)}$ and the approximation ratio ε . The algorithm returns an $\frac{\varepsilon}{2}$ -approximate solution with respect to the new link costs. As we prove below, the cost of this solution under the original cost functions is at most $(1 + \varepsilon)$ times larger than that of the optimal solution. Algorithm OPQ, described in Figure 4, summarizes the above discussion.

We proceed to prove the correctness of Algorithm OPQ.

Lemma 5: Algorithm OPQ provides an ε -approximate solution to Problem OPQ.

Proof: It is easy to verify that, after executing line 8 of Algorithm OPQ, we have lower and upper bounds L, U for which it holds that $U/L \leq n$. We denote by c_{opt} and c_{opt}^* the cost of the optimal solution

```

Algorithm OPQ ( $\mathbf{p}, \{c_l\}, \varepsilon, D$ )
parameters
 $\mathbf{p} = \{v_0 = s, \dots, v_n = t\}$ -a QoS path;
 $\{c_l\}$ - the links' cost functions;
 $\varepsilon$  - approximation ratio;
 $D$  - QoS constraint.
1  $L \leftarrow \sum_{l \in \mathbf{p}} c_l(D)$ 
2  $U \leftarrow \max_{l \in \mathbf{p}} c_l(D/n)$ 
3 while  $U > 2 \cdot L$  do
4    $c \leftarrow (L \cdot U)^{1/2}$ 
5   for each  $e \in \mathbf{p}$  do
6      $d_l = \min\{d | c_l(d) \leq c\}$ 
7   if  $\sum_{l \in \mathbf{p}} d_l \leq D$  then  $L \leftarrow c$ 
8   else  $U \leftarrow c$ 
9 POPQ( $\mathbf{p}, \{\lfloor \frac{c_l(d_l) \cdot n}{(\varepsilon/2) \cdot L} \rfloor\}, \varepsilon, \frac{n^2}{(\varepsilon/2)}$ )
10  $\hat{c} \leftarrow \min\{c | D[v_0, v_n, c] \leq D\}$ 
11 return the corresponding partition

```

Fig. 4. Algorithm OPQ

under the original and scaled cost functions, respectively. Corollary 1 implies that, after executing Algorithm OPQ, it holds that $\hat{c}^* \leq (1 + \varepsilon/2)c_{opt}^*$, where $\hat{c}^* = \min\{c | D[s, t, c] \leq D\}$. Since $c_{opt}^* \leq \frac{c_{opt} \cdot n}{(\varepsilon/2) \cdot L}$, we have $\hat{c}^* \leq (1 + \varepsilon/2) \frac{c_{opt} \cdot n}{(\varepsilon/2) \cdot L}$. From the left part of Equation 3 it follows that the cost \hat{c} of the corresponding partition with respect to the original cost functions is at most $\hat{c} \leq \frac{\hat{c}^* (\varepsilon/2) \cdot L}{n} + (\varepsilon/2) \cdot L \leq (1 + \varepsilon/2) \cdot c_{opt} + (\varepsilon/2) \cdot L \leq (1 + \varepsilon) \cdot c_{opt}$.

We conclude that the algorithm returns a feasible partition whose cost is at most $(1 + \varepsilon)$ times more than the optimum, and the lemma follows. ■

We proceed to analyze the computational complexity of Algorithm OPQ.

Lemma 6: The computational complexity of Algorithm OPQ is $O(n \cdot \log D \log \log \beta + n \log n / \varepsilon \log D + (1/\varepsilon^2)n \cdot \log n \log(n/\varepsilon))$.

Proof: Lines 1 and 2 of the algorithm require $O(n)$ time. Each iteration of the loop of line 3 requires also $O(n \cdot \log D)$. Since the total number of iterations is $O(\log \log \beta)$, we conclude that the loop requires $O(n \cdot \log D \log \log \beta)$ time. Lemma 4 implies that the application of Algorithm OPQ for $U = 2n^2/\varepsilon$ (line 9) requires $O((1/\varepsilon^2)n \cdot \log n \cdot \log(n/\varepsilon) + n \log(n/\varepsilon) \log D)$ time. Thus, we conclude that the computational complexity of the algorithm is $O(n \cdot \log D \log \log \beta + (1/\varepsilon^2)n \cdot \log n \log(n/\varepsilon) + n \log(n/\varepsilon) \log D)$. ■

Note 1: If the link cost functions are reversible (*i.e.*, the computational complexity of finding, for a given \hat{c} , the minimum d for which $c_l(d) \leq \hat{c}$, is $O(1)$), then the computational complexity of our algorithm is $O(n \cdot \log \log \beta + n \log n / \varepsilon + (1/\varepsilon^2)n \cdot \log n \log(n/\varepsilon))$. The above results are summarized by the following

theorem.

Theorem 2: Algorithm OPQ provides an $O(n \cdot \log D \log \log \beta + (1/\varepsilon^2)n \cdot \log n \log(n/\varepsilon) + n \log(n/\varepsilon) \log D)$ ε -approximate solution to Problem OPQ, *i.e.*: given a connection request with delay constraint D , Algorithm OPQ identifies, in $O(n \cdot \log D \log \log \beta + (1/\varepsilon^2)n \cdot \log n \log(n/\varepsilon) + n \log(n/\varepsilon) \log D)$ steps, a suitable QoS partition $\{d_l\}_{l \in \mathbf{p}}$, whose cost is at most $(1 + \varepsilon)$ times larger than that of the optimal partition.

C. Discussion

We proceed to compare the performance of our algorithms with that of its alternatives.

We begin with the on-demand setting. In [7] and [2], the problem of partitioning of QoS constraints was considered, in a broader context of QoS routing with cost-dependent functions. The proposed algorithms, when applied to Problem OPQ, yield computational complexities of $O(n \log D \log \log \beta + n/\varepsilon \log(n/\varepsilon)(\log D + n/\varepsilon))$ and $O(\min(D, \frac{\log U}{\varepsilon} + \log D, \frac{n}{\varepsilon} + \log D) \cdot \frac{1}{\varepsilon} n^2 \log \log U)$, respectively. The dominant terms of these expression are $O(n^2 \log(n/\varepsilon)/\varepsilon^2)$ and $O((n^2 \log U)/\varepsilon^2)$, respectively, while the dominant term in our solution is $O(n \log n \log(n/\varepsilon)/\varepsilon^2)$. We thus conclude that the computational complexity of our algorithm is significantly ($\Omega(n/\log n)$) less dependent on the topology size than that of [7] and [2], which renders it more scalable for large topologies. This improvement has been achieved by exploiting the topological structure of unicast paths.

Next, we note that our algorithm can be applied also in the practically important case of discrete cost functions. Such functions have been the focus of [11], and an $O(mn^3 \log(m\varepsilon))$ algorithm was presented there, where $m = \sum_{l \in \mathbf{p}} m_l$ and m_l is the number of different delay values supported by link l . We conclude that, even if $m = O(n)$ (*i.e.*, each link supports a fixed number of delays), we achieve a major ($\Omega(n^3/\log n)$) reduction in terms of dependency on the topology size.

We described a precomputation scheme for Problem OPQ that provides ε -optimal solutions within a computational complexity of $O((1/\varepsilon^2)n \cdot \log n \cdot \log U + n \cdot \log U \log D)$ for the first phase and $O(\log \log U + \log(1/\varepsilon) + n)$ for the second phase. Compared with an on-demand scheme, the precomputation scheme allows to significantly reduce the time required to find a suitable partition. Indeed, with precomputation, the computational complexity of finding a suitable partition is dominated by the time

necessary to describe a partition ($O(n)$), *i.e.*, it is very close to the lower bound.

We note that a precomputation scheme can be trivially constructed out of any existing approximation algorithm for Problem OPQ (*e.g.*, [7], [2]), by just sequentially executing them for a certain range of delay values. Nonetheless, as it is easy to verify, the computational complexity of such simplistic solutions is significantly ($\Omega(n \log(1/\varepsilon)/\varepsilon \log n)$) higher than that of our solution.

V. MULTICAST: PROBLEM MOPQ

In this section we deal with the problem of QoS partition on multicast trees. Since we employ ideas that are quite similar to those of the unicast setting, we shall restrict ourselves to a brief discussion.

We begin by introducing the required definitions and terminology. A *multicast connection* uses a tree \mathbf{T} to interconnect the source s and the members of a multicast group $M = \{t_1, t_2, \dots\}$. For all $t_i \in M$ there exists a path \mathbf{p}_i between s and t_i on links that belong to the tree \mathbf{T} . Given a multicast tree \mathbf{T} , our goal is to (efficiently) allocate the delay on each link $l \in \mathbf{T}$ such that the end-to-end delay is satisfied for each member t_i of the multicast group. A QoS partition P_D on a multicast tree \mathbf{T} is a set of link delay requirements $\{d_l\}_{l \in \mathbf{T}}$, which satisfies, for each $t \in M$, the end-to-end delay requirement D , *i.e.*, $\sum_{l \in \mathbf{p}_i} d_l \leq D$ for each $t_i \in M$. Each link is associated with a cost function $c_l(d_l)$, which specifies the cost of supporting a delay requirement d_l . The cost of a QoS partition is the sum of the local costs, *i.e.*, $c(P_D) = \sum_{l \in \mathbf{T}} c_l(d_l)$.

The optimal QoS partition for a multicast tree is then defined as follows.

Problem MOPQ: (Multicast Optimal Partition of QoS) Given a tree \mathbf{T} and a QoS requirement D , find a QoS partition $\{d_l\}_{l \in \mathbf{T}}$ such that $\sum_{l \in \mathbf{p}_i} d_l \leq D$ for each $t_i \in M$ and $\sum_{l \in \mathbf{T}} c_l(d_l)$ is minimized.

For clarity of exposition, we use the following notation. The number of nodes and the depth of the multicast tree are denoted by n and H , respectively. The number of children of a node v_i are denoted by m_i . The subtree originating from the node $v_i \in \mathbf{T}$ is denoted by $\mathbf{T}_{(i,i)}$. A *branch* $\mathbf{T}_{(i,j)}$ of the subtree $\mathbf{T}_{(i,i)}$ is a subtree originating from v_i , which includes the link (v_i, v_j) outgoing from i and all descendants of v_j . We assume that all parameters (cost and delays) are integers.

We employ the following divide-and-conquer scheme. A multicast tree is recursively split into a number of disjoint subtrees. For each subtree and de-

lay value d , $1 \leq d \leq D$, we attempt to compute the optimal cost of partitioning d along the subtree. The solution to the original problem, *i.e.*, on the tree \mathbf{T} and for the end-to-end delay D , is obtained by recursively combining the solutions obtained for subtrees.

More specifically, consider a multicast tree \mathbf{T} , which is referred to as a *layer-0* tree. We split \mathbf{T} into a number of *layer-1* subtrees $\{T_{(i,i)}\}$, for each child node v_i of s . Then, for each value k , $k = 1, 2, \dots, H$, each *layer- k* subtree $T_{(j,j)}$ is split into a number of *layer- $(k+1)$* subtrees, for each child node of v_j . Clearly, *layer- H* subtrees comprise of just a single node and the number of subtrees n_k of layer k is $O(2^k)$.

We introduce the following *subtree cost functions*, which capture the cost of supporting various delay constraints by a subtree $T_{(i,i)}$.

Definition 3: The *optimal subtree cost function* $C_{(i,i)}^{opt}(d)$ of the subtree $\mathbf{T}_{(i,i)}$ of \mathbf{T} is defined as the cost of the optimal partition of the QoS requirement d along the subtree $\mathbf{T}_{(i,i)}$, *i.e.*,

$$C_{(i,i)}(d) = \min_{\{d_l\}_{l \in \mathbf{T}_{(i,i)}}} \left\{ \sum_{l \in \mathbf{T}_{(i,i)}} c_l(d_l) \mid \sum_{l \in \mathbf{p}_k} d_l \leq d \text{ for each } t_k \in \mathbf{T}_{(i,i)} \right\}.$$

Definition 4: An ε -*approximate subtree cost function* $C_{(i,i)}^\varepsilon(d)$ of a subtree $\mathbf{T}_{(i,i)}$ of \mathbf{T} is an (integer valued) function that satisfies, for each $d \geq 0$, $C_{(i,i)}^\varepsilon(d) \leq (1 + \varepsilon) \cdot C_{(i,i)}^{opt}(d)$.

An ε -approximate subtree cost function $C_{(i,i)}^{opt}(d)$ is constructed by using the logarithmic scaling approach, and is stored in a table $D[i, i, c]$. When no ambiguity exists, ε -approximate cost functions will be referred to as just cost functions.

Cost functions for branches $T_{(i,j)}$ of $T_{(i,i)}$ are defined similarly.

The computation of a subtree cost function introduces some inaccuracy ε_k at each layer k . We use the following assignment of $\{\varepsilon_k\}$:

$$\varepsilon_k = \varepsilon \sqrt{\frac{n_k}{4nH}}. \quad (4)$$

The value of the scaling factor δ_k for a layer k is assigned to be $\delta_k = 1 + \varepsilon_k$.

Again, for simplicity of exposition, we begin with the precomputation scheme, and then turn to consider the on-demand setting.

A. Precomputation scheme

The purpose of the precomputation scheme is to identify the cost function $C_{(s,s)}^\varepsilon$ and the corresponding partitions. The cost functions are computed in a

bottom-up manner, first for layer- H subtrees, then for layer- $(H - 1)$ subtrees, *etc.*, up to layer 0. Note that, since a layer- H subtree $\mathbf{T}_{(i,i)}$ comprises of a single node, its cost function $C_{(i,i)}^\varepsilon(d)$ is set to 0 for all d .

More specifically, given a layer- k subtree $\mathbf{T}_{(i,i)}$, we first compute, for each branch $\mathbf{T}_{(i,j)}$ of $\mathbf{T}_{(i,i)}$, the cost function $C_{(i,j)}^\varepsilon$. This function is obtained by *merging* the cost functions of the layer- $(k+1)$ subtree $\mathbf{T}_{(j,j)}$ and link (i, j) . Then, we merge the cost functions for all branches $\{\mathbf{T}_{(i,j)}\}$ of $\{\mathbf{T}_{(i,i)}\}$ into a cost function for $\mathbf{T}_{(i,i)}$. For this purpose we define two merging procedures, which we proceed to describe in more detail.

The merger of traversal functions of a link (v_i, v_j) and a subtree $\mathbf{T}_{(j,j)}$ is similar to the merger of the traversal functions of two subpaths, as discussed in Section IV-A.1. Accordingly, we use Procedure MERGE that appears on Fig. 2.

The purpose of the second procedure, referred to as MIN-MAX-MERGE, is to calculate the cost function $C_{(i,i)}^\varepsilon(d)$ of the subtree $\mathbf{T}_{(i,i)}$ out of the cost functions $C_{(i,j)}^\varepsilon(d)$ of its branches. In order to compute $C_{(i,i)}^\varepsilon(d)$, we attempt to find, for each cost value $c_t = \delta^t$, the minimum delay that can be supported by the subtree $\mathbf{T}_{(i,i)}$ subject to budget c_t . For this purpose we need to find the local budget $\{c^j\}$ for each branch $\mathbf{T}_{(i,j)}$ in such a way that the maximum delay between v_i and a terminal $t_i \in \mathbf{T}_{(i,i)}$ is minimized, and store the resulted value in the table $D[v_i, v_i, c]$, *i.e.*, $D[v_i, v_i, c_t] = \min_{c^j} \{\max_{(v_i, v_j) \in \mathbf{T}_{(i,i)}} \{D[v_i, v_j, c^j]\} \mid \sum c^j \leq c_t\}$. We observe that the required minimum delay value belongs to the set $S = \{D[v_i, v_j, c]\}_{(i,j) \in \mathbf{T}, c = \delta_{k+1}^\varepsilon}$.

Accordingly, our purpose is to find the smallest delay $\hat{d} \in S$ for which the cost $C_{(i,i)}^\varepsilon(\hat{d})$ of supporting \hat{d} in the subtree $\mathbf{T}_{(i,i)}$ is at most c_t . A straightforward solution would be to find, for each branch $\mathbf{T}_{(i,j)}$ of $\mathbf{T}_{(i,i)}$, the minimum $d \in \{D[v_i, v_j, c]\}$ for which $C_{(i,j)}^\varepsilon(d) \leq c_t$ and take the minimum among the branches. A more efficient procedure would be consider only a value $d \in \{D[v_i, v_j, c]\}$ for which $c_t \varepsilon_k / m_i \leq c \leq c_t$. This improves the running time to $O(m \log((1/\varepsilon_{k+1}) \log(m/\varepsilon_k)))$, but introduces some additional inaccuracy. However, this inaccuracy can be compensated by taking ε_k sufficiently low, which introduces no penalty in terms of computational complexity. The total computational complexity of Procedure MIN-MAX-MERGE

is $O((1/\varepsilon_k)m \log U \log((1/\varepsilon_k) \log(m/\varepsilon_k)))$. The formal description of Procedure MIN-MAX-MERGE appears on Fig. 5.

```

Procedure MIN-MAX-MERGE ( $\mathbf{T}, v_i, \delta_{k+1}, \delta_k, U$ )
1   $d_{min} \leftarrow 0$ 
2  for  $t = 1$  to  $\lceil \log_{\delta_k} U \rceil$  do
3     $c_t \leftarrow \delta_k^t$ 
4    for each child  $j$  of  $i$  do
5       $S \leftarrow \{D[v_i, v_j, c] \mid \varepsilon \cdot c_t / m_i \leq c \leq c_t\}$ 
6       $d \leftarrow \min\{d \in S \mid \sum_{(v_i, v_j) \in \mathbf{T}} \min\{c \mid D[v_i, v_j, c] \leq \hat{d}\} \leq c_t\}$ 
7     $d_{min} = \min\{d, d_{min}\}$ 

```

Fig. 5. Procedure MIN-MAX-MERGE

We proceed to describe Algorithm PMOPQ. Given a tree $\mathbf{T}_{(i,i)}$ with root $s = v_i$, we recursively compute the cost functions $C_{(j,j)}^\varepsilon(d)$ for each child v_j of v_i . Then, we compute, for each child v_j of v_i , the cost function of the branch $C_{(i,j)}^\varepsilon(d)$ of $\mathbf{T}_{(i,i)}$ by merging the subtree cost function $C_{(v_j, v_j)}^\varepsilon(d)$ and the link cost function $c_{(i,j)}(d)$ (Procedure MERGE). Finally, we compute the the cost function $C_{(v_i, v_i)}^\varepsilon(d)$ by merging traversal functions $C_{(v_i, v_j)}^\varepsilon(d)$ (Procedure MIN-MAX-MERGE). The formal specification of Algorithm PMOPQ appears in Fig. 6.

```

Algorithm PMOPQ ( $\mathbf{T}, s, \{c_e\}, \varepsilon, U$ )

```

parameters

\mathbf{T} - the multicast tree;
 s - the root of \mathbf{T} ;
 $\{c_e\}$ - link cost functions;
 ε - approximation ratio;
 U - the upper bound on the cost of an optimal partition.

```

1  for  $k \leftarrow 1$  to  $H$  do
2     $n_k \leftarrow$  the number of nodes of layer- $k$ 
3     $\delta_k \leftarrow 1 + \sqrt{n_k \cdot \varepsilon / 4nH}$ 
4    for each  $l = (v_i, v_j) \in \mathbf{T}$  do
5       $k \leftarrow$  the layer of  $v_i$ 
6      for each  $c_t \in \{\delta_k^t\}$  do
7         $D[v_i, v_j, c_t] = \min\{d \mid c_l(d) \leq c_t\}$ 
8    PROPAGATE( $s$ )

```

Procedure PROPAGATE (v_i)

```

1   $k \leftarrow$  the layer of  $v_i$ 
2  for each  $(v_i, v_j) \in \mathbf{T}$  do
3    PROPAGATE( $v_j$ )
4    MERGE( $v_i, v_i, v_j, \delta_k, \delta_{k+1}, U$ ).
5  MIN-MAX-MERGE( $\mathbf{T}, v_i, \delta_k, \delta_{k+1}, U$ )

```

Fig. 6. Algorithm PMOPQ

Upon a request with some QoS requirement D , the optimal partition is promptly identified by examining the output of Algorithm PMOPQ. Specifically, $c(d) = \min\{c = \delta^i \mid D[s, s, c] \leq \hat{d}\}$ and it can be easily identified through a binary search, which requires $O(\log \log U + \log(1/\varepsilon) + n)$ time.

B. Analysis of the precomputation scheme

Since the analysis of the precomputation scheme is very similar to the unicast setting, we only highlight it; the detailed proofs can be found in [9].

First, it can be shown that an invocation of Procedure MIN-MAX-MERGE for a layer- k node v_i introduces an inaccuracy of ε_k , *i.e.*, given an $\bar{\varepsilon}$ -approximate cost function for each layer- $(k+1)$ subtree of $\mathbf{T}_{(i,i)}$, Procedure MIN-MAX-MERGE yields an $\tilde{\varepsilon}$ -approximate cost function $C_{v_i, v_i}^{\tilde{\varepsilon}}(d)$ for the subtree $\mathbf{T}_{(v_i, v_i)}$, where $\tilde{\varepsilon} = (1 + \varepsilon_k)(1 + \bar{\varepsilon}) - 1$. The same holds for Procedure MERGE (Lemma 1). Then, it can be shown, by induction on the layer number, that Algorithm PMOPQ identifies an $\bar{\varepsilon}$ -approximate cost function $C_{(s,s)}^{\bar{\varepsilon}}$ for $\mathbf{T}_{(s,s)}$ and stores it in the table $D[s, s, c]$, where $\bar{\varepsilon} = \prod_{k=1}^H (1 + 2\varepsilon_k) - 1 \approx \sum_{k=1}^H 2\varepsilon_k$. Substituting ε_k from Equation 4 we get $\bar{\varepsilon} \leq \varepsilon \frac{\sum_{k=1}^H \sqrt{n_k}}{\sqrt{nH}} \leq \varepsilon \frac{H\sqrt{n/H}}{\sqrt{nH}} = \varepsilon$.

We proceed to discuss the computational complexity of Algorithm PMOPQ. First, we count the time needed to process all subtrees of layer- k . For each subtree $\mathbf{T}_{(i,i)}$ we invoke procedures MERGE and MIN-MAX-MERGE. It can be verified that the average running time needed for processing a subtree is dominated by that of Procedure MERGE, which is $O((\log U)/\varepsilon_k^2) = O((1/\varepsilon^2)n \cdot H \log U/n_k)$. Thus, processing all subtrees of layer- k requires $O(n \cdot H \log U)$ running time. In addition, $O(n \cdot \log U \log D)$ time is required for processing the loop that begins on line 4. We conclude that the computational complexity of Algorithm PMOPQ is $O((1/\varepsilon^2)n \cdot H^2 \log U + n \cdot \log U \log D)$.

The above results are summarized by the following theorem.

Theorem 3: Algorithm PMOPQ precomputes, for each possible delay constraint $d, 1 \leq d \leq D$, an ε -approximate solution to Problem MOPQ within a computational complexity of $O((1/\varepsilon^2)n \cdot H^2 \log U + n \cdot \log U \log D)$.

C. On-demand computation

We proceed to discuss the on-demand setting, in which a suitable QoS partition is computed upon an incoming request with some delay constraint D . We begin by determining a threshold c , such that allocating the budget c to each link results in a feasible partition, *i.e.*, the delay between s and each $t_i \in M$ is at most D . Obvious lower and upper bounds L_1, U_1 on the threshold are $L_1 = \sum_{l \in \mathbf{T}} c_l(D)$, $U_1 = \max_{l \in \mathbf{T}} \{c_l(D/H)\}$. We denote U_1/L_1 by β . These

obvious bounds can be improved by performing a binary search in a logarithmic scale. Such a search requires $O(n \log \log \beta \log D)$ time and reduces the ratio U_1/L_1 to at most 2.

Having computed the bounds on the threshold c , we derive lower and upper bounds L, U on the cost \hat{c} of the optimal solution to Problem MOPQ. Since c constitutes a lower bound on \hat{c} , while $c \cdot n$ constitutes an upper bound, we may set $L = L_1$ and $U = n \cdot U_1$. Note that $U/L \leq 2n$. Finally, we apply Algorithm PMOPQ with respect to scaled cost functions $c_l^*(d_l) = \lfloor \frac{c_l(d_l) \cdot n}{(\varepsilon/2) \cdot L} \rfloor$, which incurs a computational complexity of $O((1/\varepsilon^2)nH^2 \log(n/\varepsilon) + n \cdot \log(n/\varepsilon) \log D)$. A suitable partition is then derived from the output of Algorithm PMOPQ: $\hat{c} = \min\{c | D[s, s, c] \leq D\}$.

The above results are summarized by the following theorem.

Theorem 4: Given a connection request with delay constraint D , a suitable QoS partition $\{d_l\}_{l \in \mathbf{T}}$, whose cost is at most $(1 + \varepsilon)$ times larger than that of the optimal partition, can be identified in $O((1/\varepsilon^2)n \cdot H^2 \log(n/\varepsilon) + n \cdot \log(n/\varepsilon) \log D + n \log \log \beta \log D)$ steps.

D. Discussion

We proceed to compare the performance of our algorithms with that of its alternatives.

The on-demand setting was considered in [7], where an ε -approximate solution to Problem MOPQ was presented. That algorithm yields a computational complexity of $O(n \log D \log \log \beta + n^2(\log D + n) \log \log H + n^2/\varepsilon(\log D + n/\varepsilon))$. The dominant term of this expression is $O(n^3/\varepsilon^2)$, while the dominant term of our solution is $O((1/\varepsilon^2)n \cdot H^2 \log(n/\varepsilon))$. It follows that, for most practical settings *i.e.*, when H is lower than n , the computational complexity of our algorithm is significantly $(\Omega(\frac{n^2}{H^2 \log(n/\varepsilon)}))$ less dependent on the topology size than that of [7]. Moreover, we note that the depth H of a typical multicast tree is $O(\log n)$, in which case our algorithm is $\Omega(\frac{n^2}{\log^2 n \log(n/\varepsilon)})$ times faster.

We described a precomputation scheme for Problem MOPQ that provides ε -optimal solutions within a computational complexity of $O((1/\varepsilon^2)n \cdot H^2 \log U + n \cdot \log U \log D)$ for the first phase and $O(\log \log U + \log(1/\varepsilon) + n)$ for the second phase. This precomputation scheme allows to promptly provide a suitable partition upon an incoming request. The computational complexity of our scheme is significantly

($\Omega(n^2/\varepsilon H^2)$) lower than that of simplistic adaptations of existing approximation algorithms.

VI. CONCLUSIONS

A fundamental problem in the support of QoS in networks is how to allocate resources along the connection's topology such that the required QoS can be guaranteed at minimum cost. This immediately translates into the optimization problem that has been the focus of this study, namely, how to optimally partition the end-to-end QoS requirement into local requirements. This problem poses major challenges in terms of algorithmic design, and has been the subject of several recent studies. These studies provided significant insight into the essence of the problem and its potential solutions. However, the solutions that have been proposed either relied on restrictive assumptions (such as convexity), or else proposed approximation schemes whose complexity considerably depended on the network size. Therefore, a scalable approach, which would be adequate for large scale networks, was called for. Such an approach should bear a smaller dependence on the size of the connection's topology, and, ultimately, provide a fast answer to the partition problem upon each incoming connection request.

Accordingly, in this study we considered the scalability perspective, taking two independent approaches. First, we proposed a novel algorithmic technique, which exploits the specific structure of the actual topologies on which connections are established, *i.e.* paths or trees. This technique resulted in a significant improvement in terms of computational complexity, in particular dependence on the size of the topology. Indeed, for the "on-demand" setting, our approach typically offers almost-linear solutions, both for unicast and for multicast, in terms of dependence on topology size. These results *per se* constitute a significant improvement upon previous solutions. Second, we devised a *precomputation scheme*. This scheme is based on the observation that, typically, network elements have the resources to perform much computation in advance. Hence, it enables to obtain fast solutions immediately upon each incoming connection request; in particular, at that time (*i.e.*, at the "second phase"), the computational complexity depends *only linearly* on the size of the topology, be it a unicast path or a multicast tree.

Several enhancements and extensions of this study are possible. For example, our layering approach allows to easily distribute the computational effort among network nodes. Indeed, at each layer, each

component (subpath or subtree) is processed independently, hence the processing can be performed concurrently, at different nodes. Based on this observation, in [9] we present distributed versions of our algorithms.

More generally, the schemes presented in this study can serve to tackle the scalability issue in other important networking problems. In particular, another fundamental problem in the context of QoS provision is that of QoS routing, *i.e.*, the proper selection of the connection's topology. In [9] we apply our schemes in order to provide efficient solutions for the QoS routing problem in large-scale networks. The key observation there is that large-scale networks typically bear a hierarchical layering structure, which provides the grounds for an efficient application of our divide-and-conquer approach.

REFERENCES

- [1] S. Blake. An architecture for Differentiated Services. - RFC No. 2475. Internet Engineering Task Force, December 1998.
- [2] L. Zhang F. Ergun, R. Sinha. QoS Routing with Performance-Dependent Costs. In *Proceedings of IEEE INFOCOM'00*, Tel-Aviv, Israel, March-April 2000.
- [3] V. Firoiu and T. Towsley. Call admission and resource reservation for multicast sessions. In *Proceedings of IEEE INFOCOM'96*, San-Francisco, CA, April 1996.
- [4] R. Guérin and A. Orda. QoS-based routing in networks with inaccurate state and metrics information. *IEEE/ACM Transactions on Networking*, 7(3):350–364, June 1999.
- [5] M. Kodialam and S. Low. Resource Allocation in a Multicast Tree. In *Proceedings of IEEE INFOCOM'99*, New York, NY, March 1999.
- [6] D. H. Lorenz and A. Orda. Optimal Partition of QoS Requirements on Unicast Paths and Multicast Trees. In *Proceedings of IEEE INFOCOM'99*, New York, NY, March 1999.
- [7] D.H. Lorenz, A. Orda, D. Raz, and Y. Shavitt. Efficient QoS Partition and Routing of Unicast and Multicast. In *Proceedings IEEE/IFIP IWQoS*, Pittsburgh, PA, June 2000.
- [8] A. Orda and A. Sprintson. QoS Routing: the Precomputation Perspective. In *Proceedings of IEEE INFOCOM'00*, Tel-Aviv, Israel, March-April 2000.
- [9] A. Orda and A. Sprintson. A Scalable Approach to the Partition of QoS Requirements in Unicast and Multicast. EE Pub. No. 1281, Department of Electrical Engineering, Technion, Haifa, Israel, June 2001. Available from: <ftp://ftp.technion.ac.il/pub/supported/ee/Network/os01.ps>.
- [10] Private Network-Network Interface Specification v1.0 (PNNI). ATM Forum Technical Committee, March 1996.
- [11] D. Raz and Y. Shavitt. Optimal partition of qos requirements with discrete cost functions. *IEEE Journal on Selected Areas in Communications*, 18(12):2593–2602, December 2000.