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

Ariel Orda, *Senior Member, IEEE,* and Alexander Sprintson, *Member, IEEE*

***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 and tree for multicast). As networks grow in size, the scalability of the solution becomes increasingly important. This calls for 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.

***Index Terms*—Multicast, performance-dependent costs, quality of service (QoS) partition, resource allocation, routing.**

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 resources (e.g., bandwidth and 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., according to the ATM PNNI recommendations [2]), 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*. For example, Fig. 1(b) shows an aggregated network that corresponds to the original network depicted on Fig. 1(a). Each link in the aggregated network is associated with a cost-delay function that represents the corresponding subnetwork.

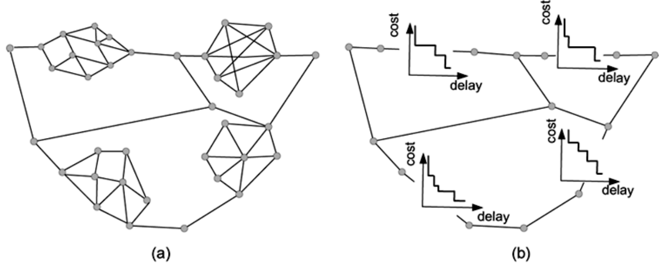

Fig. 1. (a) Original network. (b) Aggregated network; the subnetworks are represented by link cost functions.

The problem of optimal partition of QoS requirements was formulated in [3] and has been the subject of several studies [4]–[8]. Efficient optimal solutions for the special case of *convex* cost functions for both unicast and multicast were established in [3]. 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., $\mathcal{NP}$-hard [3]), suitable approximation schemes were presented in [4], [7], and [8]. While the computational complexity of those 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 better exploits the specific structure of routing

Manuscript received December 10, 2002; revised January 18, 2004, July 18, 2004, and December 17, 2004; approved by IEEE/ACM Transactions on Networking Editor M. Zukerman.

A. Orda is with the Department of Electrical Engineering, Technion—Israel Institute of Technology, Haifa 32000, Israel (e-mail: ariel@ee.technion.ac.il).

A. Sprintson is with the Department of Engineering and Applied Science, California Institute of Technology, Pasadena, CA 91125 USA (e-mail: spalex@caltech.edu).

Digital Object Identifier 10.1109/TNET.2005.857122

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. With our methods, the computational load can be efficiently distributed along network nodes. 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 [9], [10] (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 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.

Precomputation is performed by means of a two-phase procedure, referred to as a *precomputation scheme*. The first phase is executed in advance and its purpose is to precompute the optimal partition *a priori*, for each delay constraint supported by the path or tree. The computations performed at this phase are then summarized into a database 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.

The rest of this paper is organized as follows. In Section II, we formulate the network model and formally state the considered problems. Section III deals with unicast topologies and presents solutions both for performing on-demand computation as well as precomputation. Section IV presents similar solutions for the much more complex setting of multicast. Finally, conclusions are presented in Section V.

II. Model and Problem Formulation

This section formulates the general model and main problems addressed in this paper. For clarity of presentation, we focus here on unicast; the definitions and terminology for multicast are presented in Section IV.

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 $\mathcal{P}=\{v_0,v_1,\ldots,v_n\}$, such that, for $0\le i\le n-1, (v_i,v_{i+1})\in E$; $n=|\mathcal{P}|$ is then said to be the *number of hops* (*or hop count*) of $\mathcal{P}$. The subpath of $\mathcal{P}$ that extends from v_i to v_j is denoted by $\mathcal{P}_{(v_i,v_j)}$. We assume that the connection's topology, i.e., a path $\mathcal{P}$, is given.

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

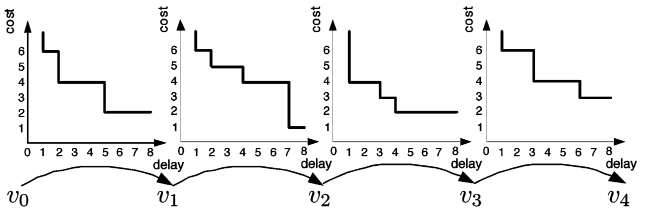

Fig. 2. Instance of Problem OPQ.

guarantees supported by link l. A QoS partition on a unicast path $\mathcal{P}$ is a set $\{d_l\}_{l\in\mathcal{P}}$ 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 on a path $\mathcal{P}$ is a set $\{d_l\}_{l\in\mathcal{P}}$ such that $\sum_{l\in\mathcal{P}}d_l\le D$. 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* $c_l(d)$ which assigns a cost to each delay guarantee d that the link offers. We assume the $c_l(d)$ is higher for tighter delay constraints, i.e., the function $c_l(d)$ is monotonically decreasing. For clarity of presentation, we assume that, if a delay guarantee d is not supported by a link l, then $c_l(d)=\infty$. 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 and its location. The link cost function can be specified by either an algebraic expression or by a table that specifies costs for supporting various delay guarantees. In the latter case, we say it is a *discrete* cost function. We shall assume that all parameters (both delay guarantees and costs) are (positive) integers. The overall cost of a partition $\{d_l\}_{l\in\mathcal{P}}$ is the sum of the local costs, i.e., $\sum_{l\in\mathcal{P}}c_l(d_l)$.

The optimal QoS partition problem is then defined as follows.

Problem OPQ (Optimal Partition of QoS): Given a path $\mathcal{P}=\{v_0,\ldots,v_n\}$ and a delay constraint D, find a QoS partition $\{d_l\}_{l\in\mathcal{P}}$ such that $\sum_{l\in\mathcal{P}}d_l\le D$ and $\sum_{l\in\mathcal{P}}c_l(d_l)$ is minimized.

The solution $\{d_l\}_{l\in\mathcal{P}}$ of Problem OPQ is referred to as an *optimal partition of a QoS requirement D along* $\mathcal{P}$.

Fig. 2 demonstrates an instance of Problem OPQ. Suppose we need to establish a connection with a delay requirement 8 between v_0 and v_4. For this purpose, we use path $\mathcal{P}=\{v_0,\ldots,v_4\}$, with link cost functions, as depicted in Fig. 2. The optimal partition for this instance is $\{2,2,1,3\}$, i.e., the delay requirement for the first link is 2, for the second link is 2, etc. The cost of the optimal partition is 17.

As mentioned in the Introduction, we devise efficient schemes for precomputation of optimal partitions for a wide range of delay constraints. The related problem is defined as follows.

Problem POPQ (Precomputation of OPQ): Given a path $\mathcal{P}=\{v_0,\ldots,v_n\}$, find, *for each delay requirement D*, a QoS partition $\{d_l\}_{l\in\mathcal{P}}$ such that $\sum_{l\in\mathcal{P}}d_l\le D$ and $\sum_{l\in\mathcal{P}}c_l(d_l)$ is minimized.

For clarity of presentation, we make the following simplifying assumptions.

1) The number of links n in path $\mathcal{P}$ is a power of 2, i.e., $n = 2^K$, for some integer K.
2) Given delay constraint d, the cost $c_l(d)$ of supporting d by link l can be computed in $\mathcal{O}(1)$ time.
3) Given a cost c, the minimum delay constraint d supported by link l at cost c, can be computed in $\mathcal{O}(1)$ time.

Dropping Assumption 1 requires a mild and straightforward modification of our results, with no penalty in terms of computational complexity. Assumptions 2 and 3 require that the functions $c_l(d)$ and the corresponding inverse functions can be easily computed. Since a value of the inverse function can be computed through binary search, dropping Assumption 3 results in a small penalty in terms of computational complexity.

The computational complexity of our solutions depends on the maximum cost $C^{\max}$ of supporting a delay constraint by a link in $\mathcal{P}$. More specifically, let $d_l^{\min}$ be the minimum delay constraint supported by a link l, i.e., $d_l^{\min} = \min\{d \,|\, c_l(d) \neq \infty\}$. Then $C^{\max} = \max_{l\in\mathcal{P}}\{c_l(d_l^{\min})\}$. For example, in the instance of Problem POPQ depicted in Fig. 2 we have $d_l^{\min} = 1$ and $C^{\max} = 6$.

In general, Problem OPQ and Problem POPQ are intractable, i.e., $\mathcal{NP}$-hard [3]. Accordingly, in this work we resort to scalable ε-approximate solutions for fixed $0 < \varepsilon \leq 1$, i.e., solutions of (low) polynomial complexity, whose cost is at most $(1+\varepsilon)$ times higher than the cost of the optimal solution. Specifically, we present algorithms for Problems OPQ and POPQ whose computational complexity is $\mathcal{O}(\frac{1}{\varepsilon^2} n \log(\frac{n}{\varepsilon}) + n \log\log C^{\max})$ and $\mathcal{O}(\frac{1}{\varepsilon^2} n \log(n C^{\max}))$, respectively.

III. QoS Partition for Unicast

In this section, we deal with the partition of QoS requirements along unicast paths. We begin by presenting our novel approximation approach. Then, we present approximation schemes for Problems OPQ and POPQ.

A. Our Approach

We observe that the optimal solution to the problem of QoS partition on a path contains within it optimal solutions for its subpaths. For example, in Fig. 2, the optimal partition $\{2,2,1,3\}$ of delay constraint 8 for path $\mathcal{P} = \{v_0,\ldots,v_4\}$ contains within it the optimal partition $\{2,2\}$ of delay constraint 4 for the subpath $\mathcal{P}_{(v_0,v_2)}$. Accordingly, we compute the solutions to Problems OPQ and POPQ in the following "divide-and-conquer" fashion. We recursively split the given unicast path $\mathcal{P}$ into two disjoint subpaths. We compute the set of delay guarantees supported by each subpath at different costs. These delay guarantees and the corresponding partitions are summarized by means of *delay functions*, defined below. We then obtain a solution to the original problem, i.e., a partition of delay constraint D on path $\mathcal{P}$, by recursively combining the delay functions obtained for the subpaths.

1) Delay Functions: We begin by defining a new structure, namely, *optimal delay functions*, whose purpose is to summarize the delay guarantees that can be offered by subpaths at different costs.

Definition 1: The *optimal delay function* $D^{\text{opt}}_{(v_i,v_j)}(c)$ of a subpath $\mathcal{P}_{(v_i,v_j)}$ of $\mathcal{P}$ is defined as the minimum delay requirement D supported by $\mathcal{P}_{(v_i,v_j)}$ at cost c, i.e.,

$$D^{\text{opt}}_{(v_i,v_j)}(c) = \min\left\{ D \,|\, \exists\, \{d_l\}_{l\in\mathcal{P}_{(v_i,v_j)}} \text{ such that } \sum_{l\in\mathcal{P}_{(v_i,v_j)}} d_l \leq D \text{ and } \sum_{l\in\mathcal{P}_{(v_i,v_j)}} c_l(d_l) \leq c \right\}. \quad (1)$$

Note that, if the subpath consists of a single link (v_i, v_{i+1}), then $D^{\text{opt}}_{(v_i,v_{i+1})}(c)$ is the inverse of the cost function of that link, i.e., $D^{\text{opt}}_{(v_i,v_{i+1})}(c) = \min\{d \,|\, c_{(v_i,v_{i+1})}(d) \leq c\}$.

While optimal delay functions accurately capture the delays supported by subpaths of $\mathcal{P}$ at different costs, they are impractical, since their computation is intractable and, moreover, their storage requirements are prohibitively large. Accordingly, we use approximate delay functions that allow the cost of supporting a delay constraint to be slightly higher than the optimum.

Definition 2: A delay function $D_{(v_i,v_j)}(c)$ of a subpath $\mathcal{P}_{(v_i,v_j)}$ is said to be *ε-approximate*, for some constant ε, if for each $c \geq 0$ it holds that

$$D_{(v_i,v_j)}(c\cdot(1+\varepsilon)) \leq D^{\text{opt}}_{(v_i,v_j)}(c). \quad (2)$$

Given a ε-approximate delay function $D_{(v_i,v_j)}(c)$, we refer to ε as the *approximation error* of $D_{(v_i,v_j)}(c)$.

Let D be an arbitrary delay constraint. We denote $\hat{c} = \min\{c \,|\, D_{(v_i,v_j)}(c) \leq D\}$ and $c^{\text{opt}} = \min\{c \,|\, D^{\text{opt}}_{(v_i,v_j)}(c) \leq D\}$, i.e., $\hat{c}$ and c^{opt} are the costs of supporting the delay constraint D according to the functions $D_{(v_i,v_j)}(c)$ and $D^{\text{opt}}_{(v_i,v_j)}(c)$, respectively. Equation (2) implies that $\hat{c} \leq (1+\varepsilon)c^{\text{opt}}$, i.e., the cost $\hat{c}$ of supporting the delay constraint according to the function $D_{(v_i,v_j)}(c)$ is at most $(1+\varepsilon)$ times higher than the optimum.

In the next section, we construct approximate delay functions whose computation and storage requirements are feasible. For the sake of clarity, and when no ambiguity exists, ε-approximate delay functions shall be referred to as just *delay functions*.

2) Logarithmic Sampling: Approximate delay functions can be constructed from optimal delay functions by employing *logarithmic sampling*. The idea is to sample the optimal delay function at cost values $\{1, 1+\delta, (1+\delta)^2, \ldots\}$, where δ is a constant, referred to as an *approximation parameter*. In general, δ must be sufficiently small in order to compute an approximate delay function with small approximation error ε.

For each cost c, $(1+\delta)^i \leq c < (1+\delta)^{i+1}$, and for each $i \geq 0$, the value of the approximate delay function at cost c is equal to the optimal delay function at cost $(1+\delta)^i$, i.e., $D_{(v_i,v_j)}(c) = D^{\text{opt}}_{(v_i,v_j)}(c')$, where $c' = \max\{(1+\delta)^t \,|\, (1+\delta)^t \leq c, t = 1, 2, \ldots\}$.

For example, consider the optimal delay function $D^{\text{opt}}_{(v_i,v_j)}(c)$ depicted in Fig. 3(a). The function is sampled at five cost values $\{1, 1+\delta, \ldots (1+\delta)^4\}$. The resulting approximate delay function $D_{(v_i,v_j)}(c)$, depicted in Fig. 3(b), is a piecewise-constant function whose segments correspond to the values $\{d_1,\ldots,d_4\}$ of the optimal function at sampled costs. In our approximation

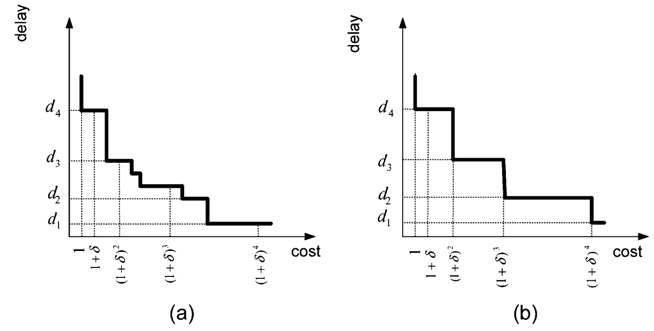

Fig. 3. (a) Optimum delay function. (b) Approximate delay function.

schemes, we store a delay function $D_{(v_i,v_j)}(c)$ by keeping the values of cost and delay for each segment of $D_{(v_i,v_j)}(c)$.

3) Layers: We proceed to describe our approach in more detail. Consider a unicast path $\mathcal{P} = \{v_0, \ldots, v_n\}$, which is referred to as a *layer-0* path. Recall that our assumption is that $n = 2^K$. We split $\mathcal{P}$ into two *layer*-1 subpaths $\mathcal{P}_{(v_0,v_b)}$ and $\mathcal{P}_{(v_b,v_n)}$, where $b = n/2$. Then, for each value $k, k = 1, 2, \ldots, K-1$, each layer-k subpath $\mathcal{P}_{(v_i,v_j)}$ is split into two layer-$(k+1)$ subpaths $\mathcal{P}_{(v_i,v_b)}$ and $\mathcal{P}_{(v_b,v_j)}$, where $b = (i+j)/2$. Note that a layer-K subpath consists of just a single link. Clearly, the number of subpaths of a layer k is $\mathcal{O}(2^k)$.

The goal of our scheme is to compute, for each layer $k, 0 \le k \le K$, the delay functions of layer-k subpaths. We begin by computing the delay functions of subpaths of layer K. Since these subpaths consist of just a single link, their delay functions can be obtained by applying logarithmic sampling on the inverted cost functions for corresponding links. Then, for each $k, 1 \le k \le K-1$, we compute the delay functions of layer-k subpaths by *merging* previously computed delay functions for subpaths of layer-$(k+1)$. The merging procedure is discussed in detail in Section III.A.4.

The computation of a delay function introduces some error at each layer, which accumulates as we proceed to lower layers. The error depends on the approximation parameter δ used in the logarithmic sampling process. The key idea of our scheme is to use different approximation parameters δ_k for different layers. The values δ_k are chosen in a way that minimizes the computational complexity of the algorithm, while ensuring that the total accumulated error is at most ε. The assignment of δ_k is discussed in detail in Section III-A5.

4) Procedure MERGE: The merging procedure receives, as input, the delay functions $D_{(v_i,v_b)}(c)$ and $D_{(v_b,v_j)}(c)$ of two layer-$(k+1)$ subpaths, $\mathcal{P}_{(v_i,v_b)}$ and $\mathcal{P}_{(v_b,v_j)}$, and an upper bound U. The procedure computes the values of delay function $D_{(v_i,v_j)}(c)$ of layer-k subpath $\mathcal{P}_{(v_i,v_j)}$ for each $1 \le c \le U$. To that end, it computes for each $c, 1 \le c \le U$, the partition (c_1, c_2) of a budget c between the subpaths, which minimizes the delay of the subpath $\mathcal{P}_{(v_i,v_j)}$ under budget constraint c.

A straightforward solution would be to examine all possible partitions (c_1, c_2) of the budget c. Since the choice of c_1 determines c_2, it is sufficient to consider each $c_1 \le c$. Moreover, since the delay function $D_{(v_i,v_b)}(c)$ is obtained by logarithmic sampling at costs $\{1, 1+\delta_{k+1}, (1+\delta_{k+1})^2, \ldots\}$, only these costs should be considered.

Procedure MERGE *$(D_{(v_i,v_b)}(c), D_{(v_b,v_j)}(c), \delta_k, \delta_{k+1}, U)$:*

```
parameters
    $D_{(v_i,v_b)}(c)$- the delay function for subpath $\mathcal{P}'_{(v_i,v_b)}$
    $D_{(v_b,v_j)}(c)$- the delay function for subpath $\mathcal{P}'_{(v_b,v_j)}$
    $\delta_k$- the approximation parameter for layer-$k$
    $\delta_{k+1}$ - the approximation parameter for layer-$(k+1)$
    $U$- the upper bound on the cost of a partition.
1  $c = 1$
2  while $c \le U$ do
3      $D_{(v_i,v_j)}(c) \leftarrow D_{(v_i,v_b)}(c/2) + D_{(v_b,v_j)}(c/2)$
4      $c'_2 \leftarrow \min_t\{(1+\delta_{k+1})^t \mid (1+\delta_{k+1})^t > c/2\}$
5      $c_2 \leftarrow c'_2$
6      while $c_2 \le c$ do
7          $D_{(v_i,v_j)}(c) \leftarrow \min\{D_{(v_i,v_j)}(c), D_{(v_i,v_b)}(c-c_2) + D_{(v_b,v_j)}(c_2)\}$
8          $c_2 \leftarrow c_2 \cdot (1+\delta_{k+1})$
9      $c'_1 \leftarrow \min_t\{(1+\delta_{k+1})^t \mid (1+\delta_{k+1})^t > c/2\}$
10     $c_1 \leftarrow c'_1$
11     while $c_1 \le c$ do
12         $D_{(v_i,v_j)}(c) \leftarrow \min\{D_{(v_i,v_j)}(c), D_{(v_i,v_b)}(c_1) + D_{(v_b,v_j)}(c-c_1)\}$
13         $c_1 \leftarrow c_1 \cdot (1+\delta_{k+1})$
14     $c \leftarrow (1+\delta_k) \cdot c$
15 return $D_{(v_i,v_j)}(c)$
```

Fig. 4. Procedure MERGE.

The merging can be performed more efficiently by the following procedure. We divide the set $S = \{(c_1, c_2) \mid c_1 + c_2 \le c\}$ of feasible partitions into three subsets S_1, S_2 and S_3:

1) the subset S_1 includes the partitions for which $c_1 \le c/2$ and $c_2 \le c/2$;
2) the subset S_2 includes the partitions for which $c_1 < c/2$ and $c_2 > c/2$;
3) the subset S_3 includes the partitions for which $c_1 > c/2$ and $c_2 < c/2$.

Then we identify, for each subset, the partition that minimizes the delay of the subpath $\mathcal{P}_{(v_i,v_j)}$. For the subset S_1, we note that it is sufficient to examine the partition $(c/2, c/2)$. For the subset S_2, it is sufficient to examine partitions for which values of c_2 correspond to costs $\{(1+\delta_{k+1})^t \mid c/2 < (1+\delta_{k+1})^t \le c\}$. Thus, and since $c/2 < c_2 \le c$, we need to consider only $\mathcal{O}(\log_{(1+\delta_{k+1})} 2) = \mathcal{O}(1/\delta_{k+1})$ partitions. Similarly, for the third subset, it is sufficient to consider only $\mathcal{O}(1/\delta_{k+1})$ values of $c_1 \in \{(1+\delta_{k+1})^t \mid c/2 < (1+\delta_{k+1})^t \le c\}$. We conclude that the optimal partition of the budget c requires $\mathcal{O}(1/\delta_{k+1})$ time.

The merging procedure employs logarithmic sampling for an approximation parameter δ_k, i.e., it samples the delay function $D_{(v_i,v_j)}(c)$ at cost values $c \in \{1, 1+\delta_k, (1+\delta_k)^2, \ldots U\}$. Since the number of sampled values is $\mathcal{O}(\log U/\delta_k)$, the total complexity incurred is $\mathcal{O}(\log U/(\delta_k \delta_{k+1}))$. This procedure is referred to as Procedure MERGE and its formal specification is presented in Fig. 4. For clarity of presentation, Procedure MERGE identifies only the delay function $D_{(v_i,v_j)}(c)$ of $\mathcal{P}_{(v_i,v_j)}$. The corresponding partitions can be identified by a mild and straightforward modification of the procedure, with no penalty in terms of computational complexity.

Lemma 1: Given are layer-$(k+1)$ subpaths $\mathcal{P}_{(v_i,v_b)}$ and $\mathcal{P}_{(v_b,v_j)}$ with corresponding ε-approximate delay functions $D_{(v_i,v_b)}(c)$ and $D_{(v_b,v_j)}(c)$. Then, the execution of Procedure MERGE yields an $\hat{\varepsilon}$-approximate delay function $D_{(v_i,v_j)}(c)$ for the subpath $\mathcal{P}_{(v_i,v_j)}$, where $\hat{\varepsilon} = (1+\delta_k)(1+\varepsilon) - 1$.

Proof: See Appendix A. ■

Lemma 2: The computational complexity of Procedure MERGE is $\mathcal{O}(\frac{\log U}{\delta_k \delta_{k+1}})$.

Proof: See Appendix B. ■

5) Computing the Delay Function for $\mathcal{P}$: We proceed to present Procedure UNICAST, which computes the delay function of a unicast path $\mathcal{P}$. Procedure UNICAST receives, as input, a layer-k subpath $\mathcal{P}_{(v_i,v_j)}$ of $\mathcal{P}$ and an upper bound U. The procedure computes the values of delay function $D_{(v_i,v_j)}(c)$ of $\mathcal{P}_{(v_i,v_j)}$ for each $1 \le c \le U$ in the following recursive manner. If $\mathcal{P}_{(v_i,v_j)}$ is a layer-K path, i.e., $\mathcal{P}_{(v_i,v_j)}$ consists of a single link l, then the corresponding delay function is obtained by logarithmic sampling (see Section III.A.2) of the inverse of the link cost function $c_l(d)$. Otherwise, for layer-k paths, $1 \le k \le K-1$, we first recursively compute the delay functions for subpaths $\mathcal{P}_{(v_i,v_b)}$ and $\mathcal{P}_{(v_b,v_j)}$ of $\mathcal{P}_{(v_i,v_j)}$, where $b = (i+j)/2$; the delay function of the path $\mathcal{P}_{(v_i,v_j)}$ is then computed by merging the the delay functions for $\mathcal{P}_{(v_i,v_b)}$ and $\mathcal{P}_{(v_b,v_j)}$.

As mentioned, the computation of a delay function introduces some error at each layer k, which accumulates as we proceed to lower layers. The error at layer k depends on the approximation parameter δ_k used for this layer. The accumulated error at layer-k (i.e., accumulated along the layers $k, k+1, \ldots, K$) is denoted by $\varepsilon^{(k)}$.

The major consideration in choosing the approximation parameters δ_k for each layer $1 \le k \le K$ is to minimize the computational complexity of the scheme. In addition, the values δ_k must be chosen such that the total accumulated error for the path $\mathcal{P}$ is at most ε. From Lemma 2 it follows that the accumulated error at layer 0 is

$$\varepsilon^{(0)} = \prod_{k=0}^{K}(1+\delta_k) - 1 \le 2\sum_{k=0}^{K}\delta_k$$

where the last inequality follows from Claim 1 below. Note that layer k adds δ_k to the accumulated error.

Claim 1: Let $\delta_0, \ldots, \delta_k$ be positive numbers such that $\sum_{k=0}^{K}\delta_k \le 1$. Then it holds that

$$\prod_{k=0}^{K}(1+\delta_k) \le 1 + 2\sum_{k=0}^{K}\delta_k.$$

Proof: See Appendix C. ■

By Lemma 2, the computational complexity of invoking Procedure MERGE for a layer-k subpath is $\mathcal{O}(\frac{\log U}{\delta_k\delta_{k+1}})$. As there are 2^k subpaths at that layer, the time needed for processing all layer-k paths is $\mathcal{O}(\frac{1}{\delta_k\delta_{k+1}}2^k \log U)$. We also note that the time required for logarithmic sampling of a link cost function (lines 3 and 4 of Procedure UNICAST) is $\mathcal{O}(\frac{1}{\delta_K}\log U)$ per link or $\mathcal{O}(\frac{1}{\delta_K}2^K \log U)$ in total. We conclude that the total computational complexity of the procedure is $\mathcal{O}(\log U(\frac{1}{\delta_K}2^K + \sum_{k=0}^{K-1}\frac{1}{\delta_k\delta_{k+1}}2^k))$. Thus, in order to find a suitable assignment of approximation parameters $\{\delta_k\}$, we need to solve the following optimization problem:

$$\begin{array}{ll} & \min \frac{1}{\delta_K}2^K + \sum_{k=0}^{K-1}\frac{2^k}{\delta_k\delta_{k+1}} \\ \text{subject to :} & \sum_{k=0}^{K}\delta_k \le \frac{\varepsilon}{2}. \end{array} \quad (3)$$

The following assignment of $\{\delta_k\}$ minimizes the objective function in (3):

$$\delta_k = \frac{\varepsilon}{8\sqrt[3]{2^{K-k}}} \text{ for } k = 0, \ldots, K. \quad (4)$$

As we prove in Theorem 1 below, this assignment satisfies the constraint in (3) and yields a total running time of $\mathcal{O}(\frac{n}{\varepsilon^2}\log U)$.

The detailed description of Procedure UNICAST appears in Fig. 5. As was the case with Procedure MERGE, the partitions that correspond to the delay function $D_{(v_i,v_j)}(c)$ of $\mathcal{P}_{(v_i,v_j)}$ can be identified by a mild and straightforward modification of Procedure UNICAST, with no penalty in terms of computational complexity.

Theorem 1: Procedure UNICAST identifies, in $\mathcal{O}(\frac{1}{\varepsilon^2}n\log U)$ time, an ε-approximate delay function $D_{(v_0,v_n)}(c)$ for a path $\mathcal{P}$.

Proof: See Appendix D. ■

Procedure UNICAST $(\mathcal{P}_{(v_i,v_j)}, k, \{c_l\}_{l\in\mathcal{P}_{(v_i,v_j)}}, \varepsilon, U)$:

```
parameters
    $\mathcal{P}_{(v_i,v_j)}$- a subpath of $\mathcal{P}$ of layer $k$;
    $\{c_l\}_{l\in\mathcal{P}_{(v_i,v_j)}}$- the links' cost functions;
    $\varepsilon$ - approximation error;
    $U$- the upper bound on the cost of an optimal partition.

1  $\delta_k \leftarrow \frac{\varepsilon}{8\sqrt[3]{2^{K-k}}}$
2  if $k = K$ then
3      for each $c \in \{(1+\delta_k)^t \mid (1+\delta_k)^t \le U, t = 1,2,\ldots\}$ do
4          $D_{(v_i,v_j)}(c) \leftarrow \min\{d \mid c_{(v_i,v_j)}(d) \le c\}$
5      return $D_{(v_i,v_j)}(c)$
6  $\delta_{k+1} \leftarrow \delta_k \cdot \sqrt[3]{2}$
7  $b = \frac{i+j}{2}$
8  $D_{(v_i,v_b)}(c) \leftarrow$UNICAST($\mathcal{P}_{(v_i,v_b)}, k+1, \{c_l\}_{l\in\mathcal{P}_{(v_i,v_b)}}, \varepsilon, U$)
9  $D_{(v_b,v_j)}(c) \leftarrow$UNICAST($\mathcal{P}_{(v_b,v_j)}, k+1, \{c_l\}_{l\in\mathcal{P}_{(v_b,v_j)}}, \varepsilon, U$)
10 $D_{(v_i,v_j)}(c) \leftarrow$MERGE($D_{(v_i,v_b)}(c), D_{(v_b,v_j)}(c), \delta_k, \delta_{k+1}$,
                                                        $(1+\varepsilon)U$)
11 return $D_{(v_i,v_j)}(c)$
```

Fig. 5. Procedure UNICAST.

B. On-Demand Computation: Problem OPQ

In this section, we present an algorithm for computing a suitable QoS partition upon an incoming request. The algorithm comprises the following steps. First, we obtain sufficiently tight lower and upper bounds, L and U, on c^{opt}, where c^{opt} is the cost of an optimal partition. Then, we use these bounds in order to perform *linear scaling* on link cost functions. The purpose of linear scaling is to "scale down" all the costs, i.e., reduce all the costs by dividing them by some fixed parameter. The resulting graph has smaller costs, which reduces the overall running time. Next, we find a suitable partition by using Procedure UNICAST. The obtained solution is then rounded back to the original costs, i.e., prior to the linear scaling, incurring a small error.

1) Upper and Lower Bounds: Following the algorithmic technique presented in [7], we start with trivial bounds and proceed to iteratively improve them, until they become sufficiently tight.

Let $\{d_l\}_{l\in\mathcal{P}}$ be a partition that satisfies the delay constraint D. We observe that, for each link l in $\mathcal{P}$, it holds that $1 \le c_l(d_l) \le C^{\max}$, where $C^{\max}$ is the maximum cost of supporting a delay constraint by a link in $\mathcal{P}$. Thus, $L = n$ and $U = n \cdot C^{\max}$ are obvious lower and upper bounds on the cost of an optimal partition. Note that initially the ratio of the upper bound to the lower bound is $C^{\max}$, which can be a relatively high value.

In order to reduce the ratio U/L, we perform a binary search on the interval $[L, U]$. In general, a binary search is performed by selecting a trial value $c \in [L, U]$ and comparing it to c^{opt}. However, since Problem OPQ is $\mathcal{NP}$-hard, the exact comparison between c and c^{opt} is intractable, i.e., there is no efficient procedure that determines that either $c \le c^{\text{opt}}$ or $c > c^{\text{opt}}$. Accordingly, we use a test procedure that performs the following "loose" comparison between c and c^{opt}:

1) if the procedure returns a negative answer, then $c^{\text{opt}} \ge c$;
2) if the procedure returns a positive answer, then $c^{\text{opt}} \le c \cdot n$.

The test procedure is implemented as follows. First, for each link $l \in \mathcal{P}$ we compute the minimum value of the QoS requirement d_l that can be supported by allocating a budget c to l. Next, we check whether the resulting partition $\{d_l\}_{l\in\mathcal{P}}$ satisfies the delay constraint, i.e., $\sum_{l\in\mathcal{P}} d_l \le D$. Clearly, if $\{d_l\}_{l\in\mathcal{P}}$ satisfies the delay constraint, then the cost of the optimal solution is at most $c \cdot n$, and hence the procedure returns a positive answer. Otherwise, the cost of the optimal solution is at least c, hence the procedure returns a negative answer. Accordingly, at each iteration, we either update the lower bound $L = c$ (in case of a negative answer) or update the upper bound $U = c \cdot n$ (in case of a positive answer).

At each iteration, we choose the trial value c that results in the largest possible reduction of the ratio U/L. Specifically, by choosing $c = \sqrt{\frac{U\cdot L}{n}}$, we achieve the largest reduction, regardless of the outcome of the test procedure. Indeed, let β_i be the ratio of the upper bound to the lower bound, namely L and U, at the end of iteration i. If the test procedure returns a negative answer, then

$$\beta_i = \frac{U}{c} = \sqrt{\frac{nU}{L}} = \sqrt{n\beta_{i-1}}.$$

Otherwise, the value of β_i is

$$\beta_i = \frac{nc}{L} = \sqrt{\frac{nU}{L}} = \sqrt{n\beta_{i-1}}.$$

In both cases, we have $\beta_i = \sqrt{n\beta_{i-1}}$.

If at the beginning of the execution of the algorithm, the ratio of the upper and to the lower bound is β, and then after the first iteration we have $\beta_1 = n^{\frac{1}{2}}\beta^{\frac{1}{2}}$. The second iteration yields $\beta_2 = n^{\frac{1}{2}}\beta_1^{\frac{1}{2}} = n^{\frac{3}{4}}\beta^{\frac{1}{2}}$. It can be easily verified that, for each iteration i, the value of β_i is bounded by

$$\beta_i \le n\beta^{\frac{1}{2^i}}.$$

Note that, at iteration $i = \lceil \log\log\beta \rceil$, we have $\beta_i \le 2\cdot n$. We conclude that just $\mathcal{O}(\log\log\beta) = \mathcal{O}(\log\log C^{\max})$ iterations are necessary in order to achieve $U/L \le 2 \cdot n$. Since each iteration requires $\mathcal{O}(n)$ time, the computational complexity of finding lower and upper bounds L and U, for which $U/L \le 2\cdot n$, is $\mathcal{O}(n \log\log C^{\max})$.

```
Algorithm OPQ ($\mathcal{P}, \{c_l\}_{l\in\mathcal{P}}, \varepsilon, D$):
    parameters
        $\mathcal{P} = \{v_0, ..., v_n\}$- a QoS path;
        $\{c_l\}_{l\in\mathcal{P}}$- the links' cost functions;
        $\varepsilon$ - approximation error;
        $D$- delay constraint.

1   $L \leftarrow \sum_{l\in\mathcal{P}} c_l(D)$
2   $U \leftarrow \sum_{l\in\mathcal{P}} c_l(d_l^{\min})$, where $d_l^{\min} = \min\{d \mid c_l(d) \ne \infty\}$
3   while $\frac{U}{L} > 2\cdot n$ do
4       $c \leftarrow \sqrt{\frac{L\cdot U}{n}}$
5       for each $l \in \mathcal{P}$ do
6           $d_l = \min\{d \mid c_l(d) \le c\}$
7       if $\sum_{l\in\mathcal{P}} d_l \le D$ then
8           $U \leftarrow c\cdot n$
9       else
10          $L \leftarrow c$
11  $D_{(v_0,v_n)}(c) \leftarrow$UNICAST$\left(\mathcal{P}, 0, \left\{\left\lfloor \frac{c_l(d_l)\cdot n}{(\varepsilon/2)\cdot L}\right\rfloor\right\}, \varepsilon/2, \frac{4n^2}{\varepsilon}\right)$
12  $\hat{c} \leftarrow \min\{c \mid D_{(v_0,v_n)}(c) \le D\}$
13  return partition that corresponds to $\hat{c}$
```

Fig. 6. Algorithm OPQ.

2) Algorithm: Having computed suitable bounds U and L, i.e., bounds for which $U/L \le 2 \cdot n$, 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:

$$c_l^*(d_l) = \left\lfloor \frac{2n \cdot c_l(d_l)}{\varepsilon L} \right\rfloor. \tag{5}$$

With modified link costs, the new cost c^* of a partition with original cost c is bounded by

$$\frac{2cn}{\varepsilon L} - n \le c^* \le \frac{2cn}{\varepsilon L}. \tag{6}$$

Thus, and since $U \le 2n \cdot L$, the upper bound on the solution with respect to the new link cost functions is $U^* = \frac{4n^2}{\varepsilon}$. Finally, the problem is solved by applying Procedure UNICAST to a path with the scaled cost functions $c_l^*(d_l)$. The procedure is invoked with the upper bound U^* and the approximation error $\varepsilon/2$. The procedure returns an $\frac{\varepsilon}{2}$-approximate solution with respect to the new link costs. Theorem 2 below implies that 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 Fig. 6, summarizes the above discussion.

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

Proof: See Appendix E. ■

C. Precomputation Scheme: Problem POPQ

Precomputation is performed by means of a two-phase procedure, referred to as a *precomputation scheme*. The purpose of the first phase is to compute a delay function for the path $\mathcal{P}$, which summarizes a set of suitable partitions, for each delay

```
Algorithm POPQ ($\mathcal{P}, \{c_l\}_{l\in\mathcal{P}}, \varepsilon$):
    parameters
        $\mathcal{P} = \{v_0, ..., v_n\}$- a QoS path;
        $\{c_l\}_{l\in\mathcal{P}}$- the links' cost functions;
        $\varepsilon$ - approximation error;

1   $D'_{(v_0,v_n)}(c) \leftarrow$ UNICAST($\mathcal{P}, 0, \{c_l\}_{l\in\mathcal{P}}, \varepsilon/3, n \cdot C^{\max}$)
2   for each $c \in \{(1+\varepsilon/3)^t \le n \cdot C^{\max} \mid t = 1, 2, \dots\}$ do
3       $D_{(v_0,v_n)}(c) \leftarrow D'_{(v_0,v_n)}(c)$
4   return $D_{(v_0,v_n)}(c)$
```

Fig. 7. Algorithm POPQ.

constraint. The second phase merely selects one of the solutions precomputed in the first phase.

1) First Phase: The first phase is implemented as follows. We begin by invoking Procedure UNICAST with approximation error $\varepsilon/3$, which computes an $\varepsilon/3$-approximate delay function $D'_{(v_0,v_n)}(c)$ and the corresponding partitions. Then, we use $D'_{(v_0,v_n)}(c)$ in order to compute a delay function $D_{(v_0,v_n)}(c)$ whose storage requirements are significantly smaller.

More specifically, the delay function $D'_{(v_0,v_n)}(c)$, obtained through Procedure UNICAST, is a piecewise-constant function whose segments correspond to costs $c \in \{1, (1+\delta_0), \dots, nC^{\max}\}$, where $\delta_0 = \frac{\varepsilon}{24\sqrt[3]{n}}$ (according to (4)). Thus, we need $\mathcal{O}(\frac{\sqrt[3]{n}}{\varepsilon}\log(nC^{\max}))$ space in order to store $D'_{(v_0,v_n)}(c)$. The storage requirement can be significantly reduced by logarithmic sampling. Specifically, we compute new delay function $D_{(v_0,v_n)}(c)$ out of $D'_{(v_0,v_n)}(c)$ by logarithmic sampling at costs $\{1, (1+\varepsilon/3), (1+\varepsilon/3)^2, \dots, nC^{\max}\}$. By Lemma 3 below, $D_{(v_0,v_n)}(c)$ is an ε-approximate delay function for $\mathcal{P}$. The detailed description of the first part of the precomputation scheme, implemented by Algorithm POPQ, appears in Fig. 7.

Lemma 3: Algorithm POPQ computes, in $\mathcal{O}(\frac{1}{\varepsilon^2}n\log(nC^{\max}))$ time, an ε-approximate delay function $D_{(v_0,v_n)}(c)$ for $\mathcal{P}$.

Proof: See Appendix F. ■

2) Second Phase: 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 of a suitable partition, $c = \min\{c' = (1+\varepsilon/3)^t \mid D_{(v_0,v_n)}(c') \le D\}$, and return the corresponding partition. Since the total number of precomputed partitions is $\mathcal{O}(\frac{1}{\varepsilon}\log(nC^{\max}))$, the computational complexity of this procedure is $\mathcal{O}(\log\log(C^{\max}) + \log(1/\varepsilon) + n)$. The term n in the complexity expression is due to the need to describe the partition.

D. Discussion

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

We begin with the on-demand setting. In [7] and [4], 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 $\mathcal{O}(n\log\log C^{\max} + \frac{n^2\log(n/\varepsilon)}{\varepsilon^2})$ and $\mathcal{O}(\min(D, \frac{\log C^{\max}}{\varepsilon}, \frac{n}{\varepsilon}D)\frac{1}{\varepsilon}n^2\log\log C^{\max})$, respectively. The dominant terms of these expression are $\mathcal{O}(\frac{n^2\log(n/\varepsilon)}{\varepsilon^2})$ and $\mathcal{O}(\frac{n^2\log C^{\max}}{\varepsilon^2})$, respectively, while the dominant term in our solution is $\mathcal{O}(\frac{n\log(n/\varepsilon)}{\varepsilon^2})$. We thus conclude that the computational complexity of our algorithm is significantly $(\Omega(n))$ less dependent on the topology size than that of [7] and [4], 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, i.e., step functions whose range is a discrete set of values. Such functions have been the focus of [8], and an $\mathcal{O}(rn^3\log\frac{r}{\varepsilon})$ algorithm was presented there, where $r = \sum_{l\in\mathcal{P}} r_l$ and r_l is the number of different delay values supported by link l. We conclude that, even if $r = \mathcal{O}(n)$ (i.e., each link supports a fixed number of delays), we achieve a major $(\Omega(n^3))$ reduction in terms of dependency on the topology size. It should be noted that the complexity of our algorithm is more dependent (by a factor of $\frac{1}{\varepsilon^2}$) on the value of the approximation error ε, so for very small values of ε, the algorithm presented in [8] might exhibit better performance.

We described a precomputation scheme for Problem OPQ that provides ε-optimal solutions within a computational complexity of $\mathcal{O}(\frac{1}{\varepsilon^2}n \cdot \log(n \cdot C^{\max}))$ for the first phase and $\mathcal{O}(\log\log(C^{\max}) + \log\frac{1}{\varepsilon} + n)$ for the second phase. Compared with an on-demand scheme, the precomputation scheme significantly reduces 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 $(\mathcal{O}(n))$, 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., [4], [7]) 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 higher than that of our solution.

IV. QoS Partition for Multicast

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 *directed tree* is a subgraph $\mathcal{T}$ of $G(V, E)$ having a unique node s such that every node is reached from s by a unique path; node s is referred to as the *source*. A *multicast connection* uses a tree $\mathcal{T}$ to interconnect the source s and the members of a multicast group $M = \{t_1, t_2, \dots\}$. A path between source s and a terminal t_i on links that belong to the tree $\mathcal{T}$ is denoted by $\mathcal{P}_i$. Given a multicast tree $\mathcal{T}$, our goal is to (efficiently) allocate the delay on each link $l \in \mathcal{T}$ such that the end-to-end delay is satisfied for each member t_i of the multicast group. A QoS partition on a multicast tree $\mathcal{T}$ is a set of link delay requirements $\{d_l\}_{l\in\mathcal{T}}$, which satisfies, for each $t_i \in M$, the end-to-end delay requirement D, i.e., $\sum_{l\in\mathcal{P}_i} d_l \le D$ for each $t_i \in M$. Each link is associated with a cost function $c_l(d)$, which specifies the cost of supporting a delay requirement d. The cost of a QoS partition

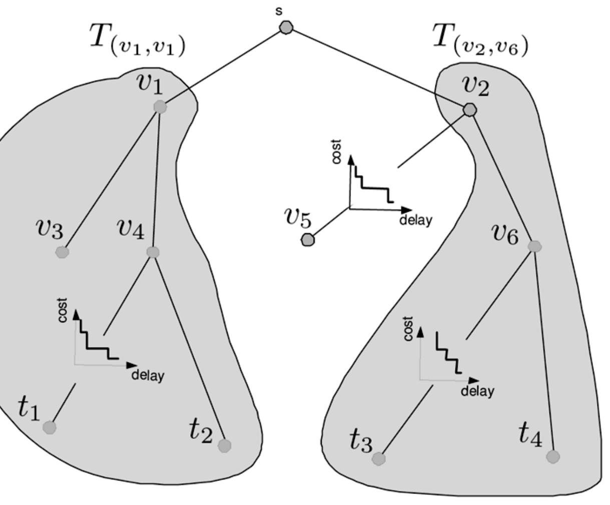

Fig. 8. Example of a multicast tree, $n = 11$ and $H = 4$.

$\{d_l\}_{l\in\mathcal{T}}$ is the sum of the local costs, i.e., $\sum_{l\in\mathcal{T}} c_l(d_l)$. We assume that all parameters (cost and delays) are (positive) integers.

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

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

We define also the related precomputation problem.

Problem PMOPQ: (Precomputation of MOPQ): Given a tree $\mathcal{T}$, find, for each delay requirement D, a QoS partition $\{d_l\}_{l\in\mathcal{T}}$ such that $\sum_{l\in\mathcal{P}_i} d_l \le D$ for each $t_i \in M$ and $\sum_{l\in\mathcal{T}} c_l(d_l)$ 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 is denoted by m_i. The subtree originating from the node $v_i \in \mathcal{T}$ is denoted by $\mathcal{T}_{(v_i,v_i)}$. A *branch* $\mathcal{T}_{(v_i,v_j)}$ of the subtree $\mathcal{T}_{(v_i,v_i)}$ is a subtree originating from v_i, which includes the link (v_i,v_j) outgoing from i and all descendants of v_j. For example, Fig. 8 shows a multicast tree $\mathcal{T}$, a subtree $\mathcal{T}_{(v_1,v_1)}$ and a branch $\mathcal{T}_{(v_2,v_6)}$.

We employ the following "divide-and-conquer scheme". A multicast tree is recursively split into a number of disjoint subtrees. We compute the set of delay guarantees supported by each subtree at different costs. These delay guarantees and the corresponding partitions are summarized by means of *delay functions*, defined below. We then obtain a solution to the original problem, i.e., a partition of delay constraint D on tree $\mathcal{T}$, by recursively combining the delay functions obtained for the subtrees of $\mathcal{T}$.

More specifically, consider a multicast tree $\mathcal{T}$, which is referred to as a *layer*-0 tree. We split $\mathcal{T}$ into a number of *layer*-1 subtrees $\{\mathcal{T}_{(v_i,v_i)}\}$, for each child node v_i of s. Then, for each value $k, k = 1,2,\ldots,H$, each layer-k subtree $\mathcal{T}_{(v_j,v_j)}$ is split into a number of layer-$(k+1)$ subtrees, for each child node of v_j. Each layer-H subtree includes just a single node. For example, in the tree $\mathcal{T}$ depicted in Fig. 8, subtrees $\mathcal{T}_{(v_1,v_1)}$ and $\mathcal{T}_{(v_2,v_2)}$ are layer-1 subtrees, while $\mathcal{T}_{(v_3,v_3)}$ and $\mathcal{T}_{(v_4,v_4)}$ are layer-2 subtrees. We denote by n_k the number of subtrees of layer-k. Clearly, $\sum_{k=1}^{H} n_k = n$.

We introduce the following *subtree delay functions*, which summarize the delay guarantees offered by a subtree at different costs.

Definition 3: The *optimal delay function* $D^{\text{opt}}_{(v_i,v_i)}(c)$ of the subtree $\mathcal{T}_{(v_i,v_i)}$ of $\mathcal{T}$ is defined as the minimum delay requirement supported by $\mathcal{T}_{(v_i,v_i)}$ at cost c, i.e.,

$$D^{\text{opt}}_{(v_i,v_i)}(c) = \min\Bigg\{ D \,|\, \exists\{d_l\}_{l\in\mathcal{T}_{(v_i,v_i)}} \text{ such that } \max_{t_k\in\mathcal{T}_{(v_i,v_i)}} \sum_{l\in\mathcal{P}_{(i,k)}} d_l \le D \text{ and } \sum_{l\in\mathcal{T}_{(v_i,v_i)}} c_l(d_l) \le c \Bigg\} \quad (7)$$

where $\mathcal{P}_{(i,k)}$ is a path between node v_i and terminal t_k on links the belong to tree $\mathcal{T}$.

Optimal delay functions for branches $\mathcal{T}_{(v_i,v_j)}$ of $\mathcal{T}_{(v_i,v_i)}$ are defined similarly.

In addition we define, for each link $(v_i,v_j) \in \mathcal{T}$, the optimal delay function $\tilde{D}^{\text{opt}}_{(v_i,v_j)}(c)$ in a way that resembles the optimal delay function of a subpath (see Definition 1, Section III-A1). Specifically, the optimal delay function $\tilde{D}^{\text{opt}}_{(v_i,v_j)}(c)$ of link (v_i,v_j) is defined as the minimum delay requirement d supported by link (v_i,v_j) at cost c, i.e., $\tilde{D}^{\text{opt}}_{(v_i,v_j)}(c) = \min\{d \,|\, c_{(v_i,v_j)}(d) \le c\}$.

Definition 4: A *ε-approximate delay function* $D_{(v_i,v_i)}(c)$ of a subtree $\mathcal{T}_{(v_i,v_i)}$ of $\mathcal{T}$ is a function that satisfies, for each $c \ge 0, D_{(v_i,v_i)}(c(1+\varepsilon)) \le D^{\text{opt}}_{(v_i,v_i)}(c)$.

We define ε-approximate delay functions for branches and links in a similar manner. When no ambiguity exists, ε-approximate delay functions will be referred to as just delay functions. Delay functions are constructed by using the logarithmic sampling approach.

A. Computation of Delay Functions

In this section, we present an overview of Procedure MULTICAST which identifies the delay functions $D_{(v_i,v_i)}(c)$ and the corresponding partitions for each subtree of each layer. The delay 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 each layer-H subtree $\mathcal{T}_{(v_i,v_i)}$ includes a single terminal node v_i. For each terminal node v_i the delay function $D_{(v_i,v_i)}(c)$ of subtree $\mathcal{T}_{(v_i,v_i)}$ is set to 0 for all c.

More specifically, we compute the delay function $D_{(v_i,v_i)}(c)$ of subtree $\mathcal{T}_{(v_i,v_i)}$ by performing the following steps.

Step 1) If v_i is a terminal, then $D_{(v_i,v_i)}(c)$ is set to 0 for all c. Otherwise, for each child node v_j of v_i:
  a) Recursively compute the delay function $D_{(v_j,v_j)}(c)$ of layer-$(k+1)$ subtree $\mathcal{T}_{(v_j,v_j)}$;
  b) Compute the delay function $\tilde{D}_{(v_i,v_j)}(c)$ of the link (v_i,v_j) by performing logarithmic sampling on the link cost function $c_{(v_i,v_j)}(d)$ of (v_i,v_j);
  c) Compute the delay function $D_{(v_i,v_j)}(c)$ of the branch $\mathcal{T}_{(v_i,v_j)}$ by merging the delay functions $\tilde{D}_{(v_i,v_j)}(c)$ and $D_{(v_j,v_j)}(c)$;

Step 2) Compute the delay function $D_{(v_i,v_i)}(c)$ of the subtree $\mathcal{T}_{(v_i,v_i)}$ by merging the delay functions of all branches $\{\mathcal{T}_{(v_i,v_j)}\}$ of $\mathcal{T}_{(v_i,v_i)}$.

As is the case for unicast, the critical part is to choose, for each layer k, the approximation parameter δ_k used for computing delay functions. The assignment of δ_k is discussed in detail in Section IV.A.2.

1) Merging Procedures: As discussed above, in order to compute the delay function $D_{(s,s)}(c)$ we need to define two merging procedures, which we proceed to describe in some more detail.

The first procedure receives, as an input, the delay functions $\tilde{D}_{(v_i,v_j)}(c)$ and $D_{(v_j,v_j)}(c)$ of link (v_i,v_j) and subtree $\mathcal{T}_{(v_j,v_j)}$, respectively, and an upper bound U. The function computes the values of the delay function $D_{(v_i,v_j)}(c)$ of the branch $\mathcal{T}_{(v_i,v_j)}$ for each $1 \le c \le U$. The goal of this procedure is to compute for each $c, 1 \le c \le U$, the partition (c_1,c_2) of a budget c between the link (v_i,v_j) and subtree $\mathcal{T}_{(v_j,v_j)}$, which minimizes the delay supported by the branch $\mathcal{T}_{(v_i,v_j)}$ under budget constraint c. The procedure is similar to the merger of the delay functions of two subpaths, as discussed in Section III-A4. Accordingly, we use Procedure MERGE that appears on Fig. 4.

The purpose of the second procedure, referred to as MIN–MAX–MERGE, is to calculate the delay function $D_{(v_i,v_i)}(c)$ of the subtree $\mathcal{T}_{(v_i,v_i)}$ out of the delay functions $\{D_{(v_i,v_j)}(c)\}$ of its branches. In order to compute $D_{(v_i,v_i)}(c)$, we find, for each cost value $c, 1 \le c \le U$, the minimum delay that can be supported by the subtree $\mathcal{T}_{(v_i,v_i)}$ subject to budget c. For this purpose, we need to find the local budget c_j for each branch $\mathcal{T}_{(v_i,v_j)}$ in such a way that the maximum delay between v_i and a terminal $t_i \in \mathcal{T}_{(v_i,v_i)}$ is minimized, i.e.,

$$D_{(v_i,v_i)}(c) = \min\left\{ D \mid \exists \{c_j\}_{(v_i,v_j)\in\mathcal{T}} \text{ such that } \max_{(v_i,v_j)\in\mathcal{T}} D_{(v_i,v_j)}(c_j) \le D \text{ and } \sum_{(v_i,v_j)\in\mathcal{T}} c_j \le c \right\}. \quad (8)$$

Note that the delay function $D_{(v_i,v_j)}(c)$ of each branch $\mathcal{T}_{(v_i,v_j)}$ is piecewise-constant. Hence, the function $D_{(v_i,v_i)}(c)$ of the subtree $\mathcal{T}_{(v_i,v_i)}$ is also piecewise-constant and can be computed by identifying its segments. We begin with segments that correspond to lower costs and then proceed with segments that correspond to higher costs. Since the cost of supporting a delay requirement by each branch $\mathcal{T}_{(v_i,v_j)}$ is at least 1, then the minimum cost for supporting a delay requirement by subtree $\mathcal{T}_{(v_i,v_i)}$ is m_i, hence we set

$$D_{(v_i,v_i)}(m_i) = \max_{(v_i,v_j)\in\mathcal{T}} \left\{ D_{(v_i,v_j)}(1) \right\}.$$

Thus, the first segment corresponds to cost m_i and delay $D_{(v_i,v_i)}(m_i)$. Suppose that we have identified the segment of $D_{(v_i,v_i)}(c)$ that corresponds to delay constraint $\hat{d}$, cost $\hat{c}$ of supporting $\hat{d}$ and the corresponding partition $\{\hat{c}_j\}_{(v_i,v_j)\in\mathcal{T}}$, i.e., $\hat{d} = D_{(v_i,v_i)}(\hat{c})$ and $\sum_{(v_i,v_j)\in\mathcal{T}} \hat{c}_j = \hat{c}$. We show how to identify the next segment of $D_{(v_i,v_i)}(c)$ that corresponds to delay $\hat{d}'$ and cost $\hat{c}' = \sum_{(v_i,v_j)\in\mathcal{T}} \hat{c}'_j$. Note that $\hat{c}'$ is the minimum cost that must be paid in order to support a delay constraint lower than $\hat{d}$, i.e., $\hat{c}' = \min\{c \mid D_{(v_i,v_i)}(c) < \hat{d}\}$ and $\hat{d}' = D_{(v_i,v_i)}(\hat{c}')$.

```
Procedure MIN-MAX-MERGE ($\mathcal{T}_{(v_i,v_i)}, U$):
    parameters
        $\mathcal{T}_{(v_i,v_i)}$ the subtree of $\mathcal{T}$
        $U$- the upper bound on the cost of a partition.

1  for each $(v_i,v_j) \in \mathcal{T}$ do
2      $c_j \leftarrow 1$
3      $d_j \leftarrow D_{(v_i,v_j)}(c_j)$
4  $c \leftarrow \sum_{(v_i,v_j)\in\mathcal{T}} c_j$
5  $d \leftarrow \max_{(v_i,v_j)\in\mathcal{T}} d_j$
6  while $c \le U$ do
7      $D_{(v_i,v_i)}(c) \leftarrow d$
8      $S \leftarrow \{j \mid D_{(v_i,v_j)}(c_j) = d\}$
9      for each $j \in S$ do
10         $c_j \leftarrow \min\{c \mid D_{(v_i,v_j)}(c) < d_j\}$
11         $d_j \leftarrow D_{(v_i,v_j)}(c_j)$
12     $c \leftarrow \sum_{(v_i,v_j)\in\mathcal{T}} c_j$
13     $d \leftarrow \max_{(v_i,v_j)\in\mathcal{T}} d_j$
```

Fig. 9. Procedure MIN–MAX–MERGE.

We observe, by (8), that there exists a link $(v_i,v_j) \in \mathcal{T}$ for which it holds that $D_{(v_i,v_i)}(\hat{c}) = D_{(v_i,v_j)}(\hat{c}_j)$. We denote by $S = \{j \mid D_{(v_i,v_j)}(\hat{c}_j) = D_{(v_i,v_i)}(\hat{c})\}$. Since $\hat{d}' < \hat{d}$, for each $j \in S$, the delay supported by branch $\mathcal{T}_{(v_i,v_j)}$ at cost $\hat{c}'_j$ must be lower than $\hat{d}$. Thus, we set $\hat{c}'_j$ to be the minimum cost of supporting a delay lower than $\hat{d}$ by branch $\mathcal{T}_{(v_i,v_j)}$. For each $j \notin S$ we set $\hat{c}'_j = \hat{c}_j$. As we prove in Lemma 4 below, the next segment of $D_{(v_i,v_j)}(c)$ corresponds to cost $\hat{c}' = \sum_{(v_i,v_j)\in\mathcal{T}} \hat{c}'_j$ and delay $\hat{d}' = \max_{(v_i,v_j)\in\mathcal{T}}\{D_{(v_i,v_j)}(\hat{c}'_j)\}$. The formal description of Procedure MIN–MAX–MERGE appears in Fig. 9.

Lemma 4: Let $\mathcal{T}_{(v_i,v_i)}$ be a layer-k subtree. Suppose that each branch $\mathcal{T}_{(v_i,v_j)}$ of $\mathcal{T}_{(v_i,v_i)}$ has a corresponding ε-approximate delay function $D_{(v_j,v_j)}(c)$ that includes $\mathcal{O}(\frac{\log U}{\delta_k})$ segments. Then, Procedure MIN–MAX–MERGE computes, in $\mathcal{O}(\frac{1}{\delta_k} m_i \log m_i \log U)$ time, a ε-approximate delay function $D_{(v_i,v_i)}(c)$ for the subtree $\mathcal{T}_{(v_i,v_i)}$.

Proof: See Appendix G. ■

2) Procedure MULTICAST: We proceed to discuss Procedure MULTICAST in more detail. The formal description of the procedure appears on Fig. 10. The procedure receives as input a layer-k subtree $\mathcal{T}_{(v_i,v_i)}$, a set of link cost functions $\{c_l\}_{l\in\mathcal{T}_{(v_i,v_i)}}$, an approximation error ε, and an upper bound U.

In line 1, we set the value of the approximation parameter δ_k for layer k. We explain our choice of δ_k later in this section. Lines 2–4 handle the case in which node v_i is a terminal node. In line 6, we compute the delay function $\hat{D}_{(v_j,v_j)}(c)$ for each layer-$(k+1)$ subtree $\mathcal{T}_{(v_j,v_j)}$ by recursively invoking Procedure MULTICAST for $\mathcal{T}_{(v_j,v_j)}, k+1, \{c_e\}, \varepsilon$, and U. In lines 7 and 8, we perform an additional procedure in order to reduce the number of segments in $\hat{D}_{(v_i,v_i)}(c)$. Specifically, we perform logarithmic sampling at costs $1, 1+\delta_k, (1+\delta_k)^2, \ldots$, and denote the resulting function by $D_{(v_j,v_j)}(c)$. In lines 9 and 10, we compute the delay function $\tilde{D}_{(v_i,v_j)}(c)$ of link (v_i,v_j) by performing logarithmic sampling at costs $1, 1+\delta_k, (1+\delta_k)^2, \ldots$. Lines 7–10 ensure that the number of segments in the functions $D_{(v_j,v_j)}(c)$ and $\tilde{D}_{(v_i,v_j)}(c)$ is bounded by $\mathcal{O}(\frac{1}{\delta_k}\log U)$. In line 11, we compute the delay function of the branch $\mathcal{T}_{(v_i,v_j)}$

Procedure MULTICAST ($\mathcal{T}_{(v_i,v_i)}, k, \{c_l\}_{l\in\mathcal{T}_{(v_i,v_i)}}, \varepsilon, U$):

```
parameters
    $\mathcal{T}_{(v_i,v_i)}$- subtree of layer $k$
    $\{c_l\}_{l\in\mathcal{T}_{(v_i,v_i)}}$- the links' cost functions;
    $\varepsilon$ - approximation error;
    $U$- the upper bound on the cost of an optimal partition.

1  $\delta_k \leftarrow \frac{\varepsilon\sqrt[3]{n_{k+1}}}{6\sum_{k=1}^{H}\sqrt[3]{n_k}}$
2  if $v_i$ is a terminal then
3      $D_{(v_i,v_i)}(c) \leftarrow 0$ for all $c$
4      return $D_{(v_i,v_i)}(c)$
5  for each $(v_i,v_j)\in\mathcal{T}$ do
6      $\hat{D}_{(v_j,v_j)}(c) \leftarrow$MULTICAST($\mathcal{T}_{(v_j,v_j)}, k+1, \{c_e\}, \varepsilon, U$)
7      for each $c\in\{(1+\delta_k)^t \mid (1+\delta_k)^t\le U, t=1,2,\dots\}$ do
8          $D_{(v_j,v_j)}(c) \leftarrow \hat{D}_{(v_j,v_j)}(c)$
9      for each $c\in\{(1+\delta_k)^t \mid (1+\delta_k)^t\le U, t=1,2,\dots\}$ do
10         $\tilde{D}_{(v_i,v_j)}(c) \leftarrow \min\{d \mid c_{(v_i,v_j)}(d)\le c\}$
11     $D_{(v_i,v_j)}(c) \leftarrow$MERGE($\tilde{D}_{(v_i,v_j)}(c), D_{(v_j,v_j)}(c), \delta_k,$
                                                    $\delta_k, (1+\varepsilon)U$)
12 $D_{(v_i,v_i)}(c) \leftarrow$MIN-MAX-MERGE($\mathcal{T}_{(v_i,v_i)}, (1+\varepsilon)U$)
13 return $D_{(v_i,v_i)}(c)$
```

Fig. 10. Procedure MULTICAST.

by invoking Procedure MERGE. Having computed the delay function for each branch $\mathcal{T}_{(v_i,v_j)}$ of $\mathcal{T}_{(v_i,v_i)}$, we identify the delay function $D_{(v_i,v_i)}(c)$ of $\mathcal{T}_{(v_i,v_i)}$ by invoking Procedure MIN–MAX–MERGE (line 12).

We denote by $\varepsilon^{(k)}$ the accumulated error at layer-k, that is, for layer-k nodes, Procedure MULTICAST computes $\varepsilon^{(k)}$-approximate delay functions. Note that, for each terminal v_i, the procedure computes the optimal delay function, hence $\varepsilon^{(H)}=0$. Suppose that the accumulated error at layer-$(k+1)$ is $\varepsilon^{(k+1)}$, i.e., for each layer-$(k+1)$ node v_j Procedure MULTICAST returns $\varepsilon^{(k+1)}$-approximate delay functions $\hat{D}_{(v_j,v_j)}(c)$. In lines 7 and 8, we perform logarithmic sampling with the parameter δ_k, hence the resulting function $D_{(v_j,v_j)}(c)$ is ε'-approximate, where $\varepsilon'=(1+\delta_k)(1+\varepsilon^{(k+1)})-1$. In lines 9 and 10, we compute an δ_k-approximate delay function $\tilde{D}_{(v_i,v_j)}(c)$ for link (v_i,v_j). Then, we apply Procedure MERGE for the functions $D_{(v_j,v_j)}(c)$ and $\tilde{D}_{(v_i,v_j)}(c)$. By Lemma 1, Procedure MERGE computes an $\hat{\varepsilon}$-approximate delay function $D_{(v_i,v_j)}(c)$ for the branch $\mathcal{T}_{(v_i,v_j)}$, where $\hat{\varepsilon}=(1+\delta_k)^2(1+\varepsilon^{(k+1)})-1$. Thus, by Lemma 4, Procedure MULTICAST returns a $\varepsilon^{(k)}$-approximate delay function for the subtree $\mathcal{T}_{(v_i,v_i)}$, where $\varepsilon^{(k)}=(1+\delta_k)^2(1+\varepsilon^{(k+1)})-1$.

We conclude that the accumulated error at layer-0 is

$$\begin{aligned}\varepsilon^{(0)} &= \prod_{k=0}^{H-1}(1+\delta_k)^2-1\\ &= \prod_{k=0}^{H-1}\left(1+2\delta_k+\delta_k^2\right)-1\\ &\le \prod_{k=0}^{H-1}(1+3\delta_k)-1\\ &\le 6\sum_{k=0}^{H-1}\delta_k\end{aligned}$$

where the last inequality follows from Claim 1.

The time needed for processing all subtrees is dominated by the time required for the execution of Procedure MERGE for all subtrees of all layers. Since Procedure MERGE is applied for functions with $\mathcal{O}(\frac{1}{\delta_k}\log U)$ segments, its computational complexity is $\mathcal{O}(\frac{\log U}{\delta_k^2})$ (by Lemma 2). As the number of branches of layer-k subtrees is n_{k+1}, the total running time required for invoking Procedure MERGE for layer k subtrees is $\mathcal{O}(\frac{n_{k+1}\log U}{\delta_k^2})$. The total computational complexity of the algorithm is $\mathcal{O}(\sum_{k=0}^{H-1}\frac{n_{k+1}\log U}{\delta_k^2})$. Thus, in order to find a suitable assignment of approximation parameters $\{\delta_k\}$, we need to solve the following optimization problem:

$$\begin{aligned}&\min\sum_{k=0}^{H-1}\frac{n_{k+1}}{\delta_k^2}.\\ \text{subject to :}\quad &6\sum_{k=0}^{H-1}\delta_k\le\varepsilon.\end{aligned}\tag{9}$$

The following assignment minimizes the objective function in (9):

$$\delta_k=\frac{\varepsilon\sqrt[3]{n_{k+1}}}{6\sum_{k=1}^{H}\sqrt[3]{n_k}}\tag{10}$$

for $k=0,\dots,H-1$.

As we prove in Theorem 3 below, with this assignment of $\{\delta_k\}$, the total running time required for all invocations of Procedure MERGE is $\mathcal{O}(\frac{nH^2}{\varepsilon^2}\log U)$.

Note 1: If $\mathcal{T}$ is a balanced tree, then we assign $\delta_k=\frac{\varepsilon}{24}\sqrt[3]{\frac{2^k}{n}}$ for $k=1,\dots,H-1$.

Theorem 3: Procedure MULTICAST identifies, in $\mathcal{O}(\frac{1}{\varepsilon^2}nH^2\log U)$ time, an ε-approximate delay function $D_{(s,s)}(c)$ for a tree $\mathcal{T}$.

Proof: See Appendix H. ■

Note 2: If the tree $\mathcal{T}$ is balanced, using the assignment of δ_k as specified in Note 1 yields a computational complexity of $\mathcal{O}(\frac{1}{\varepsilon^2}n\log U)$.

B. *Solving Problem MOPQ*

The algorithm for Problem MOPQ follows the same lines as Algorithm OPQ for Problem OPQ, described in Section III-B, with the following modifications.

1) The procedure returns a partition $\{d_l\}_{l\in\mathcal{T}}$. A partition $\{d_l\}_{l\in\mathcal{T}}$ satisfies the delay constraint D if for each terminal t_i it holds that $\sum_{l\in\mathcal{P}_i}d_l\le D$.
2) The initial lower and upper bounds L and U are computed as follows: $L=\sum_{l\in\mathcal{T}}c_l(D)$ and $U=\sum_{l\in\mathcal{T}}c_l(d_l^{\min})$, where $d_l^{\min}=\min\{d\,|\,c_l(d)\ne\infty\}$.
3) We use Procedure MULTICAST instead of Procedure UNICAST.

We summarize our results in the following theorem.

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

Proof: The proof is similar to that of Theorem 2. ■

Note 3: If the tree $\mathcal{T}$ is balanced, an ε-approximate solution can be identified in $\mathcal{O}(\frac{1}{\varepsilon^2}n\log\frac{n}{\varepsilon}+n\log\log C^{\max})$ time.

The precomputation scheme for Problem MOPQ is also similar to that of Problem OPQ (Section III-C). The only difference between the two schemes is that we use Procedure MULTICAST instead of Procedure UNICAST. As a result, the computational complexity is $\mathcal{O}(\frac{1}{\varepsilon^2} n \cdot H^2 \log(n \cdot C^{\max}))$ for the first phase and $\mathcal{O}(\text{loglog}(C^{\max}) + \log \frac{1}{\varepsilon} + n)$ for the second phase. For the special case of balanced trees, the computational complexity is $\mathcal{O}(\frac{1}{\varepsilon^2} n \cdot \log(n \cdot C^{\max}))$.

C. 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 $\mathcal{O}(n \log D \log\log \beta + n^2(\log D + n) \log\log H + \frac{n^2}{\varepsilon}(\log D + \frac{n}{\varepsilon}))$. The dominant term of this expression is $\mathcal{O}(\frac{n^3}{\varepsilon^2})$, while the dominant term of our solution is $\mathcal{O}((1/\varepsilon^2) n \cdot H^2 \log \frac{n}{\varepsilon})$. It follows that, for most practical settings i.e., when H is significantly 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 $\mathcal{O}(\log n)$, in which case our algorithm is $\Omega(\frac{n^2}{\log^2 n \log(n/\varepsilon)})$ times faster. Furthermore, in the special case of balanced trees, the computational complexity of our solution is just $\mathcal{O}(\frac{1}{\varepsilon^2} n \log \frac{n}{\varepsilon} + n \log\log C^{\max})$, which is $\Omega(\frac{n^2}{\log(n/\varepsilon)})$ times faster than that of [7].

We described a precomputation scheme for Problem MOPQ that provides ε-optimal solutions within a computational complexity of $\mathcal{O}(\frac{1}{\varepsilon^2} n \cdot H^2 \log(n C^{\max}))$ for the first phase and $\mathcal{O}(\text{loglog}(C^{\max}) + \log \frac{1}{\varepsilon} + n)$ for the second phase. This precomputation scheme promptly provides a suitable partition upon an incoming request. The computational complexity of our scheme is significantly lower than that of simplistic adaptations of existing approximation algorithms.

V. CONCLUSION

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 be less dependent 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 of the 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 can be considered. For example, our layering approach makes it possible to 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.

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. The key observation there is that large-scale networks typically have a hierarchical layering structure, which provides the grounds for an efficient application of our "divide-and-conquer" approach.

APPENDIX

A. Proof of Lemma 1

Lemma 1: Given are layer-$(k+1)$ subpaths $\mathcal{P}_{(v_i,v_b)}$ and $\mathcal{P}_{(v_b,v_j)}$ with corresponding ε-approximate delay functions $D_{(v_i,v_b)}(c)$ and $D_{(v_b,v_j)}(c)$. Then, the execution of Procedure MERGE yields an $\hat{\varepsilon}$-approximate delay function $D_{(v_i,v_j)}(c)$ for the subpath $\mathcal{P}_{(v_i,v_j)}$, where $\hat{\varepsilon} = (1+\delta_k)(1+\varepsilon) - 1$.

Proof: Let $\hat{c}$ be an arbitrary cost. We denote by d^{opt} the minimum delay supported by $\mathcal{P}_{(v_i,v_j)}$ at cost $\hat{c}$, i.e., $d^{\text{opt}} = D^{\text{opt}}_{(v_i,v_j)}(\hat{c})$ and by $\{d_l^{\text{opt}}\}_{l \in \mathcal{P}_{(v_i,v_j)}}$ the optimal partition of delay d^{opt}. In addition, we denote by $\hat{d}_1 = \sum_{l \in \mathcal{P}_{(v_i,v_b)}} d_l^{\text{opt}}, \hat{d}_2 = \sum_{l \in \mathcal{P}_{(v_b,v_j)}} d_l^{\text{opt}}, \hat{c}_1 = \min\{c \,|\, D_{(v_i,v_b)}(c) \leq \hat{d}_1\}$, and $\hat{c}_2 = \min\{c \,|\, D_{(v_b,v_j)}(c) \leq \hat{d}_2\}$. The condition of the lemma implies that $\hat{c}_1 \leq (1+\varepsilon) \sum_{l \in \mathcal{P}_{(v_i,v_b)}} c_l(d_l^{\text{opt}})$ and $\hat{c}_2 \leq (1+\varepsilon) \sum_{l \in \mathcal{P}_{(v_b,v_j)}} c_l(d_l^{\text{opt}})$. This, in turn, implies that $\hat{c}_1 + \hat{c}_2 \leq (1+\varepsilon)\hat{c}$.

We prove that $D_{(v_i,v_j)}((1+\hat{\varepsilon})\hat{c}) \leq d^{\text{opt}}$. Consider the invocation of the loop that begins at line 2 for $c = \min_t\{(1+\delta_k)^t \,|\, (1+\delta_k)^t \geq \hat{c}_1 + \hat{c}_2\}$. Clearly, $\hat{c}_1 + \hat{c}_2 \leq c \leq (1+\delta_k)(\hat{c}_1 + \hat{c}_2)$.

We consider three possible cases.

1) $\hat{c}_1 \leq \frac{c}{2}$ and $\hat{c}_2 \leq \frac{c}{2}$. Then, after execution of line 3, it holds that $D_{(v_i,v_j)}(c) \leq D_{(v_i,v_b)}(\hat{c}_1) + D_{(v_b,v_j)}(\hat{c}_2) \leq \hat{d}_1 + \hat{d}_2 = d^{\text{opt}}$.

2) $\hat{c}_1 < \frac{c}{2}$ and $\hat{c}_2 > \frac{c}{2}$. Then, after the iteration of the loop that begins on line 6 for $c_2 = \hat{c}_2$, it holds that $D_{(v_i,v_j)}(c) \le D_{(v_i,v_b)}(\hat{c}_1) + D_{(v_b,v_j)}(\hat{c}_2) \le \hat{d}_1 + \hat{d}_2 = d^{\text{opt}}$.
3) $\hat{c}_2 < \frac{c}{2}$ and $\hat{c}_1 > \frac{c}{2}$. Then, after the iteration of the loop that begins on line 11 for $c_1 = \hat{c}_1$, it holds that $D_{(v_i,v_j)}(c) \le D_{(v_i,v_b)}(\hat{c}_1) + D_{(v_b,v_j)}(\hat{c}_2) \le \hat{d}_1 + \hat{d}_2 = d^{\text{opt}}$.

In both cases, we showed that there exists $c \le (1+\delta_k)(\hat{c}_1 + \hat{c}_2)$ for which $D_{(v_i,v_j)}(c) \le d^{\text{opt}}$. Since $\hat{c}_1 + \hat{c}_2 \le (1+\varepsilon)\hat{c}$, we have $c \le (1+\delta_k)(1+\varepsilon)\hat{c} = (1+\hat{\varepsilon})\hat{c}$, which in turn implies that $D_{(v_i,v_j)}((1+\hat{\varepsilon})\hat{c}) \le d^{\text{opt}}$. Since $\hat{c}$ is arbitrary, the lemma follows. ■

B. Proof of Lemma 2

Lemma 2: The computational complexity of Procedure MERGE is $\mathcal{O}(\frac{\log U}{\delta_k \delta_{k+1}})$.

Proof: First, let us count the number of iterations t of the procedure's main loop (i.e., the loop beginning on line 2). Clearly, $(1+\delta_k)^t \le U$ hence $t \le \frac{\log U}{\log(1+\delta_k)}$. Since for all $0 \le x \le 1$ it holds that $x \le 2\ln(1+x)$, we have $\log(1+\delta_k) = \mathcal{O}(\delta_k)$ and $t = \mathcal{O}(\frac{\log U}{\delta_k})$.

The loop that begins on line 6 is executed for values of c that belong to the set $\{c_2'(1+\delta_{k+1}), c_2'(1+\delta_{k+1})^2, \ldots, c\}$. Thus, the number of iterations of this loop is $\frac{\log(c/c_2')}{\log(1+\delta_{k+1})}$. We note that the value assigned to c_2' at line 4 is at least $\frac{c}{2}$, hence $\frac{c}{c_2'} \le 2$. Thus, and since $\log(1+\delta_{k+1}) = \mathcal{O}(\delta_{k+1})$ for $\delta_{k+1} \le 1$, it holds that the number of iterations in the loop is $\mathcal{O}(\frac{1}{\delta_{k+1}})$. A single iteration requires $\mathcal{O}(1)$ time, hence the running time of the loop is $\mathcal{O}(\frac{1}{\delta_{k+1}})$. By using the same argument, we can show that the running time of the loop that begins on line 11 is also $\mathcal{O}(\frac{1}{\delta_{k+1}})$.

We conclude that the computational complexity of the procedure is indeed $\mathcal{O}(\frac{\log U}{\delta_k \delta_{k+1}})$. ■

C. Proof of Claim 1

Claim 1: Let $\delta_0, \ldots, \delta_k$ be positive numbers such that $\sum_{k=0}^{K} \delta_k \le 1$. Then it holds that

$$\prod_{k=0}^{K}(1+\delta_k) \le 1 + 2\sum_{k=0}^{K}\delta_k.$$

Proof: We prove that $\ln(\prod_{k=0}^{K}(1+\delta_k)) \le \ln(1 + 2\sum_{k=0}^{K}\delta_k)$, the claim then follows from the monotonicity of $\ln(x)$.

Since for $0 \le x \le 1$, it holds that $\ln(1+x) \le x$, we have

$$\ln\left(\prod_{k=0}^{K}(1+\delta_k)\right) = \sum_{k=0}^{K}\ln(1+\delta_k) \le \sum_{k=0}^{K}\delta_k.$$

Next, we use that fact that for $0 \le x \le 1$, it holds that $x \le \ln(1+2x)$. Since $\sum_{k=0}^{K}\delta_k \le 1$ we have

$$\ln\left(\prod_{k=0}^{K}(1+\delta_k)\right) \le \sum_{k=0}^{K}\delta_k \le \ln\left(1 + 2\sum_{k=0}^{K}\delta_k\right).$$

■

D. Proof of Theorem 1

Theorem 1: Procedure UNICAST identifies, in $\mathcal{O}(\frac{1}{\varepsilon^2} n \log U)$ time, an ε-approximate delay function $D_{(v_0,v_n)}(c)$ for a path $\mathcal{P}$.

Proof: We begin by proving that Procedure UNICAST identifies, for each subpath of layer $k, 1 \le k \le K$, an $\varepsilon^{(k)}$-approximate delay function $D_{(v_i,v_j)}(c)$, where $\varepsilon^{(k)} = \prod_{t=k}^{K}(1+\delta_t) - 1$.

The proof is by induction on the layer number k. Consider a layer-K subpath $\mathcal{P}_{(v_i,v_{i+1})}$. It is immediate that lines 2–4 compute an δ_K-approximate delay function $D_{(v_i,v_{i+1})}(c)$. Assume inductively that the assertion holds for subpaths of layer-$(k+1)$, and consider a layer-k subpath $\mathcal{P}_{(v_i,v_j)}$. Since the assertion holds for the subpaths $\mathcal{P}_{(v_i,v_b)}$ and $\mathcal{P}_{(v_b,v_j)}$, $b = (j+i)/2$, the condition of Lemma 1 is satisfied for $\varepsilon^{(k+1)} = \prod_{t=k+1}^{K}(1+\delta_t) - 1$. Lemma 1 implies, in turn, that the algorithm identifies an $\varepsilon^{(k)}$-approximate delay function $D_{(v_i,v_j)}(c)$, for $\varepsilon^{(k)} = \prod_{t=k}^{K}(1+\delta_t) - 1$.

By Claim 1, $\prod_{t=0}^{K}(1+\delta_t) - 1 \le 2\sum_{t=0}^{K}\delta_t \le \varepsilon$. Thus, the assertion implies that Procedure UNICAST computes ε-approximate delay function $D_{(v_0,v_n)}(c)$ for path $\mathcal{P}$.

We proceed to analyze the computational complexity of Procedure UNICAST. The procedure is applied recursively for each subpath of $\mathcal{P}$ of each layer k. The total time required for processing layer-K paths is $\mathcal{O}(\frac{1}{\delta_K} n \cdot \log U)$. For $0 \le k \le K-1$, the time needed for processing a layer-k subpath is determined by the running time of Procedure MERGE. By Lemma 2, and since $\delta_{k+1} \ge \delta_k$, invocation of Procedure MERGE for a layer-k subpath requires $\mathcal{O}(\frac{\log U}{\delta_k \delta_{k+1}}) = \mathcal{O}(\frac{\log U}{\delta_k^2}) = \mathcal{O}(\frac{\sqrt[3]{2^{2(K-k)}}\log U}{\varepsilon^2})$ time.

Since there are 2^k subpaths of layer-k, processing layer-k requires $\mathcal{O}(\frac{\sqrt[3]{2^{2K+k}}\log U}{\varepsilon^2})$ time. The total time needed for processing each subpath of each layer is

$$\mathcal{O}\left(\left(\frac{2^K}{\varepsilon} + \frac{2^{\frac{2K}{3}}}{\varepsilon^2}\sum_{k=0}^{K-1} 2^{\frac{k}{3}}\right)\log U\right) = \mathcal{O}\left(\frac{1}{\varepsilon^2} n \cdot \log U\right).$$

We conclude the computational complexity of the algorithm is $\mathcal{O}(\frac{1}{\varepsilon^2} n \cdot \log U)$ and the theorem follows. ■

E. Proof of Theorem 2

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

Proof: In lines 1 and 2, we compute obvious lower and upper, L and U bounds on the cost of the optimal solution. As discussed in Section III-B2, the bounds remain valid during execution of the loop that begins at line 3 and after the execution of this loop it holds that $U/L \le 2n$.

We denote by c_{opt} and c^*_{opt} the cost of the optimal solution under the original and scaled cost functions, respectively. Equation (6) implies that $c^*_{\text{opt}} \le \frac{c_{\text{opt}} \cdot n}{(\varepsilon/2) \cdot L}$. By Theorem 1, Procedure UNICAST yields an $\varepsilon/2$-approximate delay function $D_{(v_0,v_n)}(c)$. Thus, after execution of line 12 it holds that $\hat{c} \le (1+\varepsilon/2)c^*_{\text{opt}}$. Since $c^*_{\text{opt}} \le \frac{c_{\text{opt}} \cdot n}{(\varepsilon/2) \cdot L}$, we have $\hat{c} \le (1+\varepsilon/2)\frac{c_{\text{opt}} \cdot n}{(\varepsilon/2) \cdot L}$. Let $\{d_l\}_{l \in \mathcal{P}}$ be the partition that corresponds to cost $\hat{c}$. From the left part of (6) it follows that the cost c of $\{d_l\}_{l \in \mathcal{P}}$ is at most $c \le \frac{\varepsilon \hat{c} \cdot L}{2n} + \frac{\varepsilon L}{2} \le (1+\varepsilon/2) \cdot c_{\text{opt}} + (\varepsilon/2) \cdot L \le (1+\varepsilon) \cdot c_{\text{opt}}$. We conclude that the algorithm returns a feasible partition whose cost is at most $(1+\varepsilon)$ times higher than the optimum.

We proceed to analyze the computational complexity of Algorithm OPQ. Lines 1 and 2 of the algorithm require $\mathcal{O}(n)$ time. Each iteration of the loop of line 3 requires also $\mathcal{O}(n)$ time. Since the total number of iterations is $\mathcal{O}(\log\log C^{\max})$, we conclude that the loop requires $\mathcal{O}(n \cdot \log\log C^{\max})$ time. Theorem 1 implies that the application of Procedure UNICAST for $U = \frac{4n^2}{\varepsilon}$ (line 11) requires $\mathcal{O}(\frac{1}{\varepsilon^2} n \cdot \log \frac{n}{\varepsilon})$ time. Thus, we conclude that the computational complexity of the algorithm is $\mathcal{O}(\frac{1}{\varepsilon^2} n \cdot \log\frac{n}{\varepsilon} + n \cdot \log\log C^{\max})$. ■

F. Proof of Lemma 3

Lemma 3: Algorithm POPQ computes, in $\mathcal{O}(\frac{1}{\varepsilon^2} n \log(nC^{\max}))$ time, an ε-approximate delay function $D_{(v_0,v_n)}(c)$ for $\mathcal{P}$.

Proof: By Theorem 1, Procedure UNICAST yields an $\varepsilon/3$-approximate delay function $D'_{(v_0,v_n)}(c)$.

Let $\hat{c}$ be an arbitrary cost. Since $D'_{(v_0,v_n)}(c)$ is an $\varepsilon/3$-approximate delay function, there exists $c' \leq (1+\varepsilon/3)\hat{c}$ such that $D'_{(v_0,v_n)}(c') \leq D^{\text{opt}}_{(v_0,v_n)}(\hat{c})$. Furthermore, let $c'' = \min\{(1+\varepsilon/3)^t \,|(1+\varepsilon/3)^t > c'\}$. Since $c'' \leq (1+\varepsilon/3)c' \leq (1+\varepsilon/3)^2\hat{c}$, it holds that $c'' \leq (1+\varepsilon)\hat{c}$ for $\varepsilon \leq 1$. After execution of the loop that begins at line 2 it holds that $D_{(v_0,v_n)}(c'') \leq D'_{(v_0,v_n)}(c')$. Hence for $c'' \leq (1+\varepsilon)\hat{c}$ it holds that $D_{(v_0,v_n)}(c'') \leq D^{\text{opt}}_{(v_0,v_n)}(\hat{c})$. Since $\hat{c}$ is arbitrary, $D_{(v_0,v_n)}(c)$ is an ε-approximate delay function for $\mathcal{P}$.

By Theorem 1, the application of Procedure UNICAST for $U = n \cdot C^{\max}$ (line 1) requires $\mathcal{O}((1/\varepsilon^2)n \cdot \log(nC^{\max}))$ time, which is also the computational complexity of Algorithm POPQ. ■

G. Proof of Lemma 4

Lemma 4: Let $\mathcal{T}_{(v_i,v_i)}$ be a layer-k subtree. Suppose that each branch $\mathcal{T}_{(v_i,v_j)}$ of $\mathcal{T}_{(v_i,v_i)}$ has a corresponding ε-approximate delay function $D_{(v_j,v_j)}(c)$ that includes $\mathcal{O}(\frac{\log U}{\delta_k})$ segments. Then, Procedure MIN-MAX-MERGE computes, in $\mathcal{O}(\frac{1}{\delta_k} m_i \log m_i \log U)$ time, an ε-approximate delay function $D_{(v_i,v_i)}(c)$ for the subtree $\mathcal{T}_{(v_i,v_i)}$.

Proof: First, we prove the following claim: at each iteration of the loop that begins on line 6 for each $(v_i,v_j) \in \mathcal{T}$ it holds that c_j is the minimum cost of supporting delay requirement d, i.e., $c_j = \min\{c\,|\,D_{(v_i,v_j)}(c) \leq d\}$. Clearly, the claim holds at the beginning of iteration 1. Suppose inductively that, the claim holds at the beginning of iteration k, we prove that the claim holds at the end of the iteration. We denote the value of d at the beginning of the iteration by d' and in the end of the iteration by d''. Note that $d'' < d'$ and in the end of the iteration it holds that $D_{(v_i,v_j)}(c_j) \leq d''$ for each $(v_i,v_j) \in \mathcal{T}$. Thus, for each $j \notin S$, since the value of c_j does not change during the iteration, it holds that c_j is a minimum cost of supporting d''. For each $j \in S, c_j$ is set to minimum cost of supporting a delay lower than d'. Thus, since $d'' < d'$ and $D_{(v_i,v_j)}(c_j) \leq d''$, it holds that c_j is a minimum cost of supporting d''.

Next, we prove that, for arbitrary cost $\hat{c}, 1 \leq \hat{c} \leq U$, it holds that $D_{(v_i,v_i)}((1+\varepsilon)\hat{c}) \leq D^{\text{opt}}_{(v_i,v_i)}(\hat{c})$. We denote by $\hat{d}^{\text{opt}}$ the minimum delay supported by $\mathcal{T}_{(v_i,v_i)}$ at cost $\hat{c}$, i.e., $\hat{d}^{\text{opt}} = D^{\text{opt}}_{(v_i,v_i)}(\hat{c})$. In addition, we denote, for each $(v_i,v_j) \in \mathcal{T}, \hat{c}^{\text{opt}}_j = \min\{c\,|\,D^{\text{opt}}_{(v_i,v_j)}(c) \leq \hat{d}^{\text{opt}}\}, \hat{c}_j = \min\{c\,|\,D_{(v_i,v_j)}(c) \leq \hat{d}^{\text{opt}}\}, \hat{d}_j = D_{(v_i,v_j)}(\hat{c}_j)$. Let $\hat{d} = \max_{(v_i,v_j)\in\mathcal{T}} \hat{d}_j$.

The condition of the lemma implies that, for each $(v_i,v_j) \in \mathcal{T}$, it holds that $\hat{c}_j \leq (1+\varepsilon)\hat{c}^{\text{opt}}_j$. Consider the iteration of the loop that begins on line 6 in which $d = \hat{d}$. The claim above implies that, for each $(v_i,v_j) \in \mathcal{T}$, it holds that $c_j = \min\{c\,|\,D_{(v_i,v_j)}(c) \leq \hat{d}\} = \hat{c}_j$. Thus, for each $(v_i,v_j) \in \mathcal{T}$, it holds that $c_j \leq (1+\varepsilon)\hat{c}^{\text{opt}}_j$ and, after execution of line 7 we have $D_{(v_i,v_i)}(c) \leq \hat{d}^{\text{opt}}$, where $c = \sum_{(v_i,v_j)\in\mathcal{T}} c_j \leq (1+\varepsilon)\hat{c}^{\text{opt}}$. We thus proved that $D_{(v_i,v_i)}(c)$ is a ε-approximate delay function of $\mathcal{T}_{(v_i,v_i)}$.

We proceed to analyze the computational complexity of Procedure MIN–MAX–MERGE. The loop that begins at line 1 requires $\mathcal{O}(m_i)$ time. At each iteration of the loop that begins at line 6, we examine a segment of $D_{(v_i,v_j)}(c)$ for some branch $\mathcal{T}_{(v_i,v_j)}$ of $\mathcal{T}_{(v_i,v_i)}$. Since the delay function of the branch has $\mathcal{O}(\frac{\log U}{\delta_k})$ segments, the number of iterations of the loop is $\mathcal{O}(\frac{1}{\delta_k} m_i \log U)$. All lines in the loop, except for lines 10–12 and 13, can be executed in $\mathcal{O}(1)$ time. The total computational complexity of line 10 is $\mathcal{O}(\frac{1}{\delta_k} m_i \log U)$. If we use a binary tree to keep values of d_j, then the total computational complexity of lines 11 and 13 is $\mathcal{O}(\frac{1}{\delta_k} m_i \log m_i \log U)$. Line 12 returns the sum c of budgets c_j allocated to each branch $\mathcal{T}_{(v_i,v_j)}$. If we maintain c by updating it each time c_j changes, then the total time required for line 12 is equal to that required for line 10, i.e., $\mathcal{O}(\frac{1}{\delta_k} m_i \log U)$. We conclude that the total computational complexity of the procedure is $\mathcal{O}(\frac{1}{\delta_k} m_i \log m_i \log U)$. ■

H. Proof of Theorem 3

Theorem 3: Procedure MULTICAST identifies, in $\mathcal{O}(\frac{1}{\varepsilon^2} nH^2 \log U)$ time, an ε-approximate delay function $D_{(s,s)}(c)$ for a tree $\mathcal{T}$.

Proof: We showed above (see Section IV.A.2) that Procedure MULTICAST identifies an $\varepsilon^{(0)}$-approximate delay function $D_{(s,s)}(c)$ for tree $\mathcal{T}$, where $\varepsilon^{(0)} = \prod_{k=0}^{H-1}(1+\delta_k)^2 - 1$. Note that $\prod_{k=0}^{H-1}(1+\delta_k)^2 = \prod_{k=0}^{H-1}(1+2\delta_k+\delta_k^2) \leq \prod_{k=0}^{H-1}(1+3\delta_k)$ and $\prod_{k=0}^{H-1}(1+3\delta_k) - 1 \leq \sum_{k=0}^{H-1} 6\delta_k$, where the last inequality follows from Claim 1. After substituting for δ_k, according to (10) we have $\sum_{k=0}^{H-1} 6\delta_k \leq \varepsilon$. We conclude that the procedure computes an ε-approximate delay function $D_{(s,s)}(c)$ for the tree $\mathcal{T}$.

We proceed to analyze the computational complexity of Procedure MULTICAST. The complexity is dominated by time required to execute Procedure MERGE, which is executed for each subtree of each layer. Since Procedure MERGE is applied for functions with $\mathcal{O}(\frac{1}{\delta_k}\log U)$ segments, its computational complexity for a branch of layer-k subtree is $\mathcal{O}(\frac{\log U}{\delta_k^2})$ (by Lemma 2). As the number of branches of layer-k subtrees is n_{k+1}, the time required for invoking Procedure MERGE for layer k subtrees is $\mathcal{O}(\frac{n_{k+1}\log U}{\delta_k^2})$. Thus, the total time T_1 required for execution of Procedure MERGE for all layers is

$\mathcal{O}(\log U \sum_{k=0}^{H-1} \frac{n_{k+1}}{\delta_k^2})$. After substituting for δ_k, according to (10) we have

$$T_1 = \mathcal{O}\left(\frac{\log U}{\varepsilon^2}\left(\sum_{k=1}^{H}\sqrt[3]{n_k}\right)^2 \sum_{k=0}^{H-1}\sqrt[3]{n_{k+1}}\right)$$
$$= \mathcal{O}\left(\frac{\log U}{\varepsilon^2}\left(\sum_{k=1}^{H}\sqrt[3]{n_k}\right)^3\right).$$

Since the latter expression is maximized when $n_k = n/(H-1)$ for $k = 1,\dots,H$, we have $T_1 = \mathcal{O}(\frac{1}{\varepsilon^2} nH^2 \log U)$.

Next, we analyze the total time T_2 required for execution of Procedure MIN–MAX–MERGE. The procedure is also executed for each subtree of each layer. By Lemma 4, the computational complexity of executing the procedure for a subtree $T_{(v_i,v_i)}$ of layer k is $\mathcal{O}(\frac{1}{\delta_k} m_i \log m_i \log U)$, where $m_i \leq n$ is the number of branches of $T_{(v_i,v_i)}$. Thus, $T_2 = \mathcal{O}(\log n \log U \sum_{k=0}^{H-1} \frac{n_{k+1}}{\delta_k})$. After substituting for δ_k, according to (10) we have

$$T_2 = \mathcal{O}\left(\frac{\log n \log U}{\varepsilon} \sum_{k=1}^{H}\sqrt[3]{n_k} \sum_{k=0}^{H-1}\sqrt[3]{n_{k+1}^2}\right).$$

Again, the latter expression is maximized when $n_k = n/(H-1)$ for $k = 1,\dots,H$, we have $T_2 = \mathcal{O}(\frac{1}{\varepsilon} nH \log n \log U)$.

Finally, in procedure Procedure MULTICAST, we perform logarithmic sampling for each link $l \in \mathcal{T}$ and each subtree $\mathcal{T}_{(v_i,v_i)}$ of $\mathcal{T}$ (lines 7–10). Performing logarithmic sampling for each function requires $\mathcal{O}(\frac{1}{\delta_k}\log U)$ time. Thus, this operation requires $\mathcal{O}(\frac{1}{\delta_k} n_{k+1} \log U)$ time for layer-$(k+1)$ and $T_3 = \mathcal{O}(\log U \sum_{k=0}^{H-1} \frac{1}{\delta_k} n_{k+1})$ overall. After substituting for δ_k according to (10), we conclude that the time required to process all links in $\mathcal{T}$ is $T_3 = \mathcal{O}(\frac{1}{\varepsilon} nH \log U)$. We conclude that the computational complexity of Procedure MULTICAST is $T_1 + T_2 + T_3 = \mathcal{O}(\frac{1}{\varepsilon^2} nH^2 \log U)$ and the theorem follows. ■

REFERENCES

[1] S. Blake, "An Architecture for Differentiated Services.—RFC no. 2475," Internet Engineering Task Force, Dec. 1998.

[2] *Private Network-Network Interface Specification v1.0 (PNNI)*, Mar. 1996.

[3] D. H. Lorenz and A. Orda, "Optimal partition of QoS requirements on unicast paths and multicast trees," *IEEE/ACM Trans. Netw.*, vol. 10, no. 1, pp. 102–114, Feb. 2002.

[4] L. Z. F. Ergun and R. Sinha, "QoS routing with performance-dependent costs," in *Proc. IEEE INFOCOM*, Tel-Aviv, Israel, Mar.–Apr. 2000, pp. 137–146.

[5] V. Firoiu and T. Towsley, "Call admission and resource reservation for multicast sessions," in *Proc. IEEE INFOCOM*, San Francisco, CA, Apr. 1996, pp. 94–101.

[6] M. Kodialam and S. Low, "Resource allocation in a multicast tree," in *Proc. IEEE INFOCOM*, New York, Mar. 1999, pp. 262–266.

[7] D. Lorenz, A. Orda, D. Raz, and Y. Shavitt, "Efficient QoS partition and routing of unicast and multicast," in *Proc. IEEE/IFIP IWQoS*, Pittsburgh, PA, Jun. 2000, pp. 75–83.

[8] D. Raz and Y. Shavitt, "Optimal partition of QoS requirements with discrete cost functions," *IEEE J. Sel. Areas Commun.*, vol. 18, no. 12, pp. 2593–2602, Dec. 2000.

[9] R. Guérin and A. Orda, "Computing shortest paths for any number of hops," *IEEE/ACM Trans. Netw.*, vol. 10, no. 5, pp. 613–620, Oct. 2002.

[10] A. Orda and A. Sprintson, "Precomputation schemes for QoS routing," *IEEE/ACM Trans. Netw.*, vol. 11, no. 4, pp. 578–591, Aug. 2003.

Ariel Orda (S'84–M'92–SM'97) received the B.Sc. (*summa cum laude*), M.Sc., and D.Sc. degrees in electrical engineering from the Technion—Israel Institute of Technology, Haifa, Israel, in 1983, 1985, and 1991, respectively.

Since 1994, he has been with the Department of Electrical Engineering, Technion, where he is currently an Associate Professor. His current research interests include network routing, network survivability, QoS provisioning, wireless networks, the application of game theory to computer networking, and network pricing.

Prof. Orda was the recipient of the Award of the Chief Scientist in the Ministry of Communication in Israel, the Gutwirth Award for Outstanding Distinction, the Research Award of the Association of Computer and Electronic Industries in Israel, the Jacknow Award for Excellence in Teaching, and the Best Paper Award at IEEE ICNP 2004. He served as a Technical Program co-chair of IEEE Infocom 2002. He is an Editor of the IEEE/ACM TRANSACTIONS ON NETWORKING and of the *Journal of Computer Networks.*

Alexander Sprintson (S'00–M'03) received the B.Sc. degree (*summa cum laude*), M.Sc., and Ph.D. degrees in electrical engineering from the Technion—Israel Institute of Technology, Haifa, Israel, in 1995, 2001, and 2003, respectively.

Since 2003, he has been with the Department of Engineering and Applied Science, California Institute of Technology, Pasadena, where he is currently a Postdoctoral Research Fellow. During the summers of 2002 and 2003, he was with the Internet Management Research Department, Bell Laboratories, Murray Hill, NJ. His research interests lie in the areas of QoS provisioning, network survivability, resource allocation, network coding, and wireless data broadcast.

Dr. Sprintson was the recipient of the Wolf Award for Distinguished Ph.D. Students, the Gutwirth Award for Outstanding Distinction, and the Knesset (Israeli Parliament) Award for Distinguished Students.