Optimal Partition of QoS Requirements for Many-to-Many Connections

Dean H. Lorenz*
Distributed Computer Systems
IBM Haifa Research Labs
Haifa, Israel
Email: dean@il.ibm.com

Ariel Orda
Department of Electrical Engineering
Technion – Israel Institute of Technology
Haifa, Israel
Email: ariel@ee.technion.ac.il

Danny Raz
Department of Computer Science
Technion – Israel Institute of Technology
Haifa, Israel
Email: danny@cs.technion.ac.il

***Abstract*— We study problems related to supporting multicast connections with Quality of Service (QoS) requirements. We investigate the problem of optimal resource allocation in the context of performance dependent costs. In this context each network element can offer several QoS guarantees, each associated with a different cost. This is a natural extension to the commonly used bi-criteria model, where each link is associated with a single delay and a single cost. This framework is simple yet strong enough to model many practical interesting networking problems.**

The fundamental multicast resource allocation problem under this framework is how to optimally allocate QoS requirements on the links of the multicast tree. One needs to partition the end-to-end QoS requirement along the various paths in a tree. The goal is to satisfy the end-to-end QoS requirement with minimum cost. Previous studies under this framework considered *single-source* multicast connections, where the End-to-end QoS requirement is specified from the source to all other multicast group members. In this paper we extend these results to the more general, and considerably harder case of *multicast sessions*, where the end-to-end requirement hold for every path between any two multicast group members. Our aim is to provide rigorous solutions, with proven performance guaranties, by way of algorithmic analysis.

The problem under investigation is NP hard for general cost functions, thus we first present a pseudo-polynomial exact solution. From this solution we derive two efficient ϵ-approximate solutions. One achieves optimal cost, but may violate the end-to-end delay requirement by a factor of $(1+\epsilon)$, and the other strictly obeys the bounds and achieves a cost within a factor of $(1+\epsilon)$ of the optimum. Furthermore, we present improved results for discrete cost functions, and give a simple linear-time exact polynomial solution for a specific, and practically interesting, family of convex cost functions.

I. Introduction

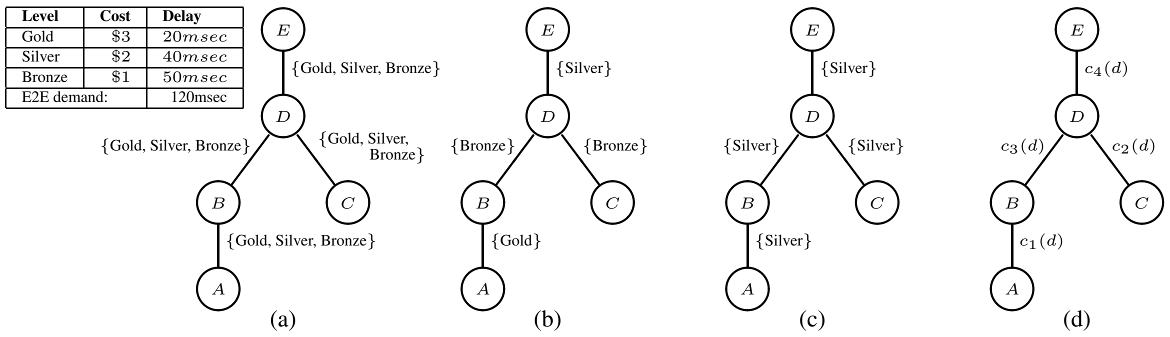
Consider a conference call that uses IP telephony. Such an application requires a delay bound of say $120msec$ in order to be at an acceptable quality level. Assume further that the participants in the meeting are located at different locations and the links among them can travel through different administrative domains. One could construct different trees connecting the participants in the conference call, each resulting in different QoS (delay) guaranteed by the different providers according to the different SLAs we have with each one of them. Even if the tree is fixed (*i.e.*, we cannot create the multicast tree), we could still decide to partition the delay "budget" in various ways among the different providers. Among all such partitions that guarantee the required QoS, we would seek to use the one that is most cost-effective.

This problem is precisely the many-to-many QoS partition problem on trees, which is the subject of this study. In this problem, we are given a set of end-points connected through a fixed given tree. Each link in this tree is associated with a cost-delay function that indicates the cost per delay guarantee on the link. We are also given a global bound on the delay between any two participants. The objective is to find the least expensive partition of the delay along the different links in the tree in a way that the delay between *any* two end-points (along the unique path in the tree) will be bounded by the global bound.

The tele-conferencing example described above is an instance of this partition problem, where each link corresponds to a provider's network. Each link is associated with several working points each representing a possible guaranteed delay and its cost. Fig. 1 shows how this problem can be modeled for a very simple example. In this example each network element offers three levels of service: *Gold*, which costs \$3 and guarantees a delay of $20msec$; *Silver*, which costs \$2 and guarantees $40msec$; and *Bronze* which costs only \$1, but guarantees $50msec$. There are different possibilities for ensuring a many-to-many bound of no more than $120msec$ and the problem is to find the least costly choice. One can consider different examples in which each link is associated with a general resource-cost function.

The cost may represent the consumption of local resources (such as buffer or bandwidth reservations) that must be reserved on every link of the route to support the QoS guarantee. The cost may also represent the decrease in overall network performance resulting from establishing the selected connection. For instance, there may be loss of revenue due to blocked future calls or there may be management costs. If we assume some of the suggested pricing schemes for QoS provision, the cost may represent a real price that the user has to pay in order to use the link. All these examples indicate that the performance-dependent model is of practical importance, and any efficient solutions established within its framework may become essential for future network planning and control. This is particularly true as applications like

*Part of this work done while Dean Lorenz was with the Department of Electrical Engineering, Technion – Israel Institute of Technology, Haifa, Israel.



Level	Cost	Delay
Gold	\$3	$20msec$
Silver	\$2	$40msec$
Bronze	\$1	$50msec$
E2E demand:		120msec

Fig. 1. Example: The many-to-many partition problem

The multicast tree (a) offers three levels of service on each link: {Gold, Silver, Bronze} associated with delays $\{20msec, 40msec, 50msec\}$ and costs {\$3, \$2, \$1}. The desired end-to-end delay bound is $120msec$. The allocations (b), (c) demonstrate that, even in this simple case of identical links, there are several possible ways to split the delay between the links. The end-to-end delay between any two nodes in the tree must satisfy the connection requirements of $120msec$. Option (b), which induces an overall cost of \$7 is better than option (c), which costs \$8. In the general case (d), there is an arbitrary cost function on each link, which maps delay guarantees to cost.

video conferencing and video streaming over wireless are expected to become popular, and as different providers are deploying various pricing schemes in order to create revenue from the new services offered. Accordingly, this study investigates the problem of many-to-many multicast sessions within the performance-dependent model, and employs algorithmic analysis in order to provide rigorous solutions, with proven performance guaranties.

Similar problems, generally referred to as QoS routing and partitioning, were considered recently, both for unicast and for multicast [1]–[3]. However, in the context of multicast they all dealt with the restricted case of *one*-to-many connections, where the QoS requirement is just between the common root to the other members. Since we deal with *many*-to-many connections, the bound on the delay should be valid for any pair of end users and not only from the root to the end users. This requirement makes the problem different, and inherently more difficult. Since the tree no longer has a clear root, building a dynamic solution based on the tree structure becomes more complex.

In general, the partition problem is intractable. The special case where the link cost functions (*i.e.*, the functions that describe the cost of allocating a QoS parameter on a link) are continuous and convex was addressed recently by several works. Multicast trees for the strongly convex case were dealt in [4]. In [5], polynomial algorithms both for trees and paths for weakly convex cost functions were presented, and the QoS routing problem was also addressed. The discrete cost function case was addressed in [6], and approximation algorithms were considered in [1], [2], [7].

The specific aspect of resource allocation in this context has also been extensively studied; in particular, a similar framework was studied by [3]–[5], [8], [9]. The reader is also referred to [10] for a survey on QoS multicast routing algorithms, though from a slightly different perspective.

In this paper we provide exact and approximated solutions for various cases. We first present a pseudo polynomial solution that uses dynamic programming to find the best partition of the tree. We then describe a set of steps that allow us to use this algorithm in order to derive a polynomial approximation algorithm for the problem. We note that, while the techniques used for these results are similar to the one used in [2], [5], the problem in our case is significantly more complex and thus the actual algorithms (and their proofs) are much more complex. We then deal with specific cost-delay function which are of practical importance. These include discrete cost functions (which reflect the current DiffServe architecture [11]), and a specific case of a convex cost function that is often used in this context. For the latter, we present a remarkably efficient linear algorithm that finds the best partition of the delay along a given tree.

The rest of the paper is organized as follows. In the next section we give the basic notions and preliminary observations. Then, in Section III, we present the basic pseudo polynomial algorithm. In section IV we describe the approximation algorithms, and in Section V we present particularly efficient algorithms for some cost-delay functions of special interest. We conclude with a brief discussion.

II. Preliminaries

A. Terminology

The network is represented as an undirected graph $G(V, E)$. We are given a multicast group $M \subseteq V$ and a multicast tree $\boldsymbol{T} \subseteq E$, such that for all $u, v \in M$, there exists a unique path, $\boldsymbol{p}_{u \rightsquigarrow v}$ from u to v, on links that belong to $\boldsymbol{T}$. We use $\boldsymbol{T}$ to denote both the links and the nodes of a tree, and denote by $n = |\boldsymbol{T}|$ its size (number of links). $N(u)$ denotes all the outgoing links from u, namely $N(u) \equiv \{(u, v) \in \boldsymbol{T}\}$. If the degree of node u is one, *i.e.*, $N(u)$ consists of a single link (u, v), then we use the term *leaf* for both u and (u, v). We assume that $\boldsymbol{T}$ is a binary tree, namely $|N(u)| \leq 3$ for all $u \in \boldsymbol{T}$. This assumption greatly simplifies the presentation without loss of generality.[1]

We denote by $\boldsymbol{T}^{uv}$ the sub-tree rooted at u, for which (u, v) is the first (root) link. For example, for any node $u \in \boldsymbol{T}$ we have $\boldsymbol{T} = \cup_{v \in N(u)} \boldsymbol{T}^{uv}$. Although links are undirected, note that $\boldsymbol{T}^{uv} \neq \boldsymbol{T}^{vu}$, since these denote different sub-trees on the two sides of the link. Also, note that $\boldsymbol{T}^{uv} \cup \boldsymbol{T}^{vu} = \boldsymbol{T}$, and $\boldsymbol{T}^{uv} \cap \boldsymbol{T}^{vu} = \{(u, v)\}$. We use $\mathcal{L}_{uv}, \mathcal{R}_{uv}$ respectively to denote the "left" and "right" child of node v on $\boldsymbol{T}^{uv}$, and the corresponding sub-trees are denoted by $\boldsymbol{T}^{\mathcal{L}_{uv}}$ and $\boldsymbol{T}^{\mathcal{R}_{uv}}$. We call the union of both sub-trees the *branches* of $\boldsymbol{T}^{uv}$ and use $\boldsymbol{T}^{\mathcal{B}_{uv}} \equiv \boldsymbol{T}^{\mathcal{L}_{uv}} \cup \boldsymbol{T}^{\mathcal{R}_{uv}} \equiv \boldsymbol{T}^{uv} \setminus \{(u, v)\}$.

Each link $l \in E$ offers different delays with different costs. The cost associated with a link is a function $c_l(d_l)$ of the delay allocated to it. Each link cost function $c_l(d_l)$ is non-increasing with the delay and both the delays and associated costs are assumed to be integers.[2] A delay *partition* on a tree $\boldsymbol{T}$ specifies the delay allocated on each link, *i.e.*, is a set of link delays $\boldsymbol{d} \equiv \{d_l\}_{l \in \boldsymbol{T}}$. The cost of a partition $\boldsymbol{d}$ on a tree $\boldsymbol{T}$ is defined as the sum of all link costs, namely $cost(\boldsymbol{T}, \boldsymbol{d}) \equiv \sum_{l \in \boldsymbol{T}} c_l(d_l)$.

A given partition determines the end-to-end delay of any path $\boldsymbol{p} \subseteq \boldsymbol{T}$. Since delay QoS requirements are additive, the end-to-end delay of a path is the sum of the delays allocated on its links, *i.e.*, $delay(\boldsymbol{p}, \boldsymbol{d}) \equiv \sum_{l \in \boldsymbol{p}} d_l$. We define two end-to-end delay metrics for the whole tree which we term *depth* and *width*. The depth of a tree is the maximal end-to-end delay of any path from its root, and the width of a tree is the maximum delay between any two of its nodes. More formally, the depth and width of a tree $\boldsymbol{T}^{uv}$ are defined as $depth(\boldsymbol{T}^{uv}, \boldsymbol{d}) \equiv \max_{w \in \boldsymbol{T}^{uv}} delay(\boldsymbol{p}_{u \rightsquigarrow w}, \boldsymbol{d})$, and $width(\boldsymbol{T}^{uv}, \boldsymbol{d}) \equiv \max_{x,y \in \boldsymbol{T}^{uv}} delay\big(\boldsymbol{p}_{x \rightsquigarrow y}, \boldsymbol{d}\big)$. We say that a partition $\boldsymbol{d}$ is *D-deep* on $\boldsymbol{T}^{uv}$ if $depth(\boldsymbol{T}^{uv}, \boldsymbol{d}) \leq D$, namely $delay(\boldsymbol{p}_{u \rightsquigarrow w}, \boldsymbol{d}) \leq D$ for all $w \in \boldsymbol{T}^{uv}$. Similarly, we say that a partition $\boldsymbol{d}$ is *D-wide* on $\boldsymbol{T}$ if $width(\boldsymbol{T}, \boldsymbol{d}) \leq D$, namely $delay\big(\boldsymbol{p}_{x \rightsquigarrow y}, \boldsymbol{d}\big) \leq D$ for all $x, y \in \boldsymbol{T}^{uv}$. We use the shorthand $D_{uv}(\boldsymbol{d}) \equiv depth(\boldsymbol{T}^{uv}, \boldsymbol{d})$ and may omit the argument $\boldsymbol{d}$ when it is clear from context.

B. Problems Definition

We can now define the problems of interest, namely Problem WPQ and Problem DPQ for multicast trees.

Problem DPQ (Depth optimal Partition of QoS): Given a multicast tree $\boldsymbol{T}^{uv}$ and an end-to-end delay requirement D from u to all other $m \in M$, find a D-deep partition $\boldsymbol{d}^*$, such that $cost(\boldsymbol{d}^*) \leq cost(\boldsymbol{d})$, for every (other) D-deep partition $\boldsymbol{d}$ on $\boldsymbol{T}^{uv}$.

Problem WPQ (Width optimal Partition of QoS): Given a multicast tree $\boldsymbol{T}$ and an end-to-end delay requirement D between any two members in M, find a D-wide partition $\boldsymbol{d}^*$, such that $cost(\boldsymbol{d}^*) \leq cost(\boldsymbol{d})$, for every (other) D-wide partition $\boldsymbol{d}$ on $\boldsymbol{T}$.

[1]If the degree at any node is greater than 3 we can add "dummy" links to split the node. Overall, at worst, this doubles the number of links.

[2]This is true for any real-life network since neither link reservations nor payments can be made with arbitrary precision.

Note Both problems are NP-hard. In [12] it is shown that Problem DPQ is NP-hard even for a simple path topology. Since for a path both problems are equivalent, Problem WPQ is also intractable.

III. Solution to Problem WPQ

A. Foundations

In this section we show that any optimal partition on a tree is made of optimal partitions on sub-trees. This tree decomposition allows for the dynamic programming solution, which we present in the next section. The observation made in the following lemmas are quite intuitive, however their proofs require careful formulation. It is possible to skip directly to Section III-B, which is self-contained.

We start with Lemma 1, which states that all sub-trees of a D-wide tree are also D-wide.

Lemma 1: Let $\boldsymbol{d}$ be a D-wide partition on $\boldsymbol{T}$ and let $\boldsymbol{T}'$ be any sub-tree of $\boldsymbol{T}$. Then, $\boldsymbol{d}$ is a D-wide partition on $\boldsymbol{T}'$.

Proof: Since $\boldsymbol{d}$ is D-wide, $delay\big(\boldsymbol{p}_{x \rightsquigarrow y}\big) \leq D$ for all $x, y \in \boldsymbol{T}$ implying $delay\big(\boldsymbol{p}_{x \rightsquigarrow y}\big) \leq D$ for all $x, y \in \boldsymbol{T}' \subseteq \boldsymbol{T}$. Hence, $\boldsymbol{d}$ is D-wide partition on $\boldsymbol{T}'$. ■

Note that, although all sub-tree have width not greater than D, Lemma 1 does not imply that it is *exactly* D. In fact, their actual width may be much smaller than D. Also, note that a partition is D-wide *iff* it is D-deep on every $\boldsymbol{T}^{uv} \subseteq \boldsymbol{T}$.

Lemma 2: Let $\boldsymbol{d}$ be a D-wide partition on $\boldsymbol{T}$, let $\boldsymbol{T}^{uv}$ be any sub-tree of $\boldsymbol{T}$, and let $w \in \boldsymbol{T} \setminus \boldsymbol{T}^{uv}$ be a node outside the sub-tree. Then, $D_{uv} + delay(\boldsymbol{p}_{u \rightsquigarrow w}) \leq D$.

Proof: Let y be a D_{uv}-deep node in $\boldsymbol{T}^{uv}$, namely $delay\big(\boldsymbol{p}_{y \rightsquigarrow u}\big) = D_{uv}$ (such a node must exist in $\boldsymbol{T}^{uv}$ by the definition of the sub-tree depth, D_{uv}). Since w is outside $\boldsymbol{T}^{uv}$, $\boldsymbol{p}_{y \rightsquigarrow w} = \boldsymbol{p}_{y \rightsquigarrow u} + \boldsymbol{p}_{u \rightsquigarrow w}$.[3] Combining with the fact that this is a D-wide partition, we get

$$\begin{aligned} D_{uv} + delay(\boldsymbol{p}_{u \rightsquigarrow w}) &= \\ delay\big(\boldsymbol{p}_{y \rightsquigarrow u}\big) + delay(\boldsymbol{p}_{u \rightsquigarrow w}) &= \\ delay\big(\boldsymbol{p}_{y \rightsquigarrow w}\big) &\leq D, \end{aligned}$$

as claimed. ■

Lemma 3 provides the relation between the depths of adjacent sub-trees.

Lemma 3: Let $\boldsymbol{d}$ be a D-wide partition on $\boldsymbol{T}^{uv}$. Then $D_{\mathcal{L}_{uv}} + D_{\mathcal{R}_{uv}} \leq D$.

Proof: By Lemma 1, $\boldsymbol{d}$ is a D-wide partition on $\boldsymbol{T}^{\mathcal{B}_{uv}}$. We can apply Lemma 2 on $\boldsymbol{T}^{\mathcal{L}_{uv}}$ as a sub-tree of $\boldsymbol{T}^{\mathcal{B}_{uv}}$, therefore $D_{\mathcal{L}_{uv}} + delay(\boldsymbol{p}_{v \rightsquigarrow w}) \leq D$ for all $w \in \boldsymbol{T}^{\mathcal{R}_{uv}}$. Hence,

$$D_{\mathcal{L}_{uv}} + \max_{w \in \boldsymbol{T}^{\mathcal{R}_{uv}}} delay(\boldsymbol{p}_{v \rightsquigarrow w}) = D_{\mathcal{L}_{uv}} + D_{\mathcal{R}_{uv}} \leq D.$$

■

We will use Lemma 3 as a feasibility test when searching for an optimal partition. Also, note that the lemma can be applied in any "direction" and by symmetry we have $D_{\mathcal{L}_{uv}} + D_{vu} \leq D$ and $D_{\mathcal{R}_{uv}} + D_{vu} \leq D$.

[3]Here '+' is a path concatenation operator.

We proceed to show that an optimal partition is also optimal on sub-trees. But first, we need Lemma 4 that allows us to build a new partition by replacing part of an old one.

Lemma 4: Let $\boldsymbol{d^1}$ be a D-wide partition on $\boldsymbol{T}$, let $\boldsymbol{T}^{uv}$ be any sub-tree of $\boldsymbol{T}$ and let $\boldsymbol{d^2}$ be both a D-wide and D^1_{uv}-deep partition on $\boldsymbol{T}^{uv}$. Then, the partition

$$d_l = \begin{cases} d^2_l & l \in \boldsymbol{T}^{uv} \\ d^1_l & l \in \boldsymbol{T}^{\mathcal{B}_{vu}} \end{cases}$$

is a D-wide partition on $\boldsymbol{T}$.

Proof: $\boldsymbol{T}^{uv} \cup \boldsymbol{T}^{\mathcal{B}(v,u)} = \boldsymbol{T}$ and $\boldsymbol{T}^{uv} \cap \boldsymbol{T}^{\mathcal{B}(v,u)} = \emptyset$, thus for any $x, y \in \boldsymbol{T}$ we have three cases:

(1) $x, y \in \boldsymbol{T}^{uv}$:
In this case, $\boldsymbol{p}_{x \rightsquigarrow y} \subseteq \boldsymbol{T}^{uv}$ so

$$delay\big(\boldsymbol{p}_{x \rightsquigarrow y}, \boldsymbol{d}\big) = delay\big(\boldsymbol{p}_{x \rightsquigarrow y}, \boldsymbol{d^2}\big)\,.$$

Since $\boldsymbol{d^2}$ is a D-wide partition on $\boldsymbol{T}^{uv}$, we get

$$delay\big(\boldsymbol{p}_{x \rightsquigarrow y}, \boldsymbol{d}\big) = delay\big(\boldsymbol{p}_{x \rightsquigarrow y}, \boldsymbol{d^2}\big) \le D.$$

(2) $x, y \notin \boldsymbol{T}^{uv}$:
Here $x, y \in \boldsymbol{T}^{\mathcal{B}(v,u)}$, so

$$delay\big(\boldsymbol{p}_{x \rightsquigarrow y}, \boldsymbol{d}\big) = delay\big(\boldsymbol{p}_{x \rightsquigarrow y}, \boldsymbol{d^1}\big)\,.$$

Similarly to the previous case, we have

$$delay\big(\boldsymbol{p}_{x \rightsquigarrow y}, \boldsymbol{d}\big) = delay\big(\boldsymbol{p}_{x \rightsquigarrow y}, \boldsymbol{d^1}\big) \le D.$$

(3) $x \in \boldsymbol{T}^{uv}, y \notin \boldsymbol{T}^{uv}$:
Here $\boldsymbol{p}_{x \rightsquigarrow u} \subseteq \boldsymbol{T}^{uv}$ and $\boldsymbol{d^2}$ is D^1_{uv}-deep on $\boldsymbol{T}^{uv}$, hence $delay\big(\boldsymbol{p}_{x \rightsquigarrow u}, \boldsymbol{d^2}\big) \le D^1_{uv}$. Also, $\boldsymbol{d^1}$ is a D-wide partition on $\boldsymbol{T}$, so by Lemma 2,

$$D^1_{uv} + delay\big(\boldsymbol{p}_{u \rightsquigarrow y}, \boldsymbol{d^1}\big) \le D$$

and therefore

$$delay\big(\boldsymbol{p}_{x \rightsquigarrow u}, \boldsymbol{d^2}\big) + delay\big(\boldsymbol{p}_{u \rightsquigarrow y}, \boldsymbol{d^1}\big) \le D.$$

Since in this case $\boldsymbol{p}_{x \rightsquigarrow y} = \boldsymbol{p}_{x \rightsquigarrow u} + \boldsymbol{p}_{u \rightsquigarrow y}$ we can write

$$\begin{aligned} delay\big(\boldsymbol{p}_{x \rightsquigarrow y}, \boldsymbol{d}\big) &= delay(\boldsymbol{p}_{x \rightsquigarrow u}, \boldsymbol{d}) + delay\big(\boldsymbol{p}_{u \rightsquigarrow y}, \boldsymbol{d}\big) \\ &= delay\big(\boldsymbol{p}_{x \rightsquigarrow u}, \boldsymbol{d^2}\big) + delay\big(\boldsymbol{p}_{u \rightsquigarrow y}, \boldsymbol{d^1}\big)\,. \end{aligned}$$

Combining all, we get $delay\big(\boldsymbol{p}_{x \rightsquigarrow y}, \boldsymbol{d}\big) \le D$.

In all three cases $delay\big(\boldsymbol{p}_{x \rightsquigarrow y}, \boldsymbol{d}\big) \le D$, thus $\boldsymbol{d}$ is a D-wide partition on $\boldsymbol{T}$. ■

We have shown that any D-wide partition is made of D-wide sub-partitions. This observation also applies to optimal partitions, as stated in Lemma 5.

Lemma 5: Let $\boldsymbol{d^*}$ be an optimal D-wide partition on $\boldsymbol{T}$ and let $\boldsymbol{T}^{uv}$ be any sub-tree. Then $\boldsymbol{d^*}$ is an optimal D-wide and D^*_{uv}-deep partition on $\boldsymbol{T}^{uv}$.

Proof: By definition, $\boldsymbol{d^*}$ is a D^*_{uv}-deep partition on $\boldsymbol{T}^{uv}$ and by Lemma 1, it is D-wide. By Lemma 4, if we replace the sub-partition on $\boldsymbol{T}^{uv}$ with any D-wide and D^*_{uv}-deep partition on $\boldsymbol{T}^{uv}$ then we would still get a D-wide partition on $\boldsymbol{T}$. If $\boldsymbol{d^*}$ is not an optimal D-wide and D^*_{uv} partition on $\boldsymbol{T}^{uv}$ then we can replace it with an optimal one. The resulting overall D-wide partition would be cheaper than $\boldsymbol{d^*}$ on $\boldsymbol{T}$ contradicting the optimality of $\boldsymbol{d^*}$ on $\boldsymbol{T}$. ■

The same claims hold for any sub-branches as follows:

Corollary 1: Let $\boldsymbol{d^*}$ be an optimal D-wide partition on $\boldsymbol{T}^{uv}$. Then, $\boldsymbol{d^*}$ is an optimal D-wide and $D^*_{\mathcal{B}_{uv}}$-deep partition on $\boldsymbol{T}^{\mathcal{B}_{uv}}$, where $D^*_{\mathcal{B}_{uv}} \equiv depth\Big(\boldsymbol{T}^{\mathcal{B}_{uv}}\Big)$.

Proof: The proofs of Lemma 4 and Lemma 5 apply (with slight modifications) to this case too. ■

B. The algorithm

The previous lemmas provide us with the building blocks of our algorithm. We solve Problem WPQ using dynamic programming, computing optimal partitions on sub-trees and combining them to an optimal partition on the whole tree. Algorithm WPQ (Fig. 2) finds the best D-wide partition for every possible depth. The work is done by Procedure FIND-COST-TAB which recursively computes the tables $\mathcal{T}, \mathcal{L}, \mathcal{R}, \mathcal{B}$ corresponding to $\boldsymbol{T}^{uv}, \boldsymbol{T}^{\mathcal{L}_{uv}}, \boldsymbol{T}^{\mathcal{R}_{uv}}$ and $\boldsymbol{T}^{\mathcal{B}_{uv}}$, respectively. The tables are all of size D which hold an entry for every every depth $0 \le d \le D$. The entry for d is the best cost achieved for a D-wide and d-deep allocation on the tree. Procedure FIND-COST-TAB recursively merges tables of sub-trees until it reaches the root of the tree.

```
WPQ ($\boldsymbol{T}^{uv}, \{c_l(d)\}_{l \in \boldsymbol{T}^{uv}}, D$):
1 $\mathcal{T} \leftarrow$FIND-COST-TAB($\boldsymbol{T}^{uv}, \{c_l(d)\}_{l \in \boldsymbol{T}^{uv}}, D$)
2 if $\mathcal{T}(D) = \infty$ return FAIL
3 (else) return $\mathcal{T}(D)$ and the corresponding partition.

Procedure
FIND-COST-TAB ($\boldsymbol{T}^{uv}, \{c_l(d)\}_{l \in \boldsymbol{T}^{uv}}, D$):
1 $\mathcal{L} \leftarrow$FIND-COST-TAB($\boldsymbol{T}^{\mathcal{L}_{uv}}, \{c_l(d)\}_{l \in \boldsymbol{T}^L}, D$)
2 $\mathcal{R} \leftarrow$FIND-COST-TAB($\boldsymbol{T}^{\mathcal{R}_{uv}}, \{c_l(d)\}_{l \in \boldsymbol{T}^R}, D$)
3 for $d = 0 \ldots \lfloor D/2 \rfloor$
4     $\mathcal{B}(d) \leftarrow \mathcal{L}(d) + \mathcal{R}(d)$
5 for $d = \lceil D/2 \rceil \ldots D$
6     $\mathcal{B}(d) \leftarrow \min\{(\mathcal{L}(d) + \mathcal{R}(D-d)), (\mathcal{L}(D-d) + \mathcal{R}(d))\}$
7 for $d = 0 \ldots D$
8     $\mathcal{T}(d) \leftarrow \min_{0 \le x \le d} c_{uv}(x) + \mathcal{B}(d-x)$
9 return $\mathcal{T}$
```

Fig. 2. Algorithm WPQ

The computation of the table for $\boldsymbol{T}^{\mathcal{B}_{uv}}$ ($\mathcal{B}$) is based on Corollary 1 and Lemma 3. By Corollary 1, the optimal allocation on $\boldsymbol{T}^{\mathcal{B}_{uv}}$ is made of optimal allocations on $\boldsymbol{T}^{\mathcal{L}_{uv}}$ and $\boldsymbol{T}^{\mathcal{R}_{uv}}$. By Lemma 3, the depth of those allocations must satisfy $D_{\mathcal{L}_{uv}} + D_{\mathcal{R}_{uv}} \le D$. The computation is divided into two separate regions. For all depths $0 \le d \le \lfloor D/2 \rfloor$, the requirement $D_{\mathcal{B}_{uv}} = \max\{D_{\mathcal{L}_{uv}} + D_{\mathcal{R}_{uv}}\} \le d$ is satisfied, since $D_{\mathcal{L}_{uv}} + D_{\mathcal{R}_{uv}} \le 2\lfloor D/2 \rfloor \le D$. Therefore, the cost of the optimal D-wide and d-deep partition on $\boldsymbol{T}^{\mathcal{B}_{uv}}$ is found by simply summing the cost of d-deep partitions on $\boldsymbol{T}^{\mathcal{L}_{uv}}$ and $\boldsymbol{T}^{\mathcal{R}_{uv}}$. This is done at Line 4 of Procedure FIND-COST-TAB by summing the corresponding entries from $\mathcal{L}$ and $\mathcal{R}$. On the other hand, for all depths $\lceil D/2 \rceil \le d \le D$, if we allocate d on one sub-tree then we must allocate $D - d$ on the other in order to satisfy $D_{\mathcal{L}_{uv}}+D_{\mathcal{R}_{uv}} \leq D$. Line 6 of Procedure FIND-COST-TAB chooses the least costly among these two options.

The computation of the table for $\boldsymbol{T}^{uv}$ ($\mathcal{T}$) is based on Corollary 1 and the fact that $D_{uv} = d_{uv} + D_{\mathcal{B}_{uv}}$. By Corollary 1, the optimal allocation on $\boldsymbol{T}^{uv}$ is made of an optimal allocations on $\boldsymbol{T}^{\mathcal{B}_{uv}}$. In order to find the best partition we must check all possible depth allocations between (u,v) and $\boldsymbol{T}^{\mathcal{B}_{uv}}$. This is done at Line 8 of Procedure FIND-COST-TAB. Observe that, so long as $D_{uv} \leq D$, the resulting partition is D-wide. To see this, we first note that the delay for all paths from u to any other node in $\boldsymbol{T}^{uv}$ is less than $D_{uv} \leq D$. Second, any path between other nodes of $\boldsymbol{T}^{uv}$ is entirely inside $\boldsymbol{T}^{\mathcal{B}_{uv}}$, therefore its delay is less than D because the partition on $\boldsymbol{T}^{\mathcal{B}_{uv}}$ is D-wide.

When the recursive call returns, all entries of $\mathcal{T}$ correspond to valid D-wide partitions. We must check the entry for a depth of D, which is the least restrictive, hence should have minimal cost. If any link requires a minimal delay allocation on it then we might have infinite cost on it. If no partition satisfies the minimal delay requirement of all links then we would have overall infinite cost and the algorithm returns FAIL. With each cost entry update of $\mathcal{T}$ at Line 8 of Procedure FIND-COST-TAB, we can also record the allocation to (u,v) that achieved the minimum. Thus, if the algorithm is successful it can return both the optimal partition and its cost.

Complexity Each iteration of Procedure FIND-COST-TAB requires $O(D^2)$ operations. Each link is processed once, so the overall time complexity is $O(nD^2)$.

Parallel implementation The algorithm can be implemented in a parallel fashion, computing $\mathcal{L}$ and $\mathcal{R}$ at the same time. Such an implementation would require $O(h_{\max}D^2)$ time, where $h_{\max}$ is the depth of the recursion. Observe, that $h_{\max}$ is bounded by the maximal number of hops in any path, *i.e.*, the width of the tree in terms of hops. For balanced trees, $h_{\max}$ is of order $O(\log n)$.

Distributed implementation The algorithm can also be easily modified to operate in a distributed fashion. Each subtree can compute is optimal partition and forward the result. Such a solution is obviously parallel and also has the advantage that the link cost functions do not have to be advertised.

Note We must call Algorithm WPQ with $\boldsymbol{T}^{uv} = \boldsymbol{T}$, however we can select any leaf (u,v) of the tree.

IV. APPROXIMATE SOLUTION

Algorithm WPQ is a *pseudo polynomial* solution, which depends on the value of D. The latter could be arbitrarily large, depending on the precision in which delays are specified. Hence, in this section we improve the complexity at the expense of finding an approximate solution. The techniques we use are similar to the ones used in [2] for the approximate solution to Problem DPQ. We therefore give only a brief description and omit the proofs.

We consider two types of approximations: *delay*-approximation and *cost*-approximation. An ϵ-delay optimal solution is a partition which is $(1+\epsilon)D$-wide with cost no greater than the cost of the optimal D-wide partition. An ϵ-cost optimal solution is a D-wide partition with cost no greater than a factor of $(1+\epsilon)$ from the cost of the optimal D-wide partition.

A. Delay Approximation

Delay approximation is useful when the application can tolerate a delay that slightly higher than the required end-to-end delay. In general, as shown in [7], delay approximation is considerably simpler than cost approximation.

The approximation is based on scaling and rounding. Rather than finding the optimal cost for every possible depth $0 \leq d \leq D$, we consider only delays that correspond to integer factors of some scaling factor S. Algorithm ϵD-WPQ (Fig. 3) is a Fully Polynomial Approximation Scheme (FPAS) which finds an ϵ-delay optimal solution to Problem WPQ using scaling. It is polynomial in n and $1/\epsilon$.

```
ϵD-WPQ (T^{uv}, {c_l(d)}_{l∈T^{uv}}, D, ϵ):
1 $h_{\max} \leftarrow$ width of $\boldsymbol{T}^{uv}$ in hops
2 $S \leftarrow \frac{D\epsilon}{h_{\max}}$
3 for each $l \in \boldsymbol{T}$
4     define $\tilde{c}_l(d) \equiv c_l(\lceil d * S \rceil)$
5 $\tilde{D} \leftarrow \lceil \frac{h_{\max}}{\epsilon} \rceil$
6 return WPQ$\Big(\boldsymbol{T}^{uv}, \{\tilde{c}_l(d)\}_{l \in \boldsymbol{T}^{uv}}, \tilde{D}\Big)$
```

Fig. 3. Algorithm ϵD-WPQ

A careful choice of S is required to ensure that the resulting solution is indeed ϵ-delay approximate and the complexity is low enough. It can be shown that the overall error due to this type of scaling is $h_{\max}S$,[4] since the delay error of any path is summed over all its links. Thus, $S = \frac{D\epsilon}{h_{\max}}$ ensures an ϵ-delay optimal solution. Each D factor in the complexity expression for Algorithm WPQ is reduced to $D/S = h_{\max}/\epsilon$. This result is summarized by Theorem 1.

Theorem 1: Algorithm ϵD-WPQ finds an ϵ-delay optimal solution to Problem WPQ in

$$O\left(n(\frac{h_{\max}}{\epsilon})^2\right)$$

time.

Note A distributed/parallel implementation requires

$$O\left(h_{\max}^3/\epsilon^2\right)$$

time.

B. Cost Approximation

Often, the end-to-end requirement is strict and delay approximation cannot be used. One might be tempted to slightly strengthen the original delay requirement and then use delay approximation. However, while this indeed leads to a feasible solution,[5] its cost might be arbitrarily higher than that of the optimal solution.

In order to facilitate cost approximation, we must first reverse the roles of cost and delay in Algorithm WPQ. Instead

[4]Recall that $h_{\max}$ denotes the width of the tree.

[5]So long as there exist a solution which satisfies the stricter requirement.

of storing tables that hold the best cost achieved for every delay, we will store tables that hold the best delay achieved for every cost. Then, the same scaling techniques that were use for delay-approximation can be applied.

1) Cost-based dynamic programming: Algorithm C-WPQ (Fig. 4) is a cost-based version of Algorithm WPQ. Each table holds, for every cost c, the minimal depth achievable with overall cost no greater than c and width no greater that D. The size of each table is U, which is an upper bound on the cost. We shall assume for now that U is known and differ finding U to Section IV-B.3. The algorithm returns FAIL(Line 1) if no feasible solution can be found with cost not greater than U. Otherwise, it searches the table $\mathcal{T}$ for the min-cost entry that achieves a feasible solution (Line 3).

```
C-WPQ ($\boldsymbol{T}^{uv}, \{c_l(d)\}_{l\in\boldsymbol{T}^{uv}}, D, U$):
1  $\mathcal{T} \leftarrow$FIND-DEPTH-TAB($\boldsymbol{T}^{uv}, \{c_l(d)\}_{l\in\boldsymbol{T}^{uv}}, D, U$)
2  if $\mathcal{T}(U) > D$ return FAIL
3  (else) return $\min\{c|\mathcal{T}(c) \leq D\}$ and the corresponding partition.

Procedure
FIND-DEPTH-TAB ($\boldsymbol{T}^{uv}, \{c_l(d)\}_{l\in\boldsymbol{T}^{uv}}, D, U$):
1  $\mathcal{L} \leftarrow$FIND-DEPTH-TAB($\boldsymbol{T}^{\mathcal{L}_{uv}}, \{c_l(d)\}_{l\in\boldsymbol{T}^L}, D, U$)
2  $\mathcal{R} \leftarrow$FIND-DEPTH-TAB($\boldsymbol{T}^{\mathcal{R}_{uv}}, \{c_l(d)\}_{l\in\boldsymbol{T}^R}, D, U$)
3  for $c = 0 \dots U$
4    for $x = 0 \dots c$
5      $\mathcal{Z}(x) \leftarrow \max\{\mathcal{L}(x), \mathcal{R}(c-x)\}$
6      if $\mathcal{L}(x) + \mathcal{R}(c-x) > D$ then
7        $\mathcal{Z}(x) \leftarrow \infty$
8    $\mathcal{B}(c) \leftarrow \min_{0\leq x\leq c} \mathcal{Z}(x)$
9  for $c = 0 \dots U$
10   $\mathcal{Z}(x) \leftarrow \min\{d|c_{uv}(d) \leq c\}$
11 for $c = 0 \dots U$
12   $\mathcal{T}(c) \leftarrow \min_{0\leq x\leq c} \mathcal{Z}(x) + \mathcal{B}(c-x)$
13 return $\mathcal{T}$
```

Fig. 4. Algorithm C-WPQ

Procedure FIND-DEPTH-TAB is similar to Procedure FIND-COST-TAB of Algorithm WPQ and recursively calls itself for each sub-tree. However, it computes "depth"-tables instead of "cost"-tables. Unlike delay, the cost can be arbitrarily divided between the sub-trees, therefore we must check all possible cost allocations. In order to compute a single entry $\mathcal{B}(c)$, we must check all possible cost divisions between $\boldsymbol{T}^{\mathcal{L}_{uv}}$ and $\boldsymbol{T}^{\mathcal{R}_{uv}}$, *i.e.*, all pairs $\mathcal{L}(x), \mathcal{R}(c-x)$. This is done at Line 1 of Procedure C-WPQ and stored in a temporary table entry $\mathcal{Z}(x)$. Each entry in $\mathcal{Z}$ corresponds to the overall depth of the pair and is simply the *maximal* depth of the two. The overall depth of an allocation is the *sum* of the depths. If it is greater than D then the cost allocations is unfeasible, hence its entry is removed (at Line 3) by assigning an infinite delay. After $\mathcal{Z}(x)$ is computed for all $0 \leq x \leq c$, the minimal entry in $\mathcal{Z}$ is the best depth that can be achieved with cost c. The computation and the update to $\mathcal{B}(c)$ are done at Line 8.

Procedure FIND-DEPTH-TAB computes all the entries of $\mathcal{B}$ and continues to find the optimal partition between (u, v) and $\boldsymbol{T}^{\mathcal{B}_{uv}}$. As before, we need to check all possible divisions of cost between (u, v) and it branches. However, we need the link "delay"-function $d_{uv}(c)$, which is the equivalent of $c_{uv}(d)$ used in Algorithm WPQ. We use the $\mathcal{Z}$ table to store $d_{uv}(c)$ for all $0 \leq c \leq U$ (Line 9). Each entry represents the minimal delay that induces a cost no greater than c. Note that the cost $c_{uv}(d)$ is a monotonic non-increasing function of the delay, hence a binary search can be used to find the minimum. Once $d_{uv}(c)$ is computed, we proceed to fill $\mathcal{T}$. The computation at Line 12 is similar to the one done for $\mathcal{B}$, but here we are guaranteed a width no greater than D so long as the depth is no greater than D (see the discussion of Algorithm WPQ).

Complexity If we employ a binary search at Line 9 of Procedure FIND-DEPTH-TAB then each entry can be computed in $O(\log D)$ steps and $O(U \log D)$ for the whole table. All other computations require $O(U^2)$. Thus, $O(U(\log D + U))$ time is required for each link and the overall time complexity is

$$O\left(nU(\log D + U)\right).$$

Note A distributed, parallel implementation would require

$$O\left(h_{\boldsymbol{p}}U(\log D + U)\right)$$

time.

2) Cost scaling: We can now apply the same scaling methods as for the delay-approximation in order to achieve an ϵ-cost approximation. Algorithm ϵC-WPQ finds an ϵ-cost optimal solution, given a lower bound L and an upper bound U on the optimal cost.

```
$\epsilon C$-WPQ ($\boldsymbol{T}^{uv}, \{c_l(d)\}_{l\in\boldsymbol{T}^{uv}}, D, L, U, \epsilon$):
1  $S \leftarrow \frac{L\epsilon}{n}$
2  for each $l \in \boldsymbol{T}$
3    define $\tilde{c}_l(d) \equiv \lfloor c_l(d)/S \rfloor$
4  $\tilde{U} \leftarrow \lceil U/S \rceil$
5  return C-WPQ$\Big(\boldsymbol{T}^{uv}, \{\tilde{c}_l(d)\}_{l\in\boldsymbol{T}^{uv}}, \tilde{D}, \tilde{U}\Big)$
```

Fig. 5. Algorithm ϵC-WPQ

Theorem 2 is the equivalent of Theorem 1 for the ϵ-cost approximation. Again, a careful choice of S is required to ensure that the resulting solution is indeed an ϵ-cost approximate and the complexity is low enough. In this case, the overall error due to scaling is nS, since the cost error is summed over all links of the tree. Thus, $S = \frac{L\epsilon}{n}$ ensures an ϵ-delay optimal solution. Here, each U factor in the complexity expression for Algorithm C-WPQ is reduced to $U/S = O(nU/(\epsilon L))$.

Theorem 2: Algorithm ϵC-WPQ finds an ϵ-cost optimal solution to Problem WPQ in

$$O\left(\frac{n^2U}{\epsilon L}\left(\log D + \frac{nU}{\epsilon L}\right)\right)$$

time.

3) Finding good bounds: Algorithm ϵC-WPQ and its complexity depend on good bounds on the optimal cost c^*. If we chose $U < c^*$ then we might not be able to find a feasible partition (in which case the algorithm would FAIL). If we choose $U \gg c^*$ then U/L would be large and the complexity would be too high. The same applies for the choice of L: if $L > c^*$ then the scaling error might be too large, *i.e.*, we cannot ensure an ϵ-cost optimal solution; if $L \ll c^*$ then U/L (and the complexity) may grow up to the order of c^*.

The problem of finding tight bounds L, U that satisfy $L \leq c^* \leq U$ and $U/L = O(1)$ was addressed in relation to the restricted shortest path problem [13], [14] and to Problem DPQ [2]. The details of these techniques are out of the scope of this work, however, for completeness, we present an outline of the solution.

The general idea is to perform a binary search for a tight upper bound. At each iteration, a possible bound is tested by some test procedure that indicates whether it is too small or too large. An accurate test procedure might require a large number of operations for each run. Two key methods allow a very efficient search. First, *approximate tests* are used. These tests have a "gray-zone" around the tested value, for which they do not provide any information, however they still enable narrowing in on the bound. The idea is to initially use a simple and coarse test to find valid bounds that satisfy $U/L < n$, and then use more accurate (and costly) tests to refine these bounds. The second key is the application of *geometric* (rather than arithmetic) mean while performing the binary search. Lemma 6 and 7 summarize these techniques.

Lemma 6: Valid bounds, $L \leq c^* \leq U$, that satisfy $U/L \leq n$, can be found in $O(n \log D \log(Dn))$ time.[6]

Lemma 7: Valid bounds that satisfy $U/L \leq n$, can be refined to tight bounds that satisfy $U/L = O(1)$, in $O\left(n^2 \log\log n(\log D + n)\right)$ time.

The test procedure that enables the result of Lemma 6 searches for the minimum cost required on the most costly link. Given a possible bound β, it checks in $O(n \log D)$ whether any feasible D-wide partition exist if we assume that the cost of any single link is no more than β. Selection in matrices with sorted columns [15] is used to search for such a feasible bound among the$O(Dn)$ possible values. The test procedure that enables the result of Lemma 7 is Algorithm ϵC-WPQ with $\epsilon = 1$. Using geometric mean in the binary search enable finding the tight bounds in $O(\log\log n)$ tests.

Combining Algorithm ϵC-WPQ with the efficient bound search techniques establishes Theorem 3.

Theorem 3: An ϵ-cost optimal solution to Problem WPQ can be found in

$$O\left(n\left((\frac{n}{\epsilon})^2 + \log^2 D\right)\right)$$

time.[7]

V. Special Classes of Cost Functions

So far, the only assumption on the link cost functions was that they are monotonic non-increasing with the delay. In practice, link cost functions are more restrictive. For instance, a link may not offer the full range of delays, but rather provide only a few SLAs. The cost functions in such a case would be *discrete* functions,[8] *i.e.*, defined only for specific delays. In this section we show that, by focusing on more specific classes of cost functions, more efficient solutions can be established. We discuss two cases of practical interest.

A. Discrete Cost Functions

Discrete cost functions are defined only for a (relatively) small set $Q = \{q_1, q_2, q_3, \ldots\}$ of link QoS requirements. Alternatively, we can view each cost function as a step function with steps at each value in Q. Even if we define the cost (as a step function) over the whole range of delays, any optimal allocation would use only values from Q. Indeed, there is no gain from allocating a delay $q_i < d < q_{i+1}$ on any link, because the cost of q_i is the same with better performance.

Let q be the maximal number of values on any link, then the binary search at Line 9 of Algorithm C-WPQ can be performed in $O(\log q)$. If we use Algorithm ϵC-WPQ with the efficient bound search then the overall complexity is $O\left(n\left((\frac{n}{\epsilon})^2 + \log^2 q\right)\right) = O\left(n(\frac{n}{\epsilon})^2\right)$, assuming $q < n/\epsilon$. This is an improvement over the general case whenever $\log D > n/\epsilon$.

B. An Example of "Homogeneous" Functions

Here we demonstrate that a more significant improvement can be achieved when all link cost functions are of the same form. Specifically, we investigate cost functions of the form $c_l(d) = A^l/d^\theta + C^l$, where A^l, θ, C^l are given constants. This function has many desirable practical characteristics. It decreases with the delay, as required; it is convex; it assigns infinitely high cost when the required delay guarantee approaches zero, and it assigns a fixed minimal link usage cost C^l, even if no guarantee is required. The constant θ determines how fast the cost grows for low delays and the constant A^l is used as a scaling constant.

In the following we establish a *linear*-time algorithm. We begin by observing that the fixed cost $C = \sum_{l \in \boldsymbol{T}} C^l$ is charged for any partition, so we can just assume add it later and assume $C^l = 0$ for all $l \in \boldsymbol{T}$. We will also assume, for now, that $\theta = 1$ and shall relax this assumption later.

We first establish some properties of the optimal tree cost. We can view each table $\mathcal{T}$ computed by Algorithm WPQ as the cost function of the sub-tree $\boldsymbol{T}^{uv}$. It provides the cost of the optimal partition as a function of the depth of the sub-tree, under the assumption that it is D-wide. We call this the tree cost function and denote it by $c^{\mathcal{T}^{uv}}(d)$ or $c^{\mathcal{T}}$. Similarly, we define $c^{\mathcal{L}}, c^{\mathcal{R}}$, and $c^{\mathcal{B}}$ as the cost functions for the sub-trees $\boldsymbol{T}^{\mathcal{L}_{uv}}, \boldsymbol{T}^{\mathcal{R}_{uv}}$, and $\boldsymbol{T}^{\mathcal{B}_{uv}}$. We want to be able to compute $c^{\mathcal{T}^{uv}}(d)$ analytically, without having to compute it for every d, as is done by Algorithm WPQ. Lemma 8 provides us with a method to do so.

Lemma 8: If each link cost functions is of the form A^l/d then for any sub-tree,

$$c^{\mathcal{T}^{uv}}(d) = A^{\mathcal{T}^{uv}}/d \quad \forall d \leq D/2,$$

[6]Alternatively, such bounds can be found in $O(n \log D \log\log \beta_0)$, where β_0 is the ratio of some coarse (possibly trivial) initial bounds.

[7]Assuming $1/\epsilon > \log\log n$.

[8]We follow here the terminology of [6].

where $A^{\mathcal{T}^{uv}}$ is a scaling constant for the sub-tree. Furthermore, there exist similar scaling constants $A^{\mathcal{L}}, A^{\mathcal{R}}, A^{\mathcal{B}}$ for $\boldsymbol{T}^{\mathcal{L}_{uv}}, \boldsymbol{T}^{\mathcal{R}_{uv}}$, and $\boldsymbol{T}^{\mathcal{B}_{uv}}$.

Proof: The proof is by induction. Obviously, the claim holds if $\boldsymbol{T}^{uv}$ is a single link.

We first show that if the claim holds for $\boldsymbol{T}^{\mathcal{L}_{uv}}$ and $\boldsymbol{T}^{\mathcal{R}_{uv}}$ then it is also true for $\boldsymbol{T}^{\mathcal{B}_{uv}}$. By the results of Section III, we have for $d \leq D/2$

$$c^{\mathcal{B}}(d) = c^{\mathcal{L}}(d) + c^{\mathcal{R}}(d).$$

By the assumption $c^{\mathcal{L}}(d) = A^{\mathcal{L}}/d$ and $c^{\mathcal{R}}(d) = A^{\mathcal{R}}/d$, hence

$$c^{\mathcal{B}}(d) = A^{\mathcal{L}}/d + A^{\mathcal{R}}/d = (A^{\mathcal{L}} + A^{\mathcal{R}})/d,$$

namely $c^{\mathcal{B}}(d) = A^{\mathcal{B}}/d$, for

$$A^{\mathcal{B}} = A^{\mathcal{L}} + A^{\mathcal{R}}.$$

Next we show that if the claim holds for $\boldsymbol{T}^{\mathcal{B}_{uv}}$ then it is also true for $\boldsymbol{T}^{uv}$. Again, from Section III,

$$c^{\mathcal{T}}(d) = \min_{0 \leq x \leq d} c^{\mathcal{B}}(x) + c_{uv}(d-x).$$

Applying the assumption, we have

$$c^{\mathcal{T}}(d) = \min_{0 \leq x \leq d} A^{\mathcal{B}}/x + A^{uv}/(d-x).$$

It is easy to verify that the minimization yields a cost of $c^{\mathcal{T}}(d) = (\sqrt{A^{\mathcal{B}}} + \sqrt{A^{uv}})^2/d$ at

$$x = \frac{\sqrt{A^{\mathcal{B}}}}{\sqrt{A^{\mathcal{B}}} + \sqrt{A^{uv}}} d. \qquad (1)$$

That is, $c^{\mathcal{T}}(d) = A^{\mathcal{T}}/d$, where

$$\sqrt{A^{\mathcal{T}}} = \sqrt{A^{\mathcal{B}}} + \sqrt{A^{uv}}.$$

We can build $c^{\mathcal{T}}(d) = A^{\mathcal{T}}/d$ recursively, starting from the leafs. We have shown that the assumption is preserved throughout this build procedure. Thus, we have established the claim with

$$A^{\mathcal{B}} = A^{\mathcal{L}} + A^{\mathcal{R}} \text{ and } \sqrt{A^{\mathcal{T}}} = \sqrt{A^{\mathcal{B}}} + \sqrt{A^{uv}}.$$

■

Lemma 8 states that this type of cost function is closed under the tree building computations, for all depth no greater than $D/2$. Using this result, Procedure BUILD-A builds $A^{\mathcal{T}^{uv}}$ recursively in linear ($O(n)$) time.

Algorithm COMPUTE-A computes $A^{\mathcal{T}^{uv}}$ for all sub-trees $\boldsymbol{T}^{uv} \in \boldsymbol{T}$. Note that, $A^{\mathcal{T}^{uv}} \neq A^{\mathcal{T}^{vu}}$, therefore it is not enough to run Procedure BUILD-A for a single root link. However, after a single call to Procedure BUILD-A we have $A^{\mathcal{T}^{uv}}$ in one direction for all links of $\boldsymbol{T}$. The other direction is computed by starting from the root link and visiting all nodes in pre-order. The traversal maintains the property that after a node w is visited then $A^{\mathcal{T}^{wx}}$ is known for every link $(w,x) \in N(w)$. This property trivially holds for v, since only $A^{\mathcal{T}^{vu}}$ is not computed by the initial call to Procedure BUILD-A and $A^{\mathcal{T}^{vu}} = A^{vu}$. Throughout the traversal, we need to compute $A^{\mathcal{T}^{wx}}$ only for the parent $x = parent(w)$ of w, since all other values are

```
COMPUTE-A ($\boldsymbol{T}, \{c_l(d)\}_{l \in \boldsymbol{T}}$):
1  Select any root link $(u,v) \in \boldsymbol{T}$
2  $A^{\mathcal{T}^{uv}} \leftarrow$BUILD-A($\boldsymbol{T}^{uv}$)
3  Starting from $v$, visit all nodes $w \in \boldsymbol{T}^{uv}$ in pre-order order
4      $x \leftarrow parent(w)$
5      $A^{\mathcal{T}^{wx}} \leftarrow$BUILD-A($\boldsymbol{T}^{wx}$)

Procedure
BUILD-A ($\boldsymbol{T}^{uv}, \{c_l(d)\}_{l \in \boldsymbol{T}^{uv}}$):
1  if $\boldsymbol{T}^{uv}$ is a link then
2      $A^{\mathcal{T}^{uv}} \leftarrow A^{uv}$
3      return
4  (else) if $A^{\mathcal{T}^{uv}}$ is already known then
5      return
6  (else) call BUILD-A($\boldsymbol{T}^{\mathcal{L}_{uv}}, \{c_l(d)\}_{l \in \boldsymbol{T}^{\mathcal{L}_{uv}}}$)
7  call BUILD-A($\boldsymbol{T}^{\mathcal{R}_{uv}}, \{c_l(d)\}_{l \in \boldsymbol{T}^{\mathcal{R}_{uv}}}$)
8  $A^{\mathcal{B}} \leftarrow A^{\mathcal{L}} + A^{\mathcal{R}}$
9  $A^{\mathcal{T}^{uv}} \leftarrow (\sqrt{A^{\mathcal{B}}} + \sqrt{A^{uv}})^2$
10 return
```

Fig. 6. Algorithm COMPUTE-A

already known after the initial call to Procedure COMPUTE-A. Furthermore, the call to Procedure COMPUTE-A requires $O(1)$ operations, since a recursive call to BUILD-A($\boldsymbol{T}^{wx}$) terminates immediately. This is because x was already visited, hence $A^{\mathcal{T}^{xy}}$ is already known for all $(x,y) \in N(x)$. Thus, overall only $O(n)$ additional operations are performed after the initial call to Procedure COMPUTE-A. Theorem 4 summarizes these results.

Theorem 4: Algorithm COMPUTE-A computes $A^{\mathcal{T}^{uv}}$ for all sub-tress $\boldsymbol{T}^{uv} \in \boldsymbol{T}$ in $O(n)$ time.

We have established $c^{\mathcal{T}^{uv}}(d)$ for all sub-trees and for all $d \leq D/2$. Next we show how we can avoid having to compute $c^{\mathcal{T}^{uv}}(d)$ for larger values of d. We need to find a location in the tree that is no further than $D/2$ from any node. We call this the *center* of the optimal partition and define it formally as follows. For a given partition, we call u a *center-node* if $D_{uv} \leq D/2$ for all $(u,v) \in N(u)$; we call a link (u,v) a *center-link* if $D_{uw} \leq D/2$ for all $(u,w) \in N(u)$ and $D_{vw} < D/2$ for all $(v,w) \in N(v)$. The *center* of a partition is either a center-node or a center-link. Next, we establish some properties of the center.

Lemma 9: If u is a center-node of an optimal D-wide partition and all costs are strictly monotonic then $D_{uv} = D/2$ for all $(u,v) \in N(u)$.

Proof: Let (u,v) be a link for which $D_{uv} < D/2$. Since u is a center node, both $D_{\mathcal{L}_{vu}}$ and $D_{\mathcal{R}_{vu}}$ are less than $D/2$, hence so is $D_{\mathcal{B}_{vu}}$. This implies that the delay allocation on $\boldsymbol{T}^{uv}$ can be increased to $D/2$ without violating the width constraint. Since all costs are strictly monotonic, an increase in the delay allocation must reduce the cost, contradicting the optimality of the partition. ■

Each cost functions A^l/d is strictly monotonic so Lemma 9 can be applied.[9] Lemma 9 implies that we can have either a center-link or a center-node, but not both. The strict monotony

[9]Even if the cost functions are not strictly monotonic, the proof of Lemma 9 implies that there exists an optimal partition for which u is the center node and the equality holds.

implies that the minimal allocation on any link must be greater than zero, and there exists at least one path through every link for which $delay(\boldsymbol{p}) = D$. Otherwise, the allocation to the link can be increased, reducing the cost without violating the width assumption. It also implies that

$$D_{uv} > D_{\mathcal{B}_{uv}}. \tag{2}$$

Lemma 10: In any optimal D-wide partition there exists a unique center.

Proof: Lemma 9, together with Equation 2, implies that we cannot have more than one center-node. Equation 2 also implies that we can not have more than one center-link. Thus if either a center-node or a center-link exists then it is unique. We prove their existence by showing how a center can be found.

First, we replace each link (u, v) with two directed links $(u, v), (v, u)$. Then, we color each link green if $D_{uv} > D/2$ and red otherwise. If all outgoing links from a node u are red then u is a center-node. If both (u, v) and (v, u) are green then (u, v) is a center-link. We start from any leaf and traverse over green links only. For any leaf node u the directed link (u, v) must be colored green, since at least one path through (u, v) must have a delay of D. By Lemma 3, there is at most one green outgoing link from any node so the traversal is well defined. If we cannot continue the traversal then all outgoing links are red and we have reached a center-node. Otherwise, we must eventually enter a simple cycle. Since we started from a tree, the directed cycle must correspond to a single bi-direction link, which is a center-link. ∎

Lemma 11: Let (u, v) be a center-link of an optimal D-wide partition. Then, $\sqrt{A^{\mathcal{B}_{vu}}} < \sqrt{A^{\mathcal{B}_{uv}}} + \sqrt{A^{uv}}$.

Proof: Since (v, u) is a center-link $D_{\mathcal{B}_{vu}} < D/2$. Consider the allocation we get by moving $\delta \leq D - D_{\mathcal{B}_{vu}}$ delay from the allocation of (u, v) to increase the depth of $\boldsymbol{T}^{\mathcal{B}_{vu}}$. It is easy to verify that the allocation we get still satisfies the widths constraint. Optimality implies that there is no gain from such a move. Since this is true for any δ and both $c^{\mathcal{B}_{vu}}$ and c_{vu} are differentiable, we must have $|c'_{\mathcal{B}_{vu}}(D_{\mathcal{B}_{vu}})| \leq |c'_{uv}(d_{uv})|$. Hence,

$$\frac{A^{\mathcal{B}_{vu}}}{(D_{\mathcal{B}_{vu}})^2} \leq \frac{A^{uv}}{(d_{uv})^2},$$

implying

$$\frac{d_{uv}}{D_{\mathcal{B}_{vu}}} \leq \frac{\sqrt{A^{uv}}}{\sqrt{A^{\mathcal{B}_{vu}}}}.$$

For similar reasons,

$$\frac{D_{\mathcal{B}_{uv}}}{D_{\mathcal{B}_{vu}}} \leq \frac{\sqrt{A^{\mathcal{B}_{uv}}}}{\sqrt{A^{\mathcal{B}_{vu}}}}.$$

Summing both we get

$$\frac{D_{uv}}{D_{\mathcal{B}_{vu}}} = \frac{d_{uv} + D_{\mathcal{B}_{uv}}}{D_{\mathcal{B}_{vu}}} \leq \frac{\sqrt{A^{uv}} + \sqrt{A^{\mathcal{B}_{uv}}}}{\sqrt{A^{\mathcal{B}_{vu}}}}.$$

The lemma follows from $D_{uv} > D/2 > D_{\mathcal{B}_{vu}}$, which implies that $1 < D_{uv}/D_{\mathcal{B}_{vu}}$. ∎

Lemma 12: Let u be a center-link of an optimal D-wide partition. Then, for any $(u, v) \in N(u)$, $A^{\mathcal{T}^{uv}} \leq A^{\mathcal{B}_{uv}}$.

Proof: In this case $D_{uv} = D_{\mathcal{B}_{uv}} + d_{uv} = D/2$, hence $D_{\mathcal{B}_{uv}} < D/2$, and $d_{uv} < D/2$. Following the proof of Lemma 11, it is feasible to move δ from $\boldsymbol{T}^{\mathcal{B}_{vu}}$ to either (u, v) or $\boldsymbol{T}^{\mathcal{B}_{uv}}$. Since there must be no gain from such a move, this implies $|c'_{\mathcal{B}_{vu}}(D_{\mathcal{B}_{vu}})| \geq |c'_{uv}(d_{uv})|$ or

$$\frac{d_{uv}}{D_{\mathcal{B}_{vu}}} \geq \frac{\sqrt{A^{uv}}}{\sqrt{A^{\mathcal{B}_{vu}}}}.$$

We also have

$$\frac{D_{\mathcal{B}_{uv}}}{D_{\mathcal{B}_{vu}}} \geq \frac{\sqrt{A^{\mathcal{B}_{uv}}}}{\sqrt{A^{\mathcal{B}_{vu}}}}.$$

Summing, as before, we get

$$1 = \frac{D_{uv}}{D_{\mathcal{B}_{vu}}} = \frac{d_{uv} + D_{\mathcal{B}_{uv}}}{D_{\mathcal{B}_{vu}}} \geq \frac{\sqrt{A^{uv}} + \sqrt{A^{\mathcal{B}_{uv}}}}{\sqrt{A^{\mathcal{B}_{vu}}}} = \frac{\sqrt{A^{\mathcal{T}^{uv}}}}{\sqrt{A^{\mathcal{B}_{vu}}}},$$

hence, $\frac{A^{\mathcal{T}^{uv}}}{A^{\mathcal{B}_{vu}}} \leq 1$, and the lemma follows. ∎

After we run Algorithm COMPUTE-A, Lemma 11 allows us to test whether a link (u, v) can be a center link by comparing $\sqrt{A^{\mathcal{B}_{vu}}}$ to $\sqrt{A^{\mathcal{B}_{uv}}} + \sqrt{A^{uv}}$. Note that, $\sqrt{A^{\mathcal{B}_{uv}}} + \sqrt{A^{uv}} = \sqrt{A^{\mathcal{T}^{uv}}}$, therefore we can simply compare $A^{\mathcal{B}_{vu}}$ and $A^{\mathcal{T}^{uv}}$. If $A^{\mathcal{B}_{vu}} \geq A^{\mathcal{T}^{uv}}$ then (u, v) is not a center-link. Lemma 12 provides a similar test for a center-node. If $A^{\mathcal{T}^{uv}} > A^{\mathcal{B}_{vu}}$ then u is not a center-node.

Since $A^{\mathcal{B}_{vu}} = A^{\mathcal{L}_{vu}} + A^{\mathcal{R}_{vu}}$, from symmetry, there is at most one link $(u, v) \in N(u)$ for which $A^{\mathcal{T}^{uv}} > A^{\mathcal{B}_{vu}}$. Note that, this test *cannot* fail for any leaf $u \in \boldsymbol{T}$. In order to find the center we repeat the method in the proof of Lemma 10.

We replace each link (u, v) with two directed links (u, v) and (v, u). Then, we color each link green if $A^{\mathcal{T}^{uv}} > A^{\mathcal{B}_{vu}}$ and red otherwise. If all outgoing links from a node u are red then u is a feasible center-node. If both (u, v) and (v, u) are green then (u, v) is a feasible center-link. All conditions of that proof apply, therefore a feasible center can be found. The time complexity is $O(n)$ since at most n links can be traversed before a cycle is entered.

The actual partition on each sub-tree can be computed using Equation 1. All the above results, with slight modifications apply even if $\theta \neq 1$. The modifications are straightforward and therefore omitted. We summarize our results in Theorem 5.

Theorem 5: Given link cost functions of the form $c_l(d) = A^l/d^\theta + C^l$ for all $l \in \boldsymbol{T}$, an optimal solution to Problem WPQ can be found in $O(n)$.

VI. DISCUSSION

We have dealt with the many-to-many QoS partition problem on trees in context of performance-dependent costs [1], [2], [6], [12].

The relation between performance and cost is an important aspect of QoS provisioning in a commercial environment. Providing QoS guarantees requires resource allocations on each network element and thus is associated with direct and indirect costs. Finding the least cost allocation which satisfies the connection's QoS requirements is a fundamental network optimization problem. The performance-dependent cost framework considered in this paper is a simple yet powerful model that incorporates the *Cost-QoS* tradeoffs. It is an extension of the well known bi-criteria model in which each link is associated with a single QoS guarantee and a single cost. Instead of a single level of service, each link may offer multiple levels of service or, generally, a complete *Cost-QoS* function. This framework is more descriptive of the practical tradeoffs between the strictness of the QoS requirement and the cost of their guarantee.

We have continued the work presented in [2], which dealt with one-to-one and one-to-many connections. The present study has established the first solution to the many-to-many case, which arises in the context of QoS-supported group communication sessions, such as tele-conferencing and virtual classes. In such applications, there is an end-to-end QoS requirement between *any* two members of the session. It turns out that the many-to-many problem, with additive (delay) QoS bounds, is inherently much more complex than the one-to-many multicast tree case. Thus, a considerable amount of work was needed in order to develop efficient algorithms for this case.

We established the structure of optimal solutions to this problem and presented an optimal pseudo-polynomial solution. From this exact solution, we derived two types of efficient (fully polynomial) approximations: delay-approximation and cost-approximation. The approximations are within $(1+\epsilon)$ of the optimal solution (in either delay or cost). We showed improved results for the practical case of discrete cost functions. We further studied a special case of convex cost functions which are of practical interest. For this case, we developed a remarkably efficient algorithm that achieves an *exact* solution to the many-to-many problem in linear time.

In this study, we have assumed the existence of a single multicast tree for a session. While some proposed multicast protocols construct more complex topologies, such as mesh and reduced-mesh, most of the basic multicast protocols, such as CBT or PIM, typically maintain a single tree for each session. Extending the results of this study in order to make them apply to more complex or failure-resilient topologies would require a considerable amount of work, since the theoretical properties of the problems change significantly. Nonetheless, we believe that the results presented here, regarding trees, provide significant insight and some of the required basics for such extensions.

An important finding of this study is that some generic functions allow for more efficient solutions than even the discrete model, which is the current basis for QoS provisioning (e.g., [11]). Our results show that a more expressive multivalued cost function framework is not only feasible, but practical. We believe that efficient linear solutions can be established for other cases of special practical interest, beyond those investigated in this paper. These findings should be taken into consideration when defining future QoS pricing schemes.

References

[1] F. Ergün, R. Sinha, and L. Zhang, "QoS routing with performance-dependent costs," in *Proceedings of the IEEE INFOCOM'2000*, Tel-Aviv, Israel, Mar. 2000.

[2] D. H. Lorenz, A. Orda, D. Raz, and Y. Shavitt, "Efficient QoS partition and routing of unicast and multicast," in *In Proceeding of The 8th International Workshop on Quality of Service (IWQoS 2000)*, Pittsburgh, PA, June 2000, pp. 75–83.

[3] V. Firoiu and D. Towsley, "Call admission and resource reservation for multicast sessions," in *Proceedings of the IEEE INFOCOM'96*, San Francisco, CA, Apr. 1996, pp. 776–785.

[4] M. Kodialam and S. H. Low, "Resource allocation in a multicast tree," in *Proceedings of the IEEE INFOCOM'99*, New York, NY, Mar. 1999, pp. 262–266.

[5] D. H. Lorenz and A. Orda, "Optimal partition of QoS requirements on unicast paths and multicast trees," *IEEE/ACM Transactions on Networking*, pp. 102–114, Feb. 2002.

[6] D. Raz and Y. Shavitt, "Optimal partition of QoS requirements with discrete cost functions," *IEEE Journal on Selected Areas in Communications*, vol. 18, no. 12, pp. 2593–2602, Dec. 2000.

[7] A. Goel, K. Ramakrishnan, D. Kataria, and D. Logothetis, "Efficient computation of delay-sensitive routes from one source to all destinations," in *Proceedings of the IEEE INFOCOM'01*, Anchorage, Alaska, Apr. 2001.

[8] R. Nagarajan, J. Kurose, and D. Towsley, "Allocation of local quality of service constraints to meet end-to-end requirements," in *IFIP Workshop on the Performance Analysis of ATM Systems*, Martinique, Jan. 1993.

[9] A. Orda and A. Sprintson, "A scalable approach to the partition of QoS requirements in unicast and multicast," in *Proceedings of the IEEE INFOCOM'02*, New York, NY, June 2002, pp. 685–694.

[10] S. Chen and K. Nahrstedt, "An overview of quality-of-service routing for the next generation high-speed networks: Problems and solutions," *IEEE Network Magazine*, vol. 12, no. 6, Nov./Dec. 1998.

[11] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, "An architecture for differentiated services," Internet RFC 2475, Nov. 1998.

[12] D. H. Lorenz and A. Orda, "QoS routing in networks with uncertain parameters," *IEEE/ACM Transactions on Networking*, vol. 6, no. 6, pp. 768–778, Dec. 1998.

[13] R. Hassin, "Approximation schemes for the restricted shortest path problem," *Mathematics of Operations Research*, vol. 17, no. 1, pp. 36–42, Feb. 1992.

[14] D. H. Lorenz and D. Raz, "A simple efficient approximation scheme for the restricted shortest path problem," *Operations Research Letters*, vol. 28, no. 5, pp. 213–219, June 2001.

[15] G. N. Frederickson and D. B. Johnson, "The complexity of selection and ranking in $X+Y$ and matrices with sorted columns," *Journal of Computer and System Sciences*, vol. 24, no. 2, 1982.