Abstract

SALAMA, HUSSEIN FAROUK. Multicast Routing for Real-Time Communication on High-Speed Networks. (Under the direction of Yannis Viniotis and Douglas S. Reeves)

Real-time network applications, e.g., multimedia applications and critical control applications, are evolving at a fast pace. These applications are resource-intensive, have stringent delay requirements, and in many cases involve more than two participants. Efficient multicast communication mechanisms are, therefore, necessary to support real-time applications. In this dissertation, we study five routing problems for real-time communication on high-speed connection-oriented wide-area networks. The objective of all problems we study is to optimize the utilization of the network resources without violating the delay requirements of real-time applications. We focus primarily on multicast routing problems, but we consider also the special cases of broadcast routing and unicast routing.

MULTICAST ROUTING FOR REAL-TIME COMMUNICATION ON HIGH-SPEED NETWORKS

BY
HUSSEIN FAROUK SALAMA

A DISSERTION SUBMITTED TO THE GRADUATE FACULTY OF
NORTH CAROLINA STATE UNIVERSITY
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

RALEIGH, NC
1996

APPROVED BY:

A.A. NILSSON	M.F.M. STALLMANN
Y. VINIOTIS	D.S. REEVES
CO-CHAIR OF ADVISORY COMMITTEE	CO-CHAIR OF ADVISORY COMMITTEE

To my mother, Fakhriya Taha,
my father, Farouk Salama,
and
my grandmother, Bothaina Mohi El Din.

Biography

Hussein Salama was born in Cairo, Egypt, on October 18, 1967. He received his primary, preparatory, and secondary education at the German School in Cairo. In 1985, he received the Diploma of the German Language and the Egyptian High School Diploma with a grade of 94.25%. He then joined the Faculty of Engineering at Cairo University where he majored in Electronics and Communications Engineering. During the summers of 1986, 1987, and 1989, he worked at companies in Germany and Switzerland. In July 1990, he earned the B.Sc. degree with honors from Cairo University and was ranked fourth out of 400 students in his graduating class. During the following year, Hussein held part-time positions as a Teaching Assistant at the same university and as a Design Engineer at the Research and Electronic Manufacturing Center (REEM) in Cairo.

In August 1991, Hussein joined the Department of Electrical and Computer Engineering at North Carolina State University (NCSU). In May 1993, he obtained the M.Sc. degree in Electrical Engineering, and started working towards his Ph.D. in Computer Engineering. During his graduate studies at NCSU, Hussein held several Teaching Assistant and Research Assistant positions. His Ph.D. research was sponsored by the Center for Advanced Computing and Communication (CACC) at NCSU and by IBM.

During the summer of 1995, Hussein worked at Alcatel Network Systems in Raleigh. Following that, he spent two months as a Visiting Researcher at the Center for Open Communication Systems (GMD-FOKUS) in Berlin, Germany. He is a student member of IEEE and a member of the Honor Society of Phi Kappa Phi.

List of Publications

- H.F. Salama, D.S. Reeves, Y. Viniotis, "Delay-Constrained Shared Multicast Trees," submitted to the Seventh IFIP Conference on High Performance Networking (HPN'97), White Plains, New York, April 1997.
- H.F. Salama, D.S. Reeves, Y. Viniotis, "The Delay-Constrained Minimum Spanning Tree Problem," submitted to the Second IEEE Symposium on Computers and Communications (ISCC'97), Alexandria, Egypt, July 1997.
- H.F. Salama, D.S. Reeves, Y. Viniotis, "Evaluation of Multicast Routing Algorithms for Real-Time Communication on High-Speed Networks," accepted for publication in the IEEE Journal on Selected Areas in Communications.
- H.F. Salama, D.S. Reeves, and Y. Viniotis, "A Distributed Algorithm for Delay-Constrained Unicast Routing," accepted for IEEE INFOCOM'97, Kobe, Japan, April 1997.
- H.F. Salama, D.S. Reeves, and Y. Viniotis, "An Efficient Delay-Constrained Minimum Spanning Tree Heuristic," poster paper, in Proceedings of the Fifth International Conference on Computer Communication and Networking (IC^{3}N'96), Washington, DC, October 1996.
- S. Damaskos and H.F. Salama, "Reservation Mechanisms for Efficient Resource Management in Internetworks," in Proceedings of the Third IEEE International Conference on Multimedia Computing and Systems (ICMCS'96), Hiroshima, Japan, June 1996.
- H.F. Salama, D.S. Reeves, Y. Viniotis, and T.-L. Sheu, "Evaluation of Multicast Routing Algorithms for Real-Time Communication on High-Speed Networks," in Proceedings of the Sixth IFIP Conference on High Performance Networking (HPN'95), Palma De Mallorca, Spain, September 1995.

Acknowledgments

This dissertation would have never been completed without the will and blessing of God, the most gracious, the most merciful. AL HAMDU LELLAH.

My deepest admiration and thanks to my co-advisors, Dr. Douglas Reeves and Dr. Yannis Viniotis, for their support and guidance. I was fortunate to have them as my advisors. Dr. Reeves was never too busy to listen to me and offer his advice. Our long discussions have benefited me immensely on both the research and personal levels. Dr. Viniotis' expert counseling and suggestions at critical stages of my work are deeply appreciated.

I would also like to acknowledge my other advisory committee members, Dr. Arne Nilsson and Dr. Matt Stallmann. Dr. Nilsson's computer networks course sparked my interest in networking research. Dr. Stallmann provided valuable input on the algorithmic aspects of my research problems.

I can not find words strong enough to express my gratitude to my family. My mother Dr. Fakhriya Taha. My Father Dr. Farouk Salama. My grandmother Bothaina Mohi El Din. My sister Bothaina, her husband Omar Tawakol, and their daughter Ameena. My sister Mennatullah. Their continuous support and supplication were crucial for the success of my studies. Thank you. I also wish to extend my thanks to my aunts and uncles.

Last, but not least, I would like to thank my friends. I would like to thank my childhood friends from Cairo, Egypt. The emails and phone calls we exchanged, cheered me up at the most difficult stages of my research. I would also like to thank the friends I made in Raleigh, North Carolina, and my officemates and labmates. They added a lot of excitement and fun to my life, and I learned a lot from all of them.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Recent advances in optical fiber and switch technologies have resulted in a new generation of high-speed networks that can achieve speeds of up to a few gigabits per second, along with very low bit error rates. In addition, the progress in audio, video, and data storage technologies has given rise to new distributed real-time applications. These applications may involve multimedia, e.g., network-based education and videoconferencing, which require low end-to-end delays, or they may be distributed control applications requiring high transmission reliability. The applications' requirements, such as the end-to-end delay, delay jitter, and loss rate, are expressed as Quality of Service (QoS) parameters which must be guaranteed by the underlying network. In addition, many of these new applications may involve multiple users, and hence the importance of multicast communication.

Datagram networks can not provide QoS guarantees to real-time applications. Circuit-switched networks are capable of providing guaranteed service to the applications, but they do not manage the network bandwidth efficiently. The resulting performance is particularly poor for networks carrying bandwidth-extensive variable bit rate traffic streams, e.g., video or high-quality audio streams. Fortunately, Broadband Integrated Services Digital Networks (B-ISDN) [1] are evolving at a fast pace. B-ISDNs use the Asynchronous Transfer Mode (ATM) [1] which is based on high-speed packet-switching technology. ATM is a connection-oriented technique in the sense that an application first has to set up virtual connections between the sources and the receivers[1] prior to the actual transmission of packets (denoted as cells in

[1]The terms "source" and "sender" are used interchangeably in this dissertation. Similarly the

ATM terminology). While setting up the virtual connections, the application specifies the QoS requirements. The paths for the connections are selected and resources are allocated, based on the requested QoS and the available network resources. Thus ATM is capable of providing guaranteed service to the applications. ATM is designed to allow different applications with different QoS requirements to coexist and share the same underlying network. Statistical multiplexing can be applied in ATM networks, because of its packet-switched nature. In addition, ATM specifies multicast as part of its service. These two features enable ATM to efficiently manage the network bandwidth.

However, the ATM layer itself only provides the means for fast end-to-end transmission of packets. Efficient network layer mechanisms are needed in order to benefit from the services provided by ATM. Currently, much work is underway to develop such network layer mechanisms which include routing, resource reservation, admission control, flow control, and network security.

In the past, the routing problem in communication networks was simpler. The applications utilized a modest percentage of the available bandwidth and none of them had QoS requirements. In addition, very few applications involved more than two users. That is why simple routing techniques were sufficient in the past. The situation is different, however, for the emerging real-time applications discussed above. These applications are usually bandwidth-intensive, have QoS requirements, involve more than two users, and they are already available over current networks. For example, videoconferencing is already available over the Internet [2]. Thus the routing problem for real-time applications is more complex than the routing problem of the past. There are many variations of the routing problem depending on the architecture and size of the network and the QoS requirements of the applications. This dissertation focuses primarily on routing mechanisms for multicast communication over high-speed, connection-oriented, wide-area networks carrying real-time traffic with QoS requirements.

---

terms “receiver” and “destination” are also used interchangeably.

1.1 Multicast Communication

Multicasting is the capability of delivering a packet to multiple receivers such that exactly one copy of the packet traverses each link in the delivery tree. There are numerous examples of multicast applications. Interactive multicast applications include videoconferencing, computer-supported cooperative work, and virtual whiteboard applications. Other multicast applications such as remote education require a lesser amount of interaction. A third group of multicast applications are noninteractive, e.g., mailing lists and some real-time control applications.

In general, a multicast communication session involves multiple sources transmitting to multiple destinations. This is the **many-to-many** multicasting problem. Videoconferencing is an obvious example of an application involving multiple sources and multiple receivers. The **one-to-many** multicasting problem is a special case of the many-to-many problem, in which the multicast session involves only one source. An example of one-to-many communication is real-time control application in which a sensor transmits its readings to more than one remote control stations. One approach to establish a many-to-many communication session is by setting up multiple one-to-many sessions. We investigate the one-to-many problem in chapters 3 and 4. The many-to-many problem is studied in chapters 6, 7, and 8.

The **host group model** proposed by Deering [3, 4] for representing multicast sessions is adopted by several multicast protocols. It defines a multicast group as the set of receivers of a multicast session. A multicast group is identified by a *unique* group address. The use of a unique group address allows *logical addressing*, i.e., a source needs only to know the group address in order to reach all receivers. It does not need to know the addresses of the individual receivers. In addition, the source itself need not be a member of the multicast group. Logical addressing is advantageous for applications with large numbers of sources and receivers, such as mailing lists or news groups, and for dynamic applications, such as computer-supported cooperative work, where receivers may join or leave the group at any time and sources may start or stop transmission to that group at any time.

1.2 Routing

Bertsekas and Gallager [5] define the routing function at the network layer as consisting of two parts. The first part selects a route for the session during the connection establishment phase, and the second part ensures that each packet of that session is forwarded along the assigned route. In this dissertation, we consider only the route selection mechanisms.

A **routing algorithm** is a method to select routes connecting a set of sources belonging to a given session to the set of receivers of the same session. We classify the routing algorithms according to the type of the communication session as follows.

- A unicast session involves only one source and one receiver. A unicast routing algorithm constructs a path from the source to the receiver.
- Routing algorithms for a one-to-many multicast session construct a multicast tree rooted at the source and spanning all receivers.
- There are two approaches to the many-to-many multicast routing problem.
    - **Source-specific** multicast trees. Construct a one-to-many multicast tree for each source.
    - **Shared** multicast trees. Construct only one multicast tree to carry the traffic flowing from any source to any destination[2].

  The advantages and disadvantages of each approach will be discussed in chapter 6.
- When the set of receivers of a given session includes all nodes in the network, then the routing algorithm constructs a **broadcast** tree that spans the entire network.

The broadcast and unicast routing problems are special cases of the multicast routing problem. They are usually of lesser complexity than the general problem. The main focus of our work is on multicast routing algorithms, but we will also investigate the broadcast and unicast routing problems.

A **routing protocol** describes how to implement a theoretical routing algorithm in practical networks. Protocols must be robust and fault tolerant. For example,

[2]Some routing mechanisms permit the construction of more than one shared multicast tree per session, as will be discussed in chapters 6 and 8.

a protocol is expected to react fast and safely to link or node failures, in order to minimize the resulting instability in the network. Similarly, a protocol should be designed such that, if given incorrect or outdated input information, the results will not be disastrous for the applications. Our work is a study of routing algorithms and not routing protocols. Our objective is to study the performance of routing algorithms and determine their suitability for real-time communication over high-speed networks. Robustness and fault tolerance are examples of protocol implementation issues that are beyond the scope of this dissertation. Routing protocols usually reside in the network layer of the open systems interconnection (OSI) protocol stack.

1.3 Quality of Service Requirements

Real-time applications impose strong requirements on the underlying network. The applications' demands are expressed by QoS parameters such as acceptable end-to-end delay and delay jitter, needed bandwidth, and acceptable loss rate. The QoS parameters are usually dependent on the traffic streams. For example video and audio streams can tolerate certain loss rates, but they have stringent end-to-end delay and delay jitter requirements. High bandwidth must be guaranteed in order to accommodate the high transmission rates of real-time video. Data streams require very low loss rates, but their end-to-end delay and delay jitter requirements are lenient.

The upper bound on end-to-end delay from any source to any receiver in a real-time session is the main QoS parameter we consider during our investigation of various routing problems. In high-speed wide-area networks, the transmission delay is small and the queueing delay is also small, because small buffer sizes are used [1]. Therefore, the propagation delay is the dominant component of the link delay. The propagation delay is proportional to the distance traversed by the link. It is fixed, irrespective of the link utilization. Therefore a route selection algorithm can guarantee an upper bound on the end-to-end delay by choosing the appropriate links for the session being initiated, such that the delay from any source to any receiver does not exceed the delay bound. All routing problems studied in this dissertation are formulated as delay-constrained optimization problems, i.e., the upper bound on end-to-end delay is used as a constraint. The other QoS parameter considered in this work is the required bandwidth. Bandwidth reservation is the responsibility of the resource reservation and admission control mechanisms. It is not possible, however, to study the routing aspect of the problem in isolation from resource reservation and admission control as will be discussed in chapter 3.

1.4 Dissertation Outline

Chapter 2 starts with a classification of multicast routing algorithms. Then we survey previous work on multicast routing for communication networks. We conclude this chapter with a brief discussion of multicast routing protocols.

In chapters 3 and 4, we consider source-specific multicast trees only. In chapter 3, we use simulation to evaluate the performance of selected multicast routing algorithms based on their efficiency in managing the network bandwidth and their ability to satisfy the end-to-end delay requirements of real-time applications. We evaluate both classical multicast routing algorithms as well as new algorithms that were designed specifically for applications imposing delay constraints on the underlying network.

We study a special case of the delay-constrained multicast routing problem, the delay-constrained broadcast routing problem, in chapter 4. We formulate the problem as a delay-constrained minimum spanning tree problem. Then we prove that this problem is *NP*-complete, and we propose an efficient heuristic for solving it. Finally, we evaluate the performance of the proposed heuristic, and compare it to existing algorithms, that can be used to solve the same problem, using simulation.

In chapter 5, we study another special case of the delay-constrained multicast routing problem, the delay-constrained unicast routing problem. This problem is *NP*-complete, and therefore we propose a heuristic for solving it. We prove the correctness of the proposed heuristic and derive its complexity. Then we evaluate its performance using simulation.

Chapters 6, 7, and 8 are devoted to study the problem of constructing shared multicast trees. The routing algorithms applied to construct the shared trees are the same as those considered when evaluating source-specific trees in chapter 3. However, a shared multicast tree needs to have a node designated to be its center. These chapters focus on the center selection problem and not on the routing algorithms.

In chapter 6 we discuss the advantages and disadvantages of using shared multicast trees. Then we present the different types of shared trees and discuss the function of the tree center for each type. Finally, we survey the literature for previous work on the center selection problem.

We investigate the problem of constructing a single, delay-constrained shared multicast tree per session in chapter 7. This problem is *NP*-complete. Therefore we propose heuristics for constructing the shared tree and selecting its center. Then we evaluate the performance of the proposed heuristics.

In chapter 8, we show that, in certain cases, a single shared multicast tree may not be able to satisfy the delay constraint imposed by the application. This motivates us to study the problem of finding the minimum number of shared multicast trees, and hence the minimum number of centers, necessary to satisfy the delay constraint. Similar to all problems we studied before, this problem is *NP*-complete. So, to avoid the complexity of finding the optimal solution, we propose heuristic solutions and evaluate their performance using simulation.

In chapter 9 we present our conclusions along with directions for future work. Finally, we summarize the contributions of the research presented in this dissertation.

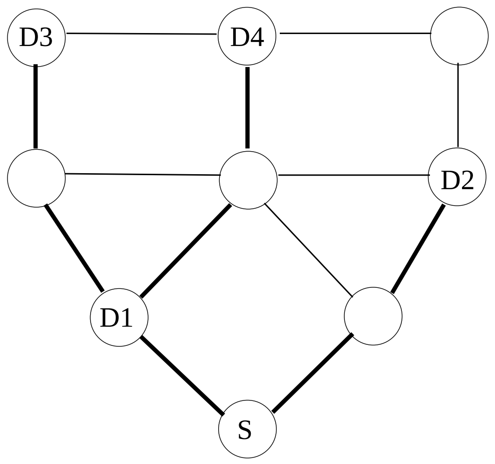
Chapter 2

The Multicast Routing Problem

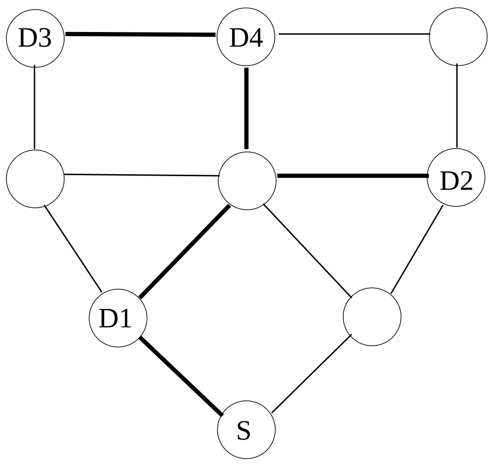
Researchers have been studying variations of the multicast routing problem in communication networks for many years. However, multicasting was not deployed over wide-area networks until recently. For example, the first sizeable multicast experiment over the Internet was the "audiocast" of the Internet Engineering Task Force (IETF) meeting to 20 sites on three continents [6]. It took place in March of 1992. Huge efforts are currently under way, at both the algorithm and protocol levels, to develop multicast routing mechanisms which:

- satisfy the QoS requirements of the rapidly evolving real-time applications,
- are capable of managing the network resources efficiently, and,
- scale well to large network sizes.

The main objective of this chapter is to survey previous work on multicast routing algorithms. In the next two sections, we present a classification of multicast routing algorithms and discuss some important criteria that should be considered when surveying previous work. In section 2.3, we present important definitions. Then in sections 2.4 and 2.5, we survey previous work on multicast routing algorithms. A discussion of multicast routing protocols is given in section 2.6. Finally, we present some concluding remarks in section 2.7.



(a) Shortest path tree. Total cost = 7, maximum path length = 3, average path length = 2.25.



(b) Minimum Steiner tree. Total cost = 5, maximum path length = 4, average path length = 2.75.

Figure 2.1: Comparison of a shortest path tree and minimum Steiner tree for the same multicast group and the same multicast source. Unit lengths and unit costs are assigned to all links. The multicast group $G = \{D_1, D_2, D_3, D_4\}$. Node S is the multicast source.

2.1 A Classification of Multicast Routing Algorithms

Multicast routing algorithms can be classified into two categories. The first category is the **shortest path** algorithms. These algorithms construct a multicast tree that minimizes the length of each path from the source node to a multicast group member node. The other category is the **minimum Steiner tree** algorithms. The objective of these algorithms is to minimize the total cost of the multicast tree. This problem is known to be *NP*-hard [7]. If the set of receivers of a minimum Steiner tree includes all nodes in the network, it is called a **minimum spanning tree**. The minimum spanning tree problem is solvable in polynomial time [8]. The example given in figure 2.1 illustrates the differences between shortest path trees and minimum Steiner trees.

In order to support real-time applications, network algorithms and protocols must be able to provide QoS guarantees. For example, a guaranteed upper bound on end-to-end delay must be provided to certain distributed multimedia applications. It is necessary and sufficient for the network to satisfy the given bound, i.e., there is no need to minimize the end-to-end delay. Multicast routing algorithms proposed specifically for high-speed networks construct an efficient multicast tree without violating the constraint implied by the upper bound on delay. These are called **delay-constrained** algorithms, to distinguish them from other algorithms which are unconstrained.

In many cases, the network nodes have a limited copying capability, i.e., there is an upper limit on the number of copies of an incoming packet which the node can forward simultaneously to next hop nodes on the multicast tree. This is known as the degree constraint, and algorithms that consider this problem are **degree-constrained** multicast routing algorithms.

Dynamic multicast routing algorithms permit sources and receivers to join and leave a multicast session and the corresponding multicast trees at any moment. In **static** multicast routing algorithms, however, the multicast group is fixed, and paths from the sources to all receivers are computed at the same time, when initiating the multicast session.

In **distributed** multicast routing algorithms, the computations required to construct a multicast tree are shared among multiple nodes. This reduces the computational overhead at each node but requires messages to be exchanged between the nodes. The complexity of these algorithms is measured by the number of messages exchanged. The amount of information about the state of the network that must be stored at each node is another factor affecting the practicality of a distributed algorithm. **Centralized** multicast routing algorithms are usually more stable than the distributed algorithms. However, complete network topology information must be available at any node running the centralized algorithm.

As has been mentioned already in section 1.2, multicast trees can be classified into source-specific trees and shared trees. The construction of a shared multicast tree consists of two parts: the **center selection** part and the **route selection** part. On the other hand, the construction of a source-specific multicast tree requires only the selection of proper routes. We focus in this chapter on route selection only. The center selection problem will be surveyed and studied in detail in chapters 6, 7, and 8.

2.2 Important Criteria for Multicast Routing Algorithms

We consider the following criteria when summarizing the features and performance of the different multicast routing algorithms.

- Network management efficiency. The ability of the algorithm to manage the network bandwidth and buffer space efficiently. The cost of a link is frequently defined as a function of the utilized link bandwidth. We therefore refer to the efficiency of an algorithm in managing the network bandwidth as the cost performance of the algorithm.
- End-to-end delay performance. It is a measure of the suitability of the algorithm for real-time applications imposing delay constraints.
- Complexity of the algorithm. The algorithm's complexity together with the possibility of distributed implementation are major factors in determining the algorithm's scalability to large network sizes.
- Symmetric/asymmetric[1] networks. The multicast routing problems in symmetric networks are less complex than the equivalent problems in asymmetric networks. Furthermore, an algorithm designed with the assumption that the network is symmetric is not guaranteed to perform well when applied to asymmetric networks, even if it performs well with symmetric networks.

The performance of multicast routing algorithms is usually evaluated using simulation. Researchers made different assumptions and used different scenarios when evaluating multicast routing algorithms. Some of the major differences between these scenarios are listed below.

- The network topology used, e.g., real networks, randomly generated networks, or mesh networks.
- The size of the networks used.

[1]Networks are in general asymmetric [9, 10], i.e., the delay and cost of a full duplex link are not necessarily equal in both directions. If these parameters are equal in both directions for all links in the network, the network is called symmetric. Exact definitions of asymmetric and symmetric networks are provided in chapter 3.

- The link cost and link delay functions used. The link cost may be some monetary cost or a function of the link's utilization, or, in some cases, unit cost is assigned to all links. In the past some researchers even defined a link's cost to be equal to its delay. The link delay may represent only the queueing component of the link's delay or only the propagation component of the link's delay or both.

If researchers base their evaluation of an algorithm's performance on simulation of special case scenarios, such as mesh networks, then it is not appropriate to generalize the results of their work to all networks. It is therefore important for us to take the evaluation scenarios into account when surveying previous work on multicast routing algorithms.

The literature survey, given in section 2.4, uses the classification of section 2.1 and the criteria listed above to differentiate between the various algorithms. Before proceeding to the survey itself, we present important definitions.

2.3 Definitions

A point-to-point communication network is represented as a directed, connected, simple network $N = (V, E)$, where V is a set of nodes and E is a set of directed links. The existence of a link $e = (u, v)$ from node u to node v implies the existence of a link $e' = (v, u)$ for any $u, v \in V$, i.e., full duplex in networking terms. A link $(u, v) \in E$ is an outgoing link for node $u \in V$ and an incoming link for $v \in V$. Any link $e = (u, v) \in E$ has a cost $C(e)$ (same as $C(u, v)$) and a delay $D(e)$ (same as $D(u, v)$) associated with it. $C(e)$ and $D(e)$ may take any nonnegative real values. The link cost $C(e)$ may be either a monetary cost or some measure of the link's utilization. We will use link cost metrics that are a function of the link utilization in this work, because our objective is to achieve efficient management of network resources. The link delay $D(e)$ is a measure of the delay a packet experiences when traversing the link e. Thus it may consist of queueing, transmission, and propagation components. Because of the asymmetric nature of computer networks, it is often the case that $C(e) \neq C(e')$ and $D(e) \neq D(e')$. If the network is symmetric, it can be represented as an undirected network in which $C(e) = C(e')$ and $D(e) = D(e')$ for all $e \in E$.

We define a path as an alternating sequence of nodes and links $P(v_0, v_k) = v_0, e_1, v_1, e_2, v_2, \ldots, v_{k-1}, e_k, v_k$, such that every $e_i = (v_{i-1}, v_i) \in E$, $1 \leq i \leq k$. A path contains loops if some of its nodes are not distinct. If all nodes are distinct, then the path is loop-free. In the remainder of this dissertation, it will be explicitly mentioned if a path contains loops. Otherwise a "path" always denotes a loop-free path. We will use the following notation to represent a path: $P(v_0, v_k) = \{v_0 \rightarrow v_1 \rightarrow \cdots \rightarrow v_{k-1} \rightarrow v_k\}$. The cost of a path $P(v_0, v_k)$ is defined as the sum of the costs of the links constituting $P(v_0, v_k)$:

$$Cost(P(v_0, v_k)) = \sum_{e \in P(v_0, v_k)} C(e). \tag{2.1}$$

Similarly, the end-to-end delay along the path $P(v_0, v_k)$ is defined as the sum of the delays on the links constituting $P(v_0, v_k)$:

$$Delay(P(v_0, v_k)) = \sum_{e \in P(v_0, v_k)} D(e). \tag{2.2}$$

The definitions given below apply for a multicast session with a single source. A multicast group $G = \{g_1, ..., g_n\} \subseteq V$, where $n = |G| \leq |V|$, is a set of nodes participating in the same network activity, and is identified by a unique group address i. A node $s \in V$ is a multicast source for the multicast group G. A multicast source s may or may not be itself a member of the group G. A source-specific multicast tree $T(s, G) \subseteq E$, is a tree rooted at the source s and spanning all members of the group G. The total cost of a tree $T(s, G)$ is simply the sum of the cost of all links in that tree.

$$Cost(T(s, G)) = \sum_{e \in T(s,G)} C(e) \tag{2.3}$$

In general, an algorithm that minimizes the total cost of a multicast tree will encourage the sharing of links[2]. The maximum end-to-end delay of a multicast tree is the maximum delay from the source to any multicast group member.

$$Max\_Delay(T(s, G)) = \max_{g \in G}\left(\sum_{e \in P_T(s,g)} D(e)\right) \tag{2.4}$$

where $P_T(s, g)$ is the path from s to g along the tree $T(s, G)$.

[2]A shared link is a link on the paths from the source to more than one destination.

2.4 Survey of Multicast Routing Algorithms

It is beyond the scope of this survey to list the pseudo code of each individual algorithm. We only present the distinguishing features of each algorithm. The algorithms are grouped together in compliance with the classification provided in the previous section. For simplicity purposes, we will consider only source-specific trees when describing the features of an algorithm. However, many of the algorithms surveyed are also applicable for the construction of shared trees.

The focus of this dissertation is on the ability of multicast routing algorithms to satisfy the delay constraints of real-time applications. Due to the large amount of work reported in the literature on multicast routing problems, we can not survey all of them in detail. So we focus primarily on static multicast routing algorithms, both unconstrained and delay-constrained. Such algorithms will be covered in more detail in the survey than the other classes of multicast routing algorithms.

2.4.1 Unconstrained Shortest Path Algorithms

As the name indicates, a shortest path algorithm minimizes the sum of the lengths of the individual links on each individual path from the source node to a multicast group member. The properties of a shortest path tree depend on the metric the link length represents. If unit link lengths are used, the resulting shortest path tree is a minimum-hop tree. If the link length is set equal to the link cost, then a shortest path algorithm, denoted as the least-cost (LC) algorithm in this case, computes the LC tree. The objective of an LC algorithm can be expressed mathematically as follows:

$$\min_{T(s,G)\in\mathcal{T}(s,G)} \quad Cost(P_T(s,g)) \qquad \forall\, g \in G, \tag{2.5}$$

where $\mathcal{T}(G)$ is the set of trees rooted at s and spanning all nodes in G. The total cost of an LC tree is not necessarily optimal. If the length of a link is a measure of the delay on that link, then a shortest path algorithm, denoted as least-delay (LD) algorithm in this case, computes the LD tree. The objective function of an LD algorithm is to:

$$\min_{T(s,G)\in\mathcal{T}(s,G)} \quad Delay(P_T(s,g)) \qquad \forall\, g \in G. \tag{2.6}$$

An LD tree is optimal with respect to end-to-end delay. In case of real-time applications, if the LD tree can not satisfy the imposed delay constraint, no other multicast tree can.

Bellman-Ford algorithm [11] and Dijkstra algorithm [12] are two well known shortest path algorithms. Both algorithms are exact and run in polynomial time. The worst case time complexity of the Bellman-Ford algorithm is $O(|V|^3)$ where $|V|$ is the number of nodes in the network. An exact, distributed version of the Bellman-Ford algorithm is given in [5]. It requires only limited information about the network topology to be kept at each node. Awerbuch et al. [13] show that the worst case message complexity of the exact, distributed Bellman-Ford algorithm may grow exponentially with the number of nodes. To avoid this excessive complexity, they propose two approximate distributed versions of the algorithm. For Dijkstra's shortest path algorithm, only centralized versions exist. Its execution time is $O(|V|^2)$ time in the worst case. Efficient, nondistributed versions of both Bellman-Ford and Dijkstra's algorithms have comparable average running times [14]. Both algorithms remain exact for asymmetric networks.

The reverse path forwarding (RPF) algorithm proposed by Dalal and Metcalfe [15] is an algorithm for broadcasting in datagram networks. Each packet is forwarded from the source to the receivers over the reverse shortest paths, i.e., the shortest paths from the receivers back to the source. Thus RPF creates an optimal shortest path broadcast tree only if the network is symmetric. Deering [4, 16] generalized the RPF algorithm to the multicast case by presenting the truncated reverse path broadcasting (TRPB) algorithm and the reverse path multicasting (RPM) algorithm. The objective function of TRPB and RPM can be stated as:

$$\min_{T(s,G)\in\mathcal{T}(s,G)} \quad \sum_{e\in P_T(s,g)} C(e')) \qquad \forall\, g \in G, \tag{2.7}$$

where $e = (u, v)$ and $e' = (v, u)$. TRPB and RPM do not suffer from some of the limitations which RPF suffers from with respect to its applicability to multi-access networks. RPF, TRPB, and RPM are distributed algorithms that rely on limited information at each node in the network. They scale well with the size of the network, and dynamic implementations of these algorithms exist.

2.4.2 Unconstrained Minimum Steiner Tree Algorithms

The objective of the minimum Steiner tree problem is to minimize the total cost of the multicast tree, i.e.,

$$\min_{T(s,G)\in\mathcal{T}(s,G)} \quad Cost(T(s,G)). \tag{2.8}$$

This problem is known to be *NP*-complete [7]. Hwang [17] provided an extensive survey of both exact and heuristic minimum Steiner tree algorithms. An earlier survey was given by Winter [18]. Very few algorithms have been proposed for the minimum Steiner tree problem in asymmetric networks, and all of them operate under special assumptions, e.g. acyclic networks. If the multicast group includes all nodes in the network, the minimum Steiner tree problem reduces to the minimum spanning tree problem. The minimum spanning tree problem in symmetric networks can be solved in $O(|V|^2)$ time in the worst case using Prim's algorithm [8]. Unconstrained minimum Steiner tree algorithms do not attempt to optimize the end-to-end delay at all. Therefore they may not be suitable for real-time applications. The best known minimum Steiner tree heuristics were proposed by Kou, Markowski, and Berman (KMB heuristic) [19], Takahashi and Matsuyama (TM heuristic) [20], and Rayward-Smith (RS heuristic) [21].

The KMB heuristic [19] uses Prim's minimum spanning tree algorithm [8] during its computation. Prim's algorithm is optimal only for symmetric networks. Thus the cost performance of the KMB heuristic may be affected if it is applied to asymmetric networks. The worst case time complexity of the KMB heuristic is $O(|G||V|^2)$, where $|G|$ is the size of the multicast group. Wall [22, 23] proposed a distributed version of the KMB heuristic. The total cost of trees generated using KMB heuristic in symmetric networks is on the average only 5% worse than the cost of the optimal minimum Steiner tree [24, 25].

The TM heuristic [20] starts with a tree that contains the source node only. Then it adds the multicast group members, one at a time, to the existing tree via the cheapest LC path to any node already in the tree. TM heuristic runs in $O(|G||V|^2)$ time in the worst case.

The RS heuristic [21] starts with a forest of trees, with each multicast group member constituting a tree. Then the heuristic unites trees that are closest to each other (in terms of cost) by adding the appropriate links until it ends up with a single tree. Using a limited number of simulations, Rayward-Smith and Clare [26] showed that RS heuristic yields tree costs that are closer to optimal than KMB and TM heuristics. Unfortunately, however, RS heuristic was designed for symmetric networks, and we can not envision an efficient method for implementing it in case of asymmetric networks.

Jiang [27] presented modified versions of KMB heuristic and RS heuristics that construct multicast trees with lower costs than the original heuristics. The author used heterogeneous link capacities in the symmetric, random networks he simulated, and he defined the link cost as function of the utilized link bandwidth. The same author also proposed a distributed minimum Steiner tree heuristic in [28].

Recently, Ramanathan [29] proposed a heuristic for constructing minimum Steiner trees in asymmetric networks. This heuristic permits trading off low tree cost for fast execution time by proper selection of a parameter κ. The author showed that Dijkstra shortest path algorithm, KMB minimum Steiner tree heuristic, and TM minimum Steiner tree heuristic are particular cases of the proposed heuristics when κ is set to 1, $(|G| + 1)$, and $|V|$ respectively.

Many other heuristics for constructing minimum Steiner trees in communication networks were proposed. See for example Chow [30], Leung and Yum [31], and Bauer and Varma [32].

2.4.3 Delay-Constrained Shortest Path Algorithms

Delay-constrained shortest path algorithms minimize the cost of each path, i.e., the sum of the link costs, from the source node to a multicast group member subject to an end-to-end delay constraint. Thus the tree is a delay-constrained LC tree. An algorithm for solving the delay-constrained LC problem has the same objective function as that of the unconstrained LC problem, stated in equation 2.5, with the added constraint that:

$$Max\_Delay(T(s, G)) \leq \Delta, \tag{2.9}$$

where Δ is the value of the imposed delay constraint. The delay-constrained shortest path problem is *NP*-hard [33]. A few algorithms for solving that problem were proposed recently, motivated by the increasing importance of end-to-end delay as a QoS constraint for real-time applications. We summarize the distinguishing characteristics of each algorithm below. Note that the delay-constrained multicast routing algorithms surveyed in this section and in the next section are only applicable for the construction of source-specific trees.

Widyono [34] presented the constrained Bellman-Ford (CBF) algorithm. CBF performs a breadth-first search to find the delay-constrained shortest path tree. CBF is optimal and therefore its running times grow exponentially with the size of the network. Widyono used CBF as a basis for several delay-constrained minimum Steiner tree heuristics which will be surveyed in the next section.

Sun and Langendoerfer [35] proposed a delay-constrained shortest path heuristic. We call it the constrained Dijkstra heuristic (CDKS) because it is based on Dijkstra shortest path algorithm. This heuristic computes an unconstrained LC tree. If the end-to-end delay to any group member violates the delay constraint, the path from the source to that group member is replaced with the LD path. Thus if the LC tree violates the delay constraint, an LD tree must be computed, and the two trees are merged. This algorithm always finds a constrained multicast tree if one exists. CDKS runs in $O(|V|^2)$ time, the same as Dijkstra's algorithm. The authors compared the cost performance of their heuristic to LD and KPP (a delay-constrained minimum Steiner tree heuristic which will be presented in the next section) using simulation over random networks. They used unit link costs and integer link delays ranging in value from 1 to 5.

Wi and Choi [36] presented a distributed LD algorithm and proposed to use it for solving the delay-constrained shortest path problem. They used simulation to evaluate its performance and execution times relative to KPP for 20-node symmetric networks.

2.4.4 Delay-Constrained Minimum Steiner Tree Algorithms

The delay-constrained source-specific minimum Steiner tree problem was first formulated by Kompella, Pasquale, and Polyzos [37, 38]. The authors proved the *NP*-completeness of the problem. The objective of the problem is to minimize the total cost of the tree, equation 2.8, without violating the imposed delay constraint, equation 2.9. Optimal algorithms for this problem exist. For example, Noronha and Tobagi [39] proposed an algorithm, based on integer programming, which constructs the optimal source-specific delay-constrained minimum Steiner trees for multiple multicast sessions simultaneously. However, this algorithm is rather complex and is useful only as a reference to evaluate heuristic solutions for the same problem.

The first heuristic for the delay-constrained minimum Steiner tree problem was given by Kompella, Pasquale, and Polyzos [37, 38]. We label this the KPP heuristic. KPP assumes that the link delays and the delay constraint, Δ, are integers while the link costs may take any positive real value. The heuristic is dominated by computing a constrained closure graph which takes time $O(\Delta|V|^3)$. Thus KPP takes polynomial time only if Δ has a fixed value. When the link delays and Δ take noninteger values, Kompella et al. propose to multiply out fractional values to get integers. Following this approach, KPP is guaranteed to construct a constrained tree if one exists. However, in some cases the granularity of the delay constraint becomes very small, and hence the number of bits required to represent it increases considerably. As a result the order of complexity, $O(\Delta|V|^3)$, may become too high. To avoid prohibitively large computation times, a fixed granularity may be used. However, fixing the granularity has side effects. When the granularity is comparable to the average link delays, KPP's accuracy is compromised and in many cases it fails to construct a constrained multicast tree when one exists. The authors proposed two alternative objective functions for KPP to use during tree construction. The first is a function of the link cost only. The second objective function is a function of both the link cost and the residual delay if this link is added to the tree. The authors used simulation of random, symmetric networks with up to 100 nodes to evaluate their heuristic.

Similar to KMB, KPP uses Prim's algorithm [8] to obtain a minimum spanning tree of a closure graph. Prim's algorithm is only optimal for symmetric networks. This might affect the performance of KPP when applied to asymmetric networks.

Kompella, Pasquale, and Polyzos also proposed a distributed heuristic solution for the delay-constrained minimum Steiner tree problem [40]. The heuristic is based on Prim's algorithm [8], but it involves the making and breaking of cycles during the construction of the multicast tree. It runs in $O(|V|^3)$ time, and is guaranteed to find a multicast tree, if one exists.

Widyono [34] proposed four delay-constrained minimum Steiner tree heuristics. The four delay-constrained heuristics are based on the CBF algorithm described in the previous section. Therefore all of them have worst case scenarios with exponentially growing execution times. Widyono's constrained adaptive ordering (CAO) heuristic yields better performance than the other three constrained heuristics he proposed. In CAO, the CBF algorithm is used to connect one group member at a time to the source. After each run of CBF, the unconnected member with the cheapest constrained LC path to the source is chosen and is added to the existing subtree. The costs of links in the already existing subtree are set to zero. CAO is always capable of constructing a constrained multicast tree, if one exists, because of the nature of the breadth-first search CBF conducts. Widyono defined the link cost as a function of the available bandwidth, the residual buffer space, and the link's delay. The link delay was defined as the sum of the queueing, transmission, and propagation delays along the link. The author evaluated his heuristics using simulation of eight by eight mesh networks.

The bounded shortest multicast algorithm (BSMA) was proposed by Zhu, Parsa, and Garcia-Luna-Aceves [41]. BSMA starts by computing an LD tree for a given source and multicast group. Then it iteratively replaces superedges[3] in the tree with cheaper superedges not in the tree, without violating the delay constraint, until the total cost of the tree can not be reduced any further. BSMA uses a kth-shortest path algorithm to find cheaper superedges. It runs in $O(k|V|^3 \log |V|)$ time. In case of large, densely connected networks, k may be very large, and it may be difficult to achieve acceptable running times. It is possible to tradeoff multicast tree cost for fast execution speed when using BSMA by either limiting the value of k in the kth-shortest path algorithm or by limiting the number of superedge replacements. BSMA always finds a constrained multicast tree, if one exists, because it starts with an LD tree. The authors defined the link cost as a function of the link utilization and defined the link delay as the sum the queueing delay, transmission delay, and propagation delay over the link. They evaluated the performance of BSMA and compared it to KMB and LD. Random networks with up to 100 nodes generated using Waxman's random

[3]A superedge is a path in the tree between two branching nodes or two multicast group members or a branching node and a multicast group member.

network generator [42] were used.

We consider the minimum Steiner tree heuristic proposed by Waters [43] to be semi-constrained, because it uses the maximum end-to-end delay from the source to any node in the network (not to any group member) as the delay constraint. Note that this constraint is not related directly to the application's QoS constraints, and that, depending on the network delays, this internally computed constraint may be too strict or too lenient as compared to the QoS requirements of the application. The heuristic then constructs a broadcast tree that does not violate the internal delay constraint. Finally the broadcast tree is pruned beyond the multicast nodes. We call this the semiconstrained (SC) heuristic. In [44], we implemented the original algorithm proposed in [43] which resembles a semi-constrained minimum spanning tree, and we also implemented a modified version which is closer to a semi-constrained shortest paths broadcast tree. Simulation results given in [44] showed that the modified version, denoted as the modified semiconstrained (MSC) heuristic always performs better than the original heuristic with respect to tree costs, end-to-end delays, and network balancing. SC and MSC is dominated by the computation of the internal delay bound. This computation uses an extension to Dijkstra's algorithm, and therefore it takes $O(|V|^2)$ time in the worst case.

In addition to the algorithms surveyed above, many other variations of the multicast routing problem have been studied over the years. Research reports on the dynamic multicast routing problem, in particular, appeared frequently in the literature. We dedicate the next section to previous work on that problem. Then, in section 2.4.6, we survey previous work on other variations of the multicast routing problem.

2.4.5 Dynamic Multicast Routing Algorithms

Dynamic multicast routing algorithms were proposed to avoid rerouting an entire multicast tree whenever a node joins or leaves a multicast session. In dynamic multicast routing algorithms, when a node leaves a multicast session, the path connecting that node is simply pruned from the tree if it is not used to connect any other multicast group members. The situation is more difficult when a node joins an existing multicast session.

Waxman [42, 45] presented a greedy dynamic multicast routing algorithm. The algorithm has a weighting parameter w that varies from 0 to 0.5. When $w = 0$, a node joins an existing source-specific multicast tree via the shortest path to the tree. When $w = 0.5$, the node is added to the existing tree via the shortest path to the source. Waxman evaluated his algorithm using simulation over randomly generated 56-node and 60-node networks. He proposed an algorithm for generating random networks that resemble realistic networks. This random network generator has been adopted by many researchers in subsequent years.

Doar and Leslie [24] investigated a naive approach that always connects a joining node to the existing tree via the shortest path from the source. They simulated this mechanism using randomly generated networks, both flat and hierarchical. Their random network generator is a modified version of Waxman's generator. Simulations over 100-node networks showed that the naive approach constructs trees that are on the average 50% more expensive than costs of trees constructed using the static KMB heuristic[4].

Kadirire [46] defined the geographic spread as the shortest distance from any node in the network to the existing tree averaged over all nodes not in the tree. He proposed a geographic spread dynamic multicast (GSDM) algorithm that maximizes the geographic spread for the multicast tree it constructs. Kadirire also evaluated the performance of GSDM and compared it to Waxman's heuristic and Doar and Leslie's heuristic in [47] using simulation over random networks with up to 100 nodes. He showed that GSDM and Waxman's heuristics yield similar performance and are consistently better that Doar and Leslie's naive approach.

Biersack and Nonnenmacher [48] proposed a dynamic, distributed multicast routing algorithm named WAVE. WAVE uses a weighted function of the cost and the delay to attach a joining node to the existing tree. The authors evaluated their algorithm in comparison to static algorithms only.

Bauer and Varma [49] presented a dynamic multicast routing algorithm: ARIES. The operation of ARIES is similar to Waxman's dynamic algorithm. In addition,

[4]With KMB, the entire tree is rerouted each time a node joins the multicast session.

however, a subtree of the multicast tree is completely reconstructed each time a pre-specified number of joins and leaves affects that subtree. The subtree reconstruction ensures that the cost of the multicast tree remains close to optimal. ARIES was evaluated using simulation over 200-node random networks. The authors used a modified version of Waxman's random network generator.

2.4.6 Other Multicast Routing Algorithms

In this subsection, we briefly survey a few more multicast routing algorithms that do not belong to any of the categories listed in the previous subsections.

Bharath-Kumar and Jaffe [50] presented a tradeoff algorithm between the minimum Steiner tree and the LD tree. This algorithm constructs the minimum Steiner tree; then it locates the receiver with the largest difference between the delay along its path in the minimum Steiner tree and the delay along the LD path from the source to that receiver. The algorithm then replaces the minimum Steiner tree path with the corresponding LD path. The same authors also proposed two distributed multicast routing heuristics which are based on local information from nearby nodes only.

Rouskas and Baldine [51] studied the problem of constructing multicast trees subject to both an end-to-end delay constraint and a delay variation constraint. They defined the delay variation constraint as the maximum difference, that can be tolerated, between the end-to-end delays along the paths from the source to any two receivers. The authors proved that this problem is *NP*-complete; then they proposed a heuristic solution.

Research on the degree-constrained multicast routing problem is motivated by the fact that current multicast capable high-speed switches have limited copy capability. In addition, limiting the maximum degree at any node in the multicast tree results in more evenly distributed load among all nodes in the network. Tode et al. [52] proposed two algorithms for degree-constrained multicast routing. The first algorithm minimizes the average degree of the multicast tree it constructs, while the second algorithm attempts to construct a low-cost multicast tree subject to a given maximum degree constraint. The authors set the link costs equal to the link delays when evaluating the performance of their heuristics.

Bauer and Varma [53] investigated a variation of the degree-constrained multicast routing problem in which the degree-constraint may vary for the different nodes in the network. Using simulation, they showed that many of the existing unconstrained minimum Steiner tree heuristics are capable of constructing degree-constraint multicast trees. The authors also proposed a simple degree-constrained heuristic which performs better than all other algorithms of the same or lesser complexity.

Ammar et al. [54] studied the problem of routing virtual paths (VP) for multicast communication in ATM networks. When constructing a multicast tree, they took into account the bandwidth cost, the switching cost, and the connection establishment cost. The authors studied different types of VPs. They formulated the problem as an integer programming problem and proposed heuristic solutions based on the transshipment simplex algorithm. The authors used a single 16-node network for evaluating their heuristics.

Kim [55] studied a similar problem. He proposed an optimal solution to the problem of routing multiple multicast connections simultaneously in ATM networks.

2.5 Previous Work on the Evaluation of Multicast Routing Algorithms

It is evident from the previous section that a large number of algorithms have been proposed for many variations of the multicast routing problem. Researchers used different networking environments and made different assumptions when evaluating the algorithms they proposed. In most cases, a newly proposed algorithm was compared only to very few of the existing algorithms. Therefore, there is a need for work dedicated to the evaluation and comparison of multicast routing algorithms. Unfortunately, very few comparative evaluation studies have been reported in the literature.

An analytical study of the tradeoffs between shortest path trees and minimum Steiner trees was reported by Bharath-Kumar and Kadaba [50] in 1983. The authors did not make any distinction between link cost and link delay, because, at the time this work was reported, QoS issues and resource management issues were not well defined yet.

Tanaka and Huang [56] compared the performance of several static unconstrained minimum Steiner tree algorithms. In addition, they evaluated one dynamic algorithm, the weighted greedy algorithm [42, 45]. The authors ran simulations of a single 20-node symmetric network with the cost of a link being proportional to the distance spanned by that link.

Wei and Estrin [57] studied the LD algorithm and the KMB heuristic for constructing minimum Steiner trees. The authors simulated 50-node and 200-node random networks generated using Waxman's random network generator. The networks were asymmetric, and each link had a delay (equal to its length) and a cost assigned to it. Simulation of 50-node networks with an average node degree of 4 showed that minimum Steiner trees are lower in cost than least-delay trees by approximately 20%. However, the maximum end-to-end delays along minimum Steiner trees are up to 60% larger than those along least-delay trees.

Noronha and Tobagi [58] reported the only evaluation of multicast routing algorithms for real-time applications known to us. They started their paper with an excellent problem formulation followed by a survey of previous work. Then the authors evaluated three unconstrained algorithms, the LD algorithm, the LC algorithm, and the KMB heuristic, with respect to their suitability for real-time applications with delay constraints. The authors used the optimal delay-constrained algorithm which they presented in [39] to benchmark the algorithms. The link cost represented a monetary cost, while the link delay represented the actual delay along the link. The following admission control policy was enforced: the sum of the bandwidths of the multicast sessions utilizing a link can not exceed the link's capacity. The authors defined the blocking probability as the probability that an algorithm fails to construct a multicast tree. There are two causes of failure: delay constraint violation and insufficient bandwidth to support the multicast session. Simulations were run on real networks as well as on random networks. Simulation results showed that, in the absence of a delay constraint, KMB heuristic is almost as good as optimal with respect to tree cost and blocking probability. When a delay constraint is enforced, however, the blocking probability of KMB is higher than those of the other algorithms studied. KMB has up to 20% blocking probability in scenarios in which no other algorithms fail. LD and LC have comparable blocking probabilities in all scenarios.

The same authors also experimented with different randomly generated network topologies, and they concluded that two-connected[5] random topologies yield simulation results that are closest to the results obtained from simulation of real networks. In real networks, short links are more likely to exist than long ones. However, the authors found that it is not necessary to bias the random network generator towards short links in order to get random topologies which yield similar performance to that of real networks.

2.6 Multicast Routing Protocols

Efforts to develop multicast routing protocols for wide-area networks have started in the late 1980s motivated by the rapid growth of the Internet and the emergence of new applications involving multiple users. Semeria and Maufer [59] provided an introduction to IP multicast routing. IP Multicast routing protocols adopt the host group addressing model [3, 4]. The Internet Society designated Class D IP addresses for multicast group addressing. The Internet Group Management Protocol (IGMP) [60, 61] is used to exchange group membership information on a local subnetwork. We will focus only on multicast routing in wide-area networks. An in depth investigation of the complete IP multicast architecture can be found for example in [62]. Currently there are two standard protocols for IP multicast routing. To avoid some shortcomings of these two protocols, two more protocols are being developed and standardized.

The Distance Vector Multicast Routing Protocol (DVMRP) [63] is the first standard protocol for IP multicast routing. It is widely implemented in commercially available routing equipment, and it is the protocol used in most routers of the Internet's Multicast Backbone (MBone) [2, 64] which currently spans thousands of nodes on all continents. DVMRP is based on the TRPB heuristic [4, 16]. It is a distributed protocol that uses the limited information available in the distance vectors of the Routing Information Protocol (RIP) [65, 66] to forward datagram packets from the source to all receivers over the reverse shortest paths. DVMRP uses source-specific multicast routing, and it allows receivers to dynamically join and leave a multicast

[5]Networks in which at least two paths exist between any two nodes.

session. In order to discover new members in a multicast session, and because it depends on soft state, DVMRP periodically sends the source's packets over a broadcast tree to all nodes in the network. Then leaf nodes which are not members of the multicast group send prune messages upstream towards the source to prune the links leading to these nodes from the tree. The occasional broadcasting behavior of DVMRP causes inefficient use of the network bandwidth, especially if the size of the multicast group is small. This limits the scalability of DVMRP to larger networks. Efforts are currently under way to develop a more efficient version of DVMRP [67].

The Multicast Extensions to OSPF (MOSPF) [68, 69, 70] is a standard multicast routing protocol for interior gateway routing, i.e., within a single domain. As the name indicates, MOSPF is based on the Open Shortest Path First (OSPF) [71] unicast routing protocol. MOSPF uses the centralized Dijkstra algorithm to construct the forward shortest path multicast tree. To achieve this, a node that runs MOSPF must maintain complete information about the network topology. Therefore complete topology information must be periodically broadcast to all MOSPF-capable nodes in the networks. The centralized nature of MOSPF as well as the necessary periodical broadcast operations severely limit the scalability of the protocol.

In addition to the limitations mentioned above, both DVMRP and MOSPF rely on specific unicast routing protocols, RIP and OSPF respectively. Thus they can not be deployed on routers not running these unicast protocols. Two new multicast routing protocols are currently being developed to avoid the shortcomings of DVMRP and MOSPF. The two new protocols are independent of the underlying unicast routing mechanisms.

The first protocol currently being developed is Protocol Independent Multicasting (PIM) [72]. PIM specifies a dense mode (PIM-DM) [73] and a sparse mode (PIM-SM) [74]. When a multicast group densely populates an internetwork, PIM-DM is used to create source-specific multicast trees. The basic operation of PIM-DM is very similar to that of DVMRP, but it is independent of the underlying unicast routing protocol. PIM-SM is specified for multicast groups where members are sparsely distributed over an internetwork. PIM-SM uses rendezvous points (RP) which are central nodes at which receivers can meet sources of the same multicast session. A shared tree is constructed around the RP. Receivers join the shared tree via the forward shortest paths towards the RP, and sources transmit to the shared tree via the forward shortest paths towards the RP. Thus packets are forwarded over the reverse shortest paths from the RP to the receivers. A receiver r discovers the existence of a source s when it receives that source's packets over the shared tree. Then the receiver r can elect to continue receiving packets from source s over the shared tree, or, alternatively, it can join that source's specific tree via the reverse shortest path from that s to r. Thus PIM-SM permits the use of both shared trees and source-specific trees. Routing for the same multicast session can use a mixture of shared trees and source-specific trees. Similar to DVMRP, PIM relies on a soft state refreshment mechanism, and thus it suffers from periodical message overhead to maintain the multicast trees. However, soft state enhances the robustness of PIM.

Core Based Trees (CBT) [75, 76, 77, 78] is the other protocol currently being developed for multicast routing over the Internet. In CBT, all sources sending to the same group use a single multicast tree to carry their traffic to all receivers belonging to that group. The multicast tree has one or more cores. Similar to the shared mode of PIM-SM, a new receiver joins an existing core based tree via the forward shortest path towards a core, and therefore packets are forwarded on the tree along the reverse shortest paths from the cores to the receivers. The cores are interconnected via a core backbone. CBT relies on an explicit reliability mechanism to maintain the multicast tree. This mechanism introduces much less message overhead than that introduced by PIM's soft state mechanism.

Both PIM and CBT are expected to scale well to large networks. Interoperability with other multicast routing protocols is being considered in the specifications of both PIM and CBT. A quantitative comparison of the two protocols and suggestions to improve their performance can be found in [79]. In chapter 6, we discuss CBT and the shared mode of PIM-SM further, with emphasis on the functions of cores and RPs respectively, as well as how to select these nodes.

Work on QoS routing has started only very recently, Spring 1996, motivated by the need for routing algorithms capable of providing QoS guarantees. An initial draft of a QoS routing protocol based on OSPF is now available. Quality of Service Extensions to OSPF (QOSPF) [80] is a protocol capable of routing both unicast and multicast connections and reserving bandwidth and other resources for those connections. Unfortunately, we are not aware of any work aimed at developing delay-constrained routing protocols, either unicast nor multicast.

Work on deploying IP multicast routing protocols over ATM networks has also started recently [81]. Work in that area is still at the stage of identifying the obstacles that have to be overcome before IP multicast routing protocols can be successfully deployed over ATM. We are not aware of any multicast routing protocols developed specifically for ATM networks.

2.7 Conclusions

In this chapter, we classified the multicast routing algorithms into different categories based on the problems they address. Then we presented important criteria to be considered when summarizing the features of a multicast routing algorithm. The bulk of this chapter was dedicated to surveying previous work on multicast routing algorithms. Over the years, a lot of algorithms have been proposed for many variations of the multicast routing problem. However, different researchers have made different assumptions when evaluating the performance of the algorithms they proposed. In addition, as can be seen from section 2.5, very few evaluation studies dedicated to comparing the performance of the different algorithms have been reported in the literature.

The objective of this dissertation is to study multicast routing algorithms for real-time applications with delay constraints. Noronha and Tobagi [58] have presented the only subjective comparison of different multicast routing algorithms. In their work they evaluated the ability of the algorithms to satisfy the delay constraints of real-time applications. However, these authors considered only unconstrained multicast routing algorithms in their study. A number of delay-constrained multicast routing algorithms have been proposed during the past few years. In the next chapter, we evaluate most of the new delay-constrained multicast routing algorithms and a few selected unconstrained algorithms and present a quantitative comparison of all these algorithms when applied in realistic high-speed networking environments. Our objective is to evaluate each algorithm's capability of providing guaranteed delay-constrained service for real-time applications and also to evaluate each algorithm's ability to manage the network bandwidth efficiently.

While performing our survey of multicast routing algorithms, we noticed that two special cases of the delay-constrained multicast routing problem have not been studied previously. The two special cases are the delay-constrained broadcast routing problem and the delay-constrained unicast routing problem. We expect these two special cases to be of lesser complexity than the general problem. In addition, there are many examples of delay-constrained applications involving either unicasting or broadcasting. This motivates us to study the two special cases in more detail. In chapter 4, we study the delay-constrained broadcast routing problem. Then we study the delay-constrained unicast routing problem in chapter 5.

After completing our survey of multicast routing algorithms, we briefly surveyed previous and current work on multicast routing protocols. Developing a multicast routing protocol is a complex process. Such a protocol must be simple, robust, fault tolerant, and it must scale well to large network sizes. We noticed that all multicast routing protocols are based on simple multicast routing algorithms. The simple algorithms are not necessarily the most efficient ones with the best performance. We therefore recommend that any future work should consider simplicity and ease of implementation as a criterion when evaluating multicast routing algorithms.

Chapter 3

Evaluation of Source-Specific Multicast Routing Algorithms

In the past, very few network applications involved multiple users, and none of them had QoS requirements. In addition, the bandwidth requirements of most applications were very modest. Thus simple multicast routing algorithms were sufficient to manage the network bandwidth. In many cases, multicast trees were simply constructed by the superposition of multiple unicast paths. The situation is different, however, for the emerging real-time applications. These applications have QoS requirements and large bandwidth demands. The survey of chapter 2 indicates that a number of new multicast routing algorithms designed specifically for real-time applications were proposed during the past few years. However, there has not been a study yet that applies all of these algorithms in a realistic network environment and provides a fair quantitative comparison of all algorithms under identical networking conditions. Such a study is necessary to determine whether or not these algorithms are capable of constructing multicast trees with characteristics suitable for real-time applications and how efficient they are in managing the network resources.

In this chapter, we study the performance of the old multicast routing algorithms, which are used in current wide-area networks, when applied to high-speed networks, and evaluate their ability to satisfy the requirements of real-time applications. In addition, we compare the performance of the new algorithms which were designed specifically for real-time applications. All algorithms evaluated in this chapter were proposed by other researchers. The results of our study would clarify which algorithms, if any, are suitable for future's high-speed networks. These results would also enable us to decide if more work needs to be done, either to improve the performance of the existing algorithms or to design other algorithms that are more capable of fulfilling the requirements of future's real-time applications. Our study focuses on a specific QoS requirement, namely the end-to-end delay constraint, because it is a QoS requirement that can be guaranteed by route selection algorithms, as has been discussed in section 1.3. In this chapter, we classify multicast routing algorithms mainly into unconstrained algorithms and delay-constrained algorithms.

Most previous work on multicast routing assumes networks with symmetric link loads. This is a special case that does not hold true for actual networks [9, 10], and thus we study the general case of networks with asymmetric link loads.

Our evaluation of the multicast routing algorithms considers the construction of source-specific trees only. It is not practical to evaluate all the multicast routing algorithms which have been proposed over the years. Therefore, we only study the following three unconstrained algorithms:

- The least-cost algorithm, **LC**. It is a simple shortest path algorithm that minimizes the cost from the source to each individual receiver. Distributed implementation of LC using Bellman-Ford algorithm is possible [5].
- The Kou, Markowski, and Berman algorithm, **KMB** [19]. This algorithm constructs Steiner trees that are on the average only 5% more expensive (in terms of total tree cost) than optimal minimum Steiner trees in case of symmetric networks. KMB is being used by many researchers as a good approximation algorithm for the minimum Steiner tree. We will study how efficient it is when applied to asymmetric networks. KMB can be implemented distributedly [22, 23].
- Several multicast routing protocols adopt algorithms that construct shortest reverse path multicast trees. We study only one of these algorithms, namely reverse path multicasting, **RPM** [15, 16]. It is a distributed, dynamic algorithm which requires only limited state information to be stored at each node.

In addition to these unconstrained algorithms, we study the following multicast routing algorithms designed specifically for real-time applications with delay constraints.

- The modified semiconstrained heuristic, **MSC** [44]. We showed that it performs better than the original semiconstrained heuristic [43], so it is sufficient to study MSC.
- The constrained Dijkstra heuristic, **CDKS** [35]. It is the only true delay-constrained shortest path heuristic we are aware of.
- The Kompella, Pasquale, and Polyzos heuristic, **KPP** [37, 38]. It was the first heuristic proposed for the delay-constrained minimum Steiner tree problem. It has been mentioned in the previous chapter that KPP assumes that the link delays and the delay constraint, Δ, are integers. We simulate networks in which the link delays may take any real positive values. Multiplying out fractional values may result in very small granularity for the delay constraint. Since KPP runs in $O(\Delta|V|^3)$, a small granularity results in large running times of the algorithm. To avoid prohibitively large computation times, we use a fixed granularity of $\Delta/10$ throughout our experiments. This reduces the accuracy of the algorithm in some cases.
- Widyono [34] proposed four delay-constrained Steiner tree heuristics. We only study the one with the best performance, namely the constrained adaptive ordering heuristic, **CAO**.
- The bounded shortest multicast algorithm, **BSMA**, is the final delay-constrained minimum Steiner tree algorithm we study in this chapter. The authors of BSMA [41] proposed and evaluated their heuristic on symmetric networks. We use a modified version of BSMA in order to account for the effects of asymmetric networks.

In addition to the above selected unconstrained, semiconstrained, and delay-constrained algorithms, we use the following three optimal algorithms as a basis for evaluating the performance of the different heuristics. The unconstrained optimal minimum Steiner tree algorithm, **OPT**, always finds the minimum cost solution for the multicast routing problem. The delay-constrained optimal minimum Steiner tree algorithm, **COPT**, finds the minimum cost solution for the same problem subject to a given delay constraint. Our implementation of OPT and COPT uses a branch and bound technique [82]. Their execution times are very large, so we could only apply them to small networks. The third optimal algorithm is the least-delay multicast routing algorithm, **LD**. We implemented it as a Dijkstra's shortest path algorithm [5] in which $C(e) = D(e)$. Therefore, it guarantees minimum end-to-end delay from the source to each multicast group member.

In this chapter, we evaluate the performance of all the algorithms listed above. This is the first reported, fair, quantitative comparison of these algorithms. Implementation issues such as distributing the algorithms and the amount of state information needed at each node are not addressed in this work. The remainder of this chapter is organized as follows. In section 3.1, we describe the experimental setup and the characteristics of the networks we study. The performance metrics used are discussed in section 3.2 followed by simulation results. Section 3.3 concludes this chapter.

3.1 The Experimental Setup

We used simulation for our experimental investigations of the different multicast routing algorithms to avoid the limiting assumptions of analytical modeling. ATM networks permit the applications to specify their own QoS requirements, and they allow cell multicasting in the physical layer. Thus, it was appropriate for us to comply with the ATM standards.

Full duplex ATM networks with homogeneous link capacities of 155 Mbps (OC3) were used in the experiments. The positions of the nodes were fixed in a rectangle of size 4000$*$2400 Km^2, roughly the dimensions of the continental United States. A random generator (based on Waxman's generator [42] with some modifications) was used to create links interconnecting the nodes. The output of this random generator is always a connected network in which each node's degree is ≥ 2. Therefore the output is always a two-connected network[1]. Noronha and Tobagi [58] have shown, using simulation, that the performance of a multicast routing algorithm when applied to a real network is almost identical to its performance when applied to a random

[1]A two-connected network has at least two paths between any pair of nodes.

two-connected network[2]. We adjusted the parameters of the random generator such that, similar to real networks, the probability of existence of a short link is larger than the probability of existence of a longer link. We also adjusted the parameters of the random generator to yield networks with an average node degree of 4 which is approximately the average node degree of current networks. Details of the random link generator algorithm we used are given in subsection 3.1.1.

Each node represented a non-blocking ATM switch, and each link had a small output buffer. The propagation speed through the links was taken to be two thirds the speed of light. Under this assumption, the size of the rectangle enclosing our network is $0.02 * 0.012$ seconds^2. In addition, we assumed a high-speed networking environment with small packet (cell) sizes and limited buffer space at each node. The link propagation delay was dominant under these assumptions, and the queueing component of the link delay was not taken into account. The link delays were thus symmetric, $D(u,v) = D(v,u)$, because the link lengths, and hence the propagation delays, were symmetric.

For the multicast sources we used variable bit rate (VBR) video sources. These represented realistic, bursty, multimedia traffic sources. Any session traversing a link e, reserved a fraction of e's bandwidth equal to the equivalent bandwidth[3] of the traffic it generated. The link cost, $C(e)$, was set equal to the reserved bandwidth on that link, because it is a suitable measure of the utilization of both the link's bandwidth and its buffer space. Therefore, the cost of a heavily utilized link was larger than the cost of a lightly utilized link. The link costs were dynamic, and varied as new sessions were established or existing sessions were torn down.

A link could accept sessions and reserve bandwidth for them until its cost, i.e., the sum of the equivalent bandwidths of the sessions traversing that link, exceeded 85% of the link's capacity; then it got saturated. This admission control policy allowed statistical multiplexing and efficient utilization of the available resources. More sophisticated admission control policies for real-time traffic exist, but the simple policy just described was sufficient for the purposes of our study of multicast routing algorithms. A detailed study of admission control algorithms for real-time traffic can

[2]A two-connected network remains connected after a single node failure.
[3]Any other suitable measure could also be used.

be found in [83].

Interactive voice and video sessions have tight delay requirements. We used a value of 0.03 seconds for Δ which represents only an upper bound on the end-to-end propagation time across the network. This relatively small value was chosen in order to allow the higher level end-to-end protocols enough time to process the transmitted information without affecting the quality of the interaction. In addition, considering that the network size is 0.02∗0.012 seconds^2, a delay constraint of 0.03 seconds is a challenging value for the routing algorithms. Even the best algorithms can not satisfy this constraint all the time.

3.1.1 The Random Link Generator

We already mentioned that, in our experiments, the positions of the network nodes were fixed in a rectangle which roughly represents the area of the United States of America. In his random network generator, Waxman [42] used the following probability function to create links interconnecting the nodes.

$$P_e(u,v) = \beta \exp \frac{-l(u,v)}{L\alpha} \tag{3.1}$$

L is the maximum distance between any two nodes in the network and $l(u,v)$ is the distance between u and v. The parameter α controls the ratio of short links to long links, while the parameter β controls the average node degree of the network. A large value of α increases the number of long links, and a large value of β results in a large average node degree. Waxman applied his probability function once to each pair of nodes. When following Waxman's algorithm, the resulting network is not always connected. In addition, fixing the values of the parameters α and β does not guarantee that the resulting networks, when applying the algorithm multiple times, have equal average node degrees[4]. Therefore his algorithm has to be applied repeatedly until a connected network with the desired average node degree is constructed. Doar [25] noted that for fixed values of α and β, increasing the number of nodes while keeping the area fixed results in a higher average node degree. Therefore he proposed a

[4]For a fixed value of β, the resulting node degrees vary within a certain range, but they are not necessarily equal.

modified probability function to ensure that the average node degree for given values of α and β is not a function of the number of nodes in the network:

$$P_e(u,v) = \frac{k\overline{e}}{|V|}\beta \exp \frac{-l(u,v)}{L\alpha}, \tag{3.2}$$

where k is a constant and $\overline{e}$ is the desired average node degree.

Applying Waxman's algorithm, or the modified algorithm proposed by Doar, repeatedly until a connected network with the desired average node degree is constructed is a time consuming procedure, especially in case of networks with small average node degrees. As has been mentioned before, in our experiments, we require the node degree of any node to be ≥ 2. This increases the time required to construct an acceptable random network even further. Therefore, neither Waxman's algorithm nor Doar's algorithm are suitable for our purposes, because we generate hundreds of random networks during our experiments.

We require the node degree of any node to be ≥ 2, because this ensures that the resulting network is two-connected. In addition, a node with a degree of 1 can be reached only via a single link. This trivializes the routing problem for that node.

To avoid running Waxman's algorithm repeatedly, we designed a random link generation algorithm which we apply only once to obtain a connected random network. In addition, the output of our algorithm is guaranteed to be a network having the desired average node degree and such that each node's degree is ≥ 2. The proposed algorithm uses the probability function of equation 3.1. It starts by selecting a random node and creating two random links connecting it to two different nodes. The result is an initial connected subnetwork consisting of three nodes connected via two links. Then the algorithm repeats the following step for each node in the network. If the node's degree is less than 2, the algorithm creates random links attached to that node until its node degree reaches 2. If that node is not connected to the already connected subnetwork, the algorithm makes sure that one of the random links it creates connects it to the already connected subnetwork. The resulting network at the end of this loop is a connected network in which the node degree of any node is ≥ 2. After that, the algorithm keeps generating links between random nodes until the desired average node degree is reached. Pseudo code of the algorithm is listed in appendix A. Note, however, that our algorithm may not be suitable for generating random networks with an average node degree less than 4 for reasons discussed in appendix A.

In our experiments we used random networks with an average node degree of 4, which is close to the average node degree of the current Internet. We selected the values for the parameters α and β of equation 3.1 carefully in order to obtain random networks with close resemblance to real networks. The values we used for α and β are listed in table 3.1. We use a constant value for β irrespective of the size of the network and irrespective of the desired average node degree, because the average node degree of the network is now controlled by the second phase of the algorithm. Figure 3.1 shows an example of a randomly generated 20-node network.

3.2 Performance Metrics and Experimental Results

The performance of a multicast routing algorithm was evaluated based on the quality of the multicast trees it creates and the algorithm's efficiency in managing the network. The quality of a multicast tree can be defined in the following ways:

- The total cost of the tree. This reflects the algorithm's ability to construct a multicast tree using low-cost, lightly utilized links.
- The maximum end-to-end delay from the source to any multicast group member. It indicates the algorithm's ability to satisfy the delay bound imposed by the application.

An algorithm's efficiency in managing the network resources was judged by monitoring how frequently that algorithm fails to construct an acceptable multicast tree for a given network with given link loads. There are two causes of failure: either the

$\|V\|$	$\bar{e}$	α	β
20	4	0.150	2.2
50	4	0.100	2.2
100	4	0.077	2.2
200	4	0.063	2.2

Table 3.1: Settings of the parameters α and β of the random link generator.

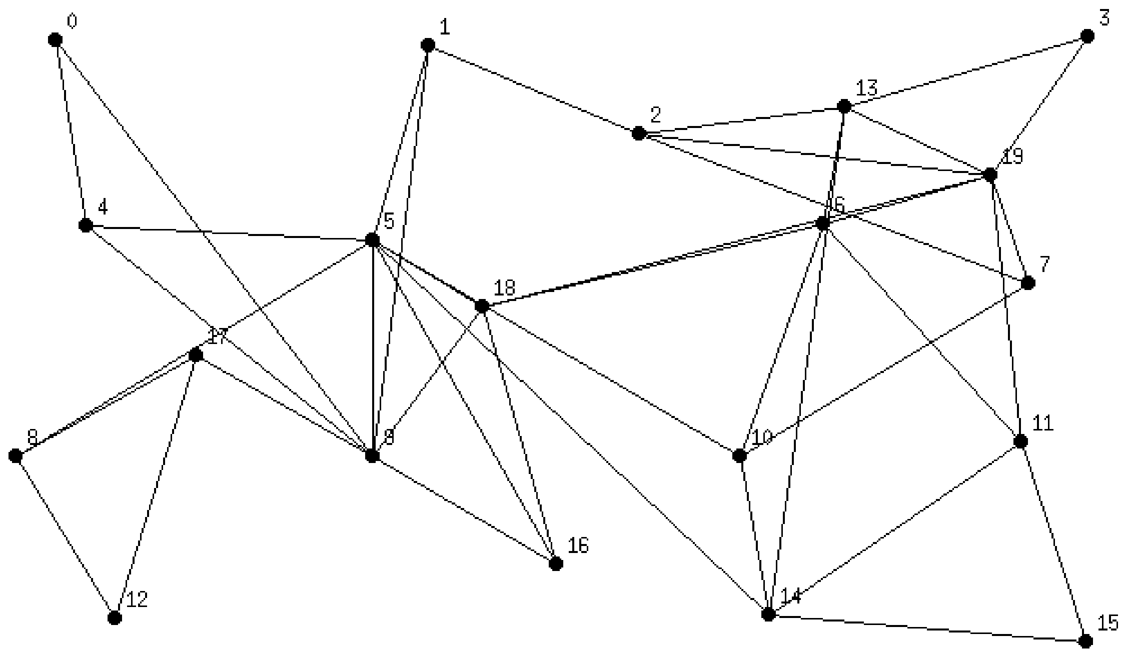


Figure 3.1: A randomly generated network, 20 nodes, average degree 4.

created tree does not satisfy the delay bound, or the algorithm fails to find unsaturated links, and thus it can not create a tree that spans all multicast group members. Another measure of an algorithm's efficiency is the number of multicast trees that the algorithm can create before the cumulative failure rate exceeds a certain limit.

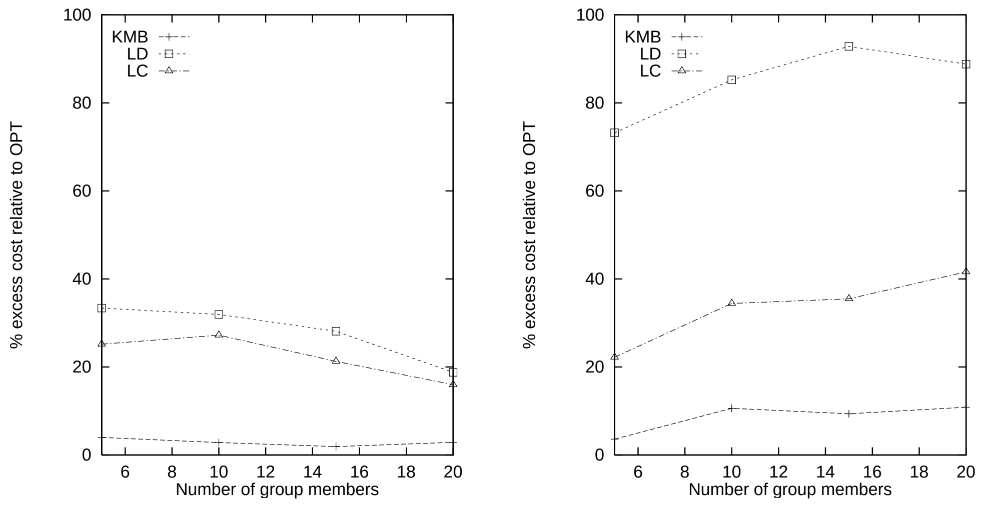
Two experiments were conducted on each of the algorithms studied. We present the simulation results for the unconstrained algorithms first, in section 3.2.1, in order to determine the conditions, if any, under which the unconstrained algorithms do not perform well. Then in section 3.2.2, we study the performance of RPM. Finally, in section 3.2.3, we show the results obtained for the delay-constrained algorithms, and discuss their advantages and disadvantages.

3.2.1 Simulation Results for the Unconstrained Algorithms

The first experiment compares the different algorithms when each of them is applied to create a multicast tree for a given source node generating video traffic with an equivalent bandwidth of 0.5 Mbps, and a given multicast group.

For each run of the experiment we generated a random set of links to interconnect the fixed nodes, we generated random background traffic for each link, we selected a random source node and a multicast group of randomly chosen destination nodes. The equivalent bandwidth of each link's background traffic was a random variable uniformly distributed between B_{min} and B_{max}. As the range of the link loads, i.e., the difference between B_{max} and B_{min}, increased, the asymmetry of the link loads also increased, because the load on link $e = (u, v)$ was independent of the load on the link $e' = (v, u)$. The experiment was repeated with different multicast group sizes. We measured the total cost of the multicast tree, the maximum end-to-end delay, and the failure rate of the algorithm. Note that an unconstrained algorithm may construct a multicast tree with a maximum delay that violates the imposed delay bound. Such a tree was considered a failure and was rejected and removed but not before we measured its characteristics. Thus the total cost and maximum end-to-end delay of multicast trees which failed to satisfy the delay bound were included in the cost and delay measurements. The experiment was run repeatedly until confidence intervals of less than 5%, using 95% confidence level, were achieved for all measured quantities. On the average, 300 different networks were simulated in each experiment in order to reach such confidence levels. At least 250 networks were simulated in each case.

Figure 3.2 shows the percentage increase in total cost of an unconstrained multicast tree relative to optimal versus the multicast group size for two different link loading conditions for 20-node networks. KMB heuristic yields very low tree costs. Note, however, that in the more asymmetric case (figure 3.2(b)) KMB's costs are approximately 10% worse than those for OPT, which is not as good as its performance when applied to symmetric networks [24]. LC does not perform as well as KMB, because it attempts to minimize the cost per path from source to destination, not the total cost of the entire tree. LC's costs are up to 40% worse than OPT. LD yields the most expensive trees, and the cost of the least-delay trees is independent of the range of the link loads. That is why its performance relative to optimal deteriorates as the range of link loads increases. We repeated the same experiment using larger networks. However, OPT could not be applied to networks with more than 20 nodes due to its excessive running times, so we had to measure the percentage increase in



(a) $B_{min} = 45$ Mbps, $B_{max} = 85$ Mbps. (b) $B_{min} = 5$ Mbps, $B_{max} = 125$ Mbps.

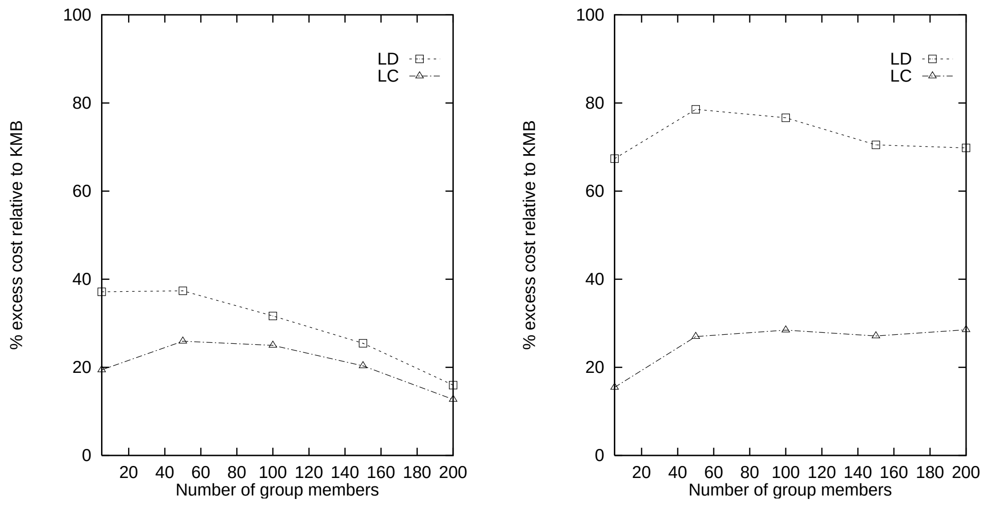
Figure 3.2: Total cost of a multicast tree relative to optimal, unconstrained algorithms, 20 nodes, average degree 4.

the total cost of a multicast tree relative to the total cost of the second best unconstrained algorithm after OPT, namely KMB. Figure 3.3 shows the cost performance of the algorithms when applied to 200-node networks. Comparing this figure with figure 3.2 indicates that the cost performance of the algorithms relative to each other hardly changes as the network size increases.

Figure 3.4 shows the maximum end-to-end delay for 20 and 200-node networks[5]. OPT and KMB perform very poorly with respect to maximum delay, because they do not attempt to minimize the end-to-end delay to the individual destinations. LC results in maximum delays that are in some cases less than 0.03 seconds, which is within the QoS requirement. It finds the least-cost path to each group member. This indirectly minimizes the number of hops for such a path and hence indirectly reduces the length of the path and the delay along that path.

As the number of group members increases, the maximum delays increase, because the multicast trees span more nodes; hence the probability of a remote node being a

[5]It is sufficient to show one case of network loading, because we found that the performance of the different algorithms relative to each other with respect to the maximum end-to-end delay is independent of the range of the link loads.



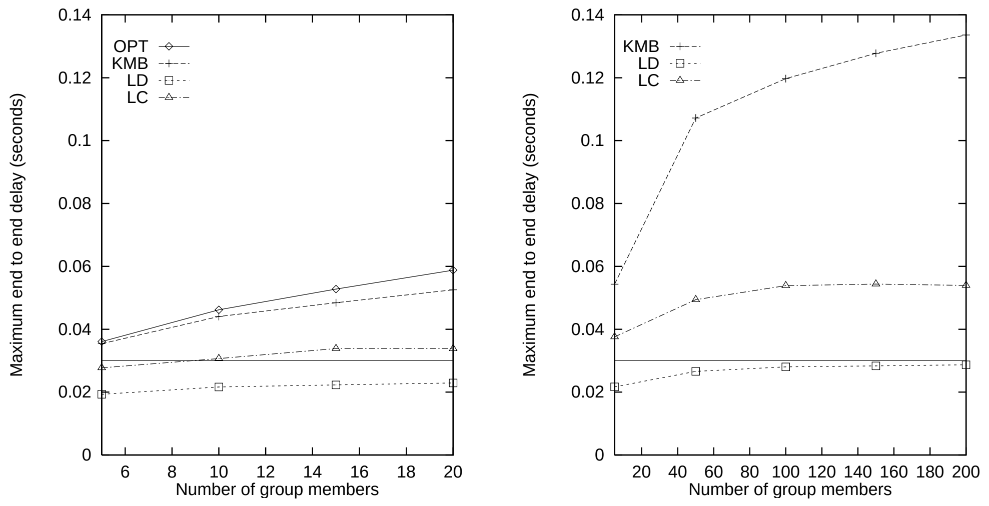
(a) $B_{min} = 45$ Mbps, $B_{max} = 85$ Mbps. (b) $B_{min} = 5$ Mbps, $B_{max} = 125$ Mbps.

Figure 3.3: Total cost of a multicast tree relative to KMB, unconstrained algorithms, 200 nodes, average degree 4.

member in the multicast group is larger. The maximum end-to-end delays increase as the network size increases, because a path connecting two randomly chosen nodes consists of fewer, but longer links (smaller end-to-end delay) in case of small networks, while in case of large networks, this path consists of more shorter links (larger end-to-end delay).

Delay bound violation is one of the reasons to reject a multicast tree. An algorithm's failure to construct a multicast tree due to delay bound violation is strongly related to the maximum delays discussed above. Therefore it is not surprising for OPT, KMB, and even LC to have very high failure rates, $> 30\%$ in case of 20-node networks. LD's failure rate, however, is $< 2\%$ in case of 20-node networks.

Figure 3.5 shows the results of the second experiment, in which we started with a completely unloaded network and kept adding multicast sessions and constructing the corresponding multicast trees until the cumulative tree failure rate exceeded 15%. A multicast session consisted of a random source node generating VBR video traffic with an equivalent bandwidth of 0.5 Mbps, and a multicast group of randomly chosen destination nodes. The experiment was repeated with multicast groups of different



(a) 20 nodes (b) 200 nodes

Figure 3.4: Maximum end-to-end delay, unconstrained algorithms, average degree 4, B_{min} = 5 Mbps, B_{max} = 125 Mbps.

sizes. Failure due to delay bound violation was disabled in this experiment, because the results of the first experiment have shown that the unconstrained algorithms can not satisfy a delay bound of 0.03 seconds. Our objective here was to determine how efficiently these unconstrained algorithms manage the network in the absence of a delay bound. The experiment was repeated, until the confidence interval for the number of successfully established multicast sessions was $< 5\%$ using the 95% confidence level. Similar to the first experiment, in this experiment a random network topology was generated before each run. All algorithms discussed in this section perform routing, admission control, and resource reservation simultaneously. Therefore, these algorithms do not add saturated links to the multicast trees being constructed. Instead, they search for alternate links or paths that are not yet saturated. This experiment could not be applied to the optimal minimum Steiner tree algorithm, OPT, because of its large execution time.

It is obvious from figure 3.5 that, as the size of the multicast group increases, the number of multicast trees that an algorithm can construct before the network saturates decreases. This is because the size of a multicast tree increases as the

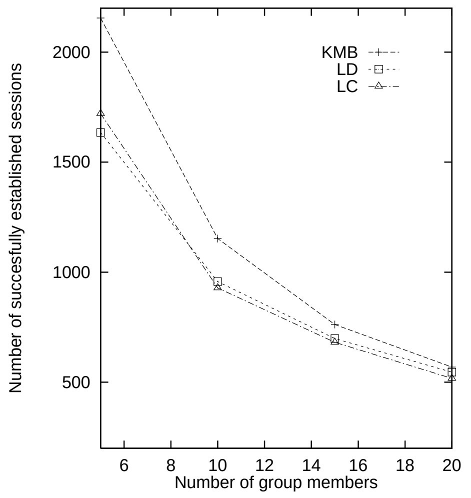


Figure 3.5: Number of successful sessions, unconstrained algorithms, 20 nodes, average degree 4, no delay constraint.

group size increases. KMB yields the best performance, because it has the ability to locate the lowest cost links in the network and include them in the multicast tree. This results in approximately uniform link load distribution across the network throughout the experiment. LD and LC can also manage the network resources efficiently, although not as efficiently as KMB. LD's performance is no more than 20% worse than KMB's performance. Combining admission control and resource reservation with routing allows LD to find alternate routes when links on the absolute least-delay paths are saturated. That is why LD performs as good as LC although it is not as efficient as LC in constructing low-cost trees.

The first experiment shows that the unconstrained algorithms (OPT, KMB, and LC) are not satisfactory for applications having delay constraints. Therefore, such algorithms will not be suitable for future high-speed networks. LD is optimal with respect to minimizing delays but it does not attempt to optimize the tree cost at all. The second experiment shows that all algorithms discussed so far are efficient network managers with KMB being the best. We will study the constrained algorithms in section 3.2.3 to determine if they can achieve a compromise between the unconstrained cost-oriented algorithms (OPT, KMB, and LC) and the delay-oriented algorithms (LD), but first we will study the performance of one more unconstrained multicast

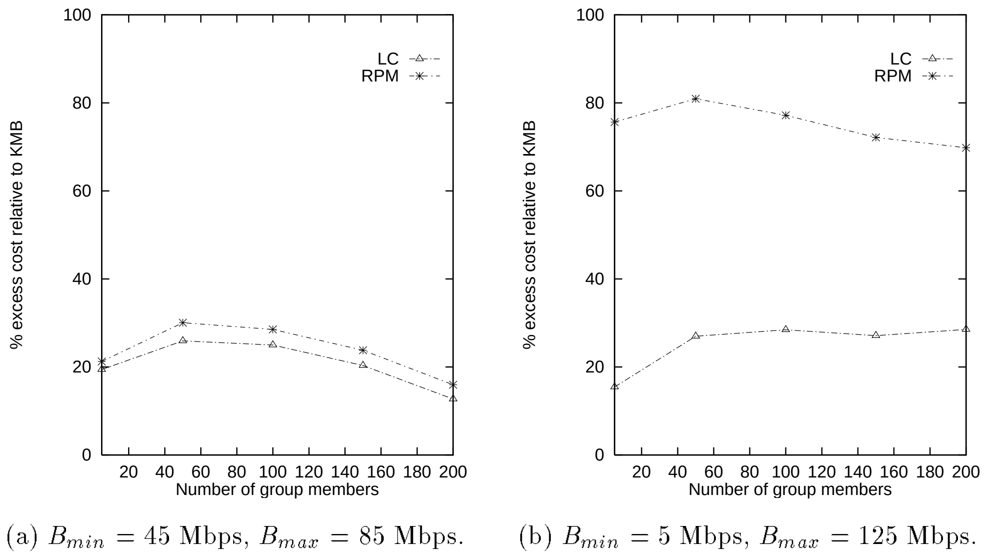


Figure 3.6: Total cost of a multicast tree relative to KMB, LC and RPM, 200 nodes, average degree 4.

routing algorithm, RPM.

3.2.2 RPM's Performance

We pointed out in section 2.4.1 that RPM generates reverse shortest path trees, i.e., trees in which the reverse paths from each destination back to the source are least-cost. If the link costs are symmetric, the costs of the resulting forward paths from the source to the destinations will also be least-cost, and RPM will construct exactly the same trees as the LC shortest path algorithm. In this section, we compare RPM's reverse shortest path trees to LC's forward shortest path trees. First we show the results of experiment 1 described in the previous section for both algorithms.

Figure 3.6 shows the percentage increase in total cost of RPM and LC generated multicast trees relative to KMB for 200-node networks. When the asymmetry of the network is small, RPM's costs are only slightly more than LC's costs. As the range of the link loads increases and hence the asymmetry of the network increases, however, the costs of RPM's trees do not change, while LC is capable of finding much lower cost trees. The figure shows that RPM is less than 30% worse than KMB when the

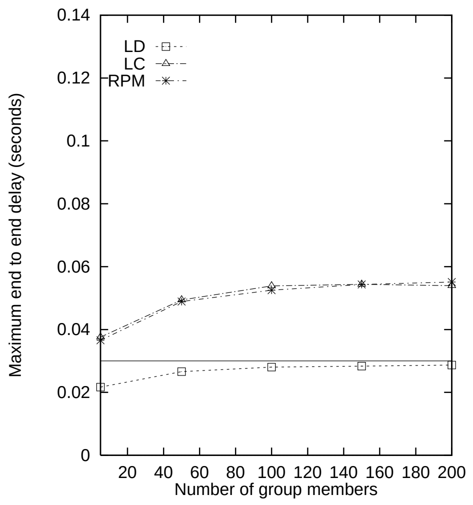


Figure 3.7: Maximum end-to-end delay, LC and RPM, 200 nodes, average degree 4, $B_{min} = 5$ Mbps, $B_{max} = 125$ Mbps.

network asymmetry is small. When the network asymmetry is large, however, RPM is up to 80% worse than KMB as can be seen from figure 3.6(b). Thus it is obvious that RPM's approach of using the reverse link's cost as an estimate of the forward link's cost is ineffective for asymmetric networks.

The end-to-end delays on RPM's multicast trees are the same as those on LC's multicast trees as can be seen from figure 3.7. This is because the average lengths of the reverse shortest paths and the forward shortest paths are equal, and thus propagation delays are equal in both directions.

RPM is used in practice [63] because it requires only limited information to be available at each node in order to construct a reverse shortest paths multicast tree. Current implementations of RPM do not perform routing, resource reservation and admission control at the same stage. Currently, separate resource reservation protocols (see e.g. [84, 85]) are applied to perform admission control tests and reserve resources on the multicast trees constructed by the routing protocols. If resource reservation fails due to the existence of saturated links in a multicast tree, then the multicast session can not be established, because none of the existing RPM-based routing protocols is capable of finding alternate links or paths to replace the saturated links. We implemented a version of RPM that separates routing from resource reservation and admission control to imitate the situation just described. We also implemented another version that performs routing, resource reservation, and admission control simultaneously. We call these two algorithms RPM_SEP and RPM_COMB[6] respectively[7]. Similarly, we implemented two versions of the shortest path algorithm, LC. The first one, LC_SEP, separates routing from resource reservation and admission control, while the second version, LC_COMB, combines routing with resource reservation and admission control. We ran experiment 2 of the previous section on these four algorithms to determine RPM's efficiency in managing the available link bandwidth and to examine the effect of separating routing from resource reservation and admission control.

Figure 3.8 shows the number of successfully established sessions versus the multicast group size for 20-node networks. Both versions of LC yield good performance, because the routing part of LC always uses low cost links to construct the multicast trees. Therefore, the load on the links increases gradually, and the difference between the minimum link load and the maximum link load at any time is small. Thus, the algorithm is capable of constructing a large number of trees before any links saturate and admission control comes into play. When links start to saturate, LC_COMB is capable of using alternate paths when it fails to add a saturated link to a tree. That's why LC_COMB's performance is better than LC_SEP's performance. RPM_SEP is very inefficient even for small group sizes. As has been mentioned before, RPM_SEP adds a link $e = (u, v)$ to an multicast tree based on the cost of the reverse link $e' = (v, u)$. If e' is lightly utilized and remains lightly utilized, RPM will keep adding sessions to e, and it will not receive any indication that e is heavily utilized. This leads to extremely asymmetric link loads. Even when the forward link e saturates, RPM_SEP will not be notified, and it will still attempt to construct multicast trees containing e. Such trees will be later rejected by admission control. Applying RPM_SEP results in extremely inefficient management of the network bandwidth. RPM_COMB causes very asymmetric link loads, similar to RPM_SEP, but the close interaction between routing and admission control enables it to find alternate paths

[6]RPM_COMB will probably require more information to be kept at each node than RPM_SEP, if it is to be implemented in practice.

[7]Both algorithms give identical results when applied to experiment 1, because in that experiment resources are always available.

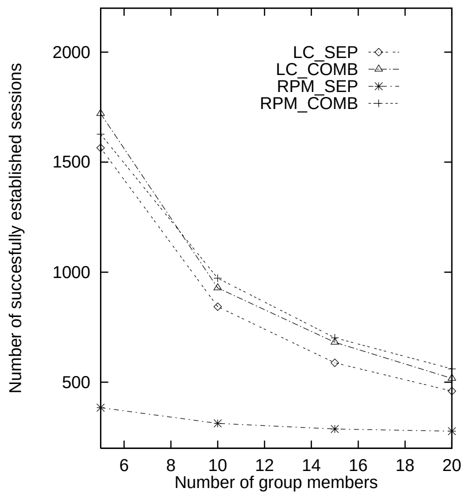


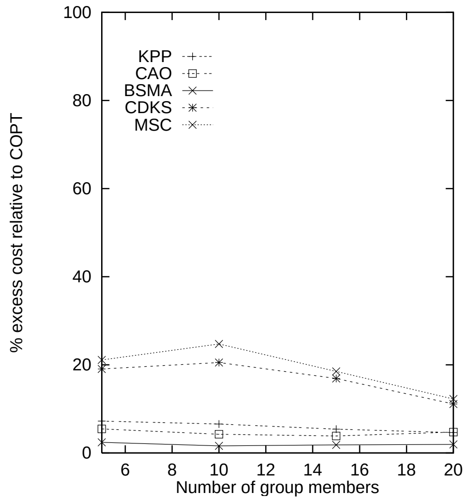
Figure 3.8: Number of successful sessions, LC and RPM, 20 nodes, average degree 4, no delay constraint.

to replace saturated links. This improves RPM_COMB's efficiency to the extent that it performs as good as LC_COMB. Thus combining resource reservation and admission control with routing leads to a much more efficient management of the available resources in case of RPM.

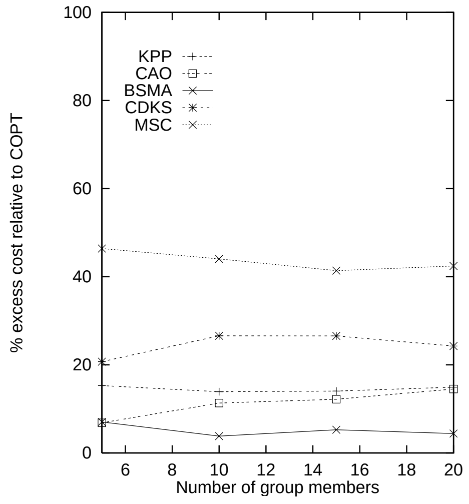
3.2.3 Simulation Results for the Delay-Constrained Algorithms

The results given in sections 3.2.1 and 3.2.2 show that the unconstrained multicast routing algorithms are not capable of providing satisfactory service to applications with delay constraints. In this section we study the performance of the delay-constrained algorithms to determine the characteristics of the multicast trees they construct.

We re-ran the same first experiment of section 3.2.1 on COPT, CDKS, the three delay-constrained minimum Steiner tree heuristics, and MSC. Figure 3.9 shows the percentage increase in the total cost of a delay-constrained (or semiconstrained) multicast tree relative to the total cost of COPT in case of 20-node networks. When the range of the link loads is larger, the difference in performance between the algorithms is larger. BSMA yields better cost performance than KPP and CAO. For the scenarios



(a) $B_{min} = 45$ Mbps, $B_{max} = 85$ Mbps

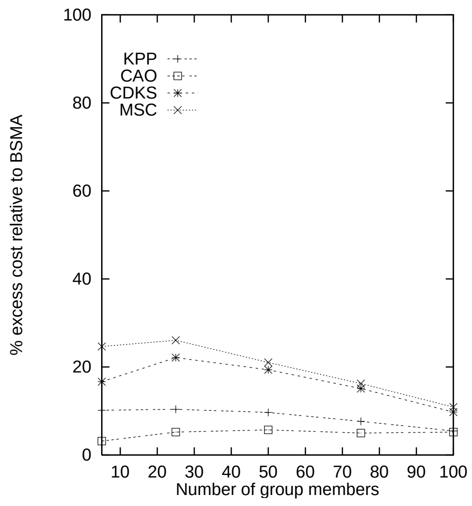


(b) $B_{min} = 5$ Mbps, $B_{max} = 125$ Mbps

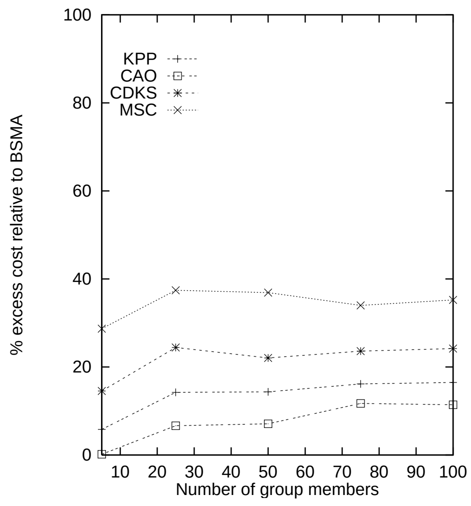
Figure 3.9: Total cost of a multicast tree relative to COPT, delay-constrained algorithms, 20 nodes, average degree 4, $\Delta = 0.03$ seconds.

we simulated, BSMA's tree costs are less than 7% more expensive than optimal, while CAO and KPP's trees are up to 15% more expensive than optimal. CDKS and MSC generate trees that are always more expensive than the trees of the delay-constrained minimum Steiner tree heuristics. CDKS's and MSC's costs are up to 25% and 45%, respectively, more than those for COPT when the range of link loads is large. MSC's internally generated delay bound is so strict for the cases we studied that it restricts the algorithm's ability to minimize the tree costs. We repeated the same experiment using networks with more than 20 nodes, but we could not apply COPT to these networks due to its excessive execution times. The percentage increase in the total costs of the delay-constrained algorithms relative to those for BSMA (the second best delay-constrained algorithm after COPT) when applied to 100-node networks[8] are shown in figure 3.10. It is evident from figures 3.9 and 3.10 that the performance of the different algorithms relative to each other remains unchanged as the size of the network increases.

[8]We could not apply BSMA to networks with more than 100 nodes, because its execution time becomes too large as will be shown in the next section.



(a) $B_{min} = 45$ Mbps, $B_{max} = 85$ Mbps

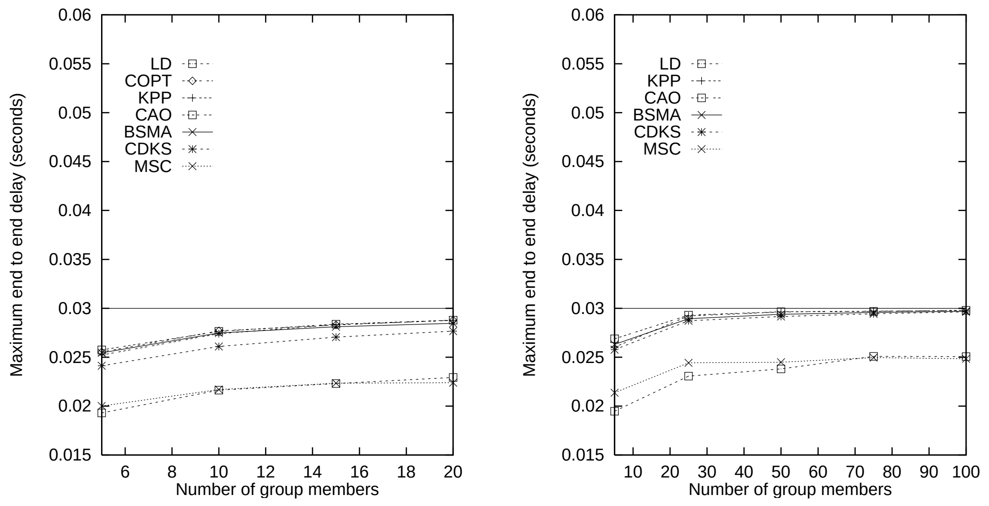


(b) $B_{min} = 5$ Mbps, $B_{max} = 125$ Mbps

Figure 3.10: Total cost of a multicast tree relative to BSMA, delay-constrained algorithms, 100 nodes, average degree 4, $\Delta = 0.03$ seconds.

It can be seen from figure 3.11 that the maximum end-to-end delays for the delay-constrained algorithms are below the 0.03 seconds delay bound. All delay-constrained algorithms yield similar delay performance, but CDKS is slightly better. This is because CDKS constructs a LC tree and then replaces paths which violate the delay bound with paths from the LD tree. Figure 3.11 also shows the maximum delays of LD for comparison. LD's maximum delays are considerably less than the maximum delays the delay-constrained algorithms can achieve. MSC's maximum delays are comparable to LD's maximum delays. However, this is not a big advantage, because it is sufficient to simply satisfy the delay constraint.

We found that the granularity we chose for KPP ($\Delta/10 = 0.003$ seconds) is sufficient in the case of 20-node networks. In that case, KPP's success rate in constructing a delay-constrained multicast tree is almost as high as the optimal success rate achieved by COPT and LD. We noticed however that KPP's success rate in constructing a delay-constrained tree is up to 5% worse than optimal for 100-node networks. This is because as the number of nodes increases within the same area, the average link delay decreases. For 100-node networks the average link delay is



(a) 20 nodes (b) 100 nodes

Figure 3.11: Maximum end-to-end delay, delay-constrained algorithms, average degree 4, $B_{min} = 5$ Mbps, $B_{max} = 125$ Mbps, $\Delta = 0.03$ seconds.

small and comparable to KPP's granularity, which affects the heuristic's accuracy and hence its success rate.

We conducted the second experiment of section 3.2.1 on the delay-constrained heuristics to evaluate their efficiency in managing the network bandwidth. The experiment was first modified, however, to permit failure due to delay bound violation. An algorithm can thus fail to construct a multicast tree due to either violating the delay bound or due to link saturation. We also conducted this modified experiment on LD. We could not run it on COPT, however, because of its large execution time. Figure 3.12 shows that the three delay-constrained minimum Steiner tree heuristics yield similar performance and that they can manage the network bandwidth efficiently, with BSMA being the best. CDKS, LD and MSC are not as efficient as the delay-constrained minimum Steiner tree heuristics.

In summary, all delay-constrained algorithms, and the semiconstrained algorithm, can meet the delay requirements of real-time applications, and thus they would all be suitable for high-speed networks. What differentiates them, of course, is their costs and execution times, which we study next.

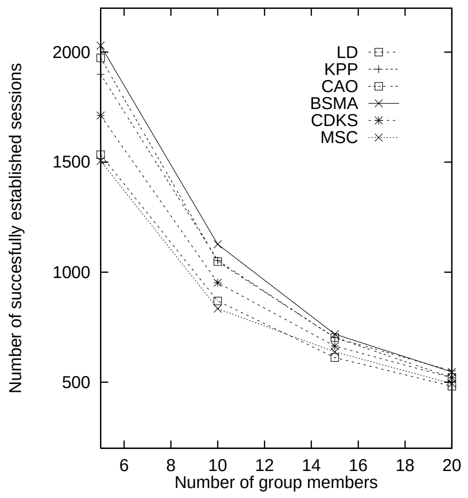
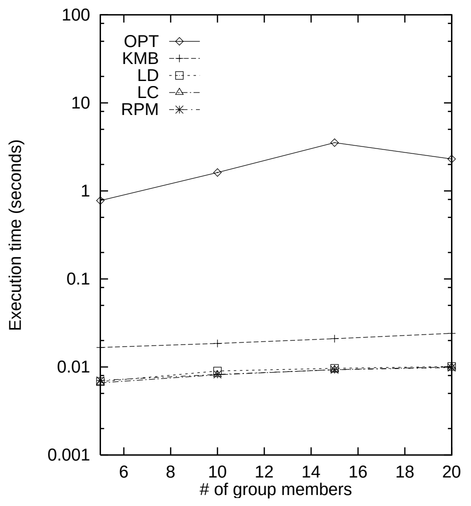


Figure 3.12: Number of successful sessions, delay-constrained algorithms, 20 nodes, average degree 4, $\Delta = 0.03$ seconds.

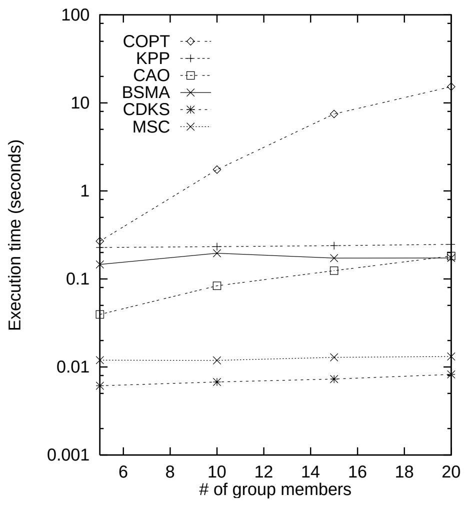
3.2.4 Execution Times

Figures 3.13, 3.14, and 3.15 show the average execution times of all algorithms studied in this paper. These execution times were measured when running the first experiment discussed in the previous sections. Note, however, that the code used for the algorithms was not optimized for speed. Figures 3.13 and 3.14 show the execution times for 20-node and 100-node networks with variable multicast group sizes, while figure 3.15 shows the growth of the execution times with the network size for a fixed multicast group of 5 members. The running times of OPT and COPT are very large as can be seen from figure 3.13. RPM[9], in spite of its poor performance, is the fastest algorithm. The running times of the delay-constrained minimum Steiner tree heuristics are large, except for CAO's running time for small group sizes. CAO's running time increases almost linearly as the group size increases, because it runs the constrained Bellman-Ford algorithm once for each group member not already in the tree. Note from figure 3.15(b) that CAO's growth rate is slower than MSC's and CDKS's growth rates. This holds for small multicast group sizes only. BSMA's running time is very large for 100-node networks as shown in figure 3.14(b). It is particularly slow

[9]We implemented a centralized version of RPM.



(a) Unconstrained algorithms

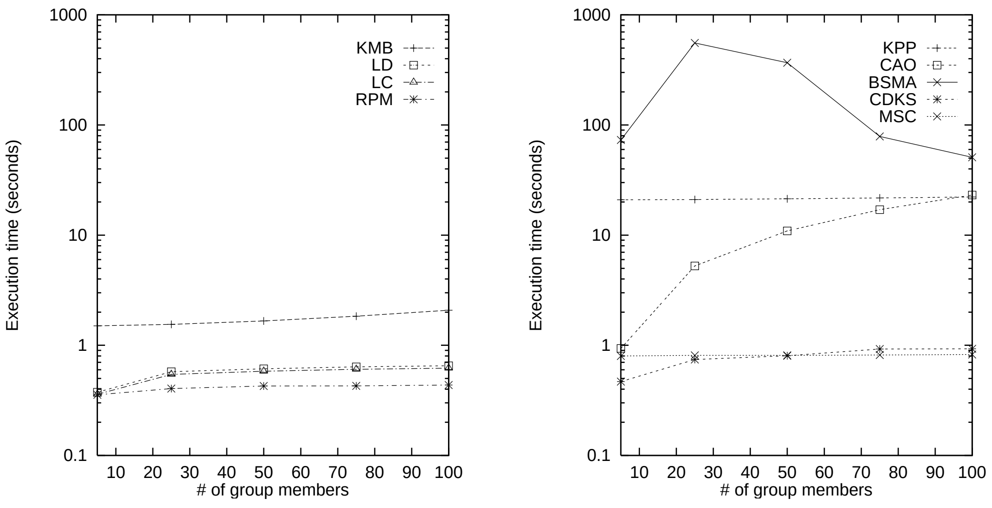


(b) Constrained algorithms

Figure 3.13: Execution times, 20 nodes, average degree 4, variable multicast group size, $B_{min} = 5$ Mbps, $B_{max} = 125$ Mbps, $\Delta = 0.03$ seconds.

for multicast groups of medium size. For small multicast groups, the number of superedges in a multicast tree is small, and BSMA does not have to apply the time consuming kth-shortest path algorithm many times. For large multicast groups, the kth-shortest path algorithm is fast because the number of nodes outside the initial tree is small, and thus the number of possible alternate paths to replace a superedge is small. For medium size multicast groups, however, the number of superedges is large and at the same time the number of nodes outside the tree is also large, which leads to the long running times of BSMA. CDKS and MSC are as fast as the unconstrained algorithms. Therefore, the three delay-constrained minimum Steiner tree heuristics may be too slow, in spite of their high efficiency, to use on networks with thousands of nodes, while the less efficient but much faster CDKS and MSC may be better suited for large networks.

The simulation results presented in this section clearly indicate the need for delay-constrained multicast routing algorithms. The delay-constrained minimum Steiner tree heuristics we simulated are quite efficient in managing the network resources, but they have large execution times. On the other hand, CDKS and MSC have fast



(a) Unconstrained algorithms (b) Constrained algorithms

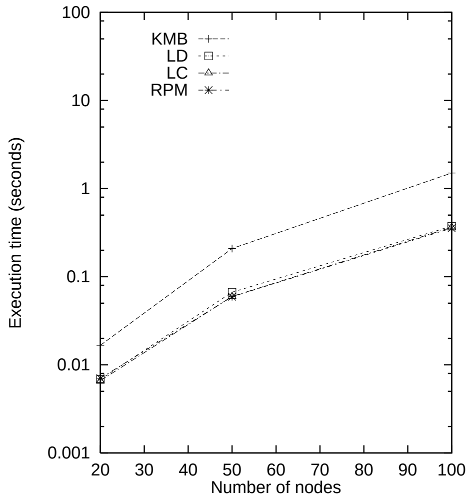
Figure 3.14: Execution times, 100 nodes, average degree 4, variable multicast group size, $B_{min} = 5$ Mbps, $B_{max} = 125$ Mbps, $\Delta = 0.03$ seconds.

execution times, but they construct multicast trees that are more expensive than the multicast trees constructed by the delay-constrained minimum Steiner tree heuristics.

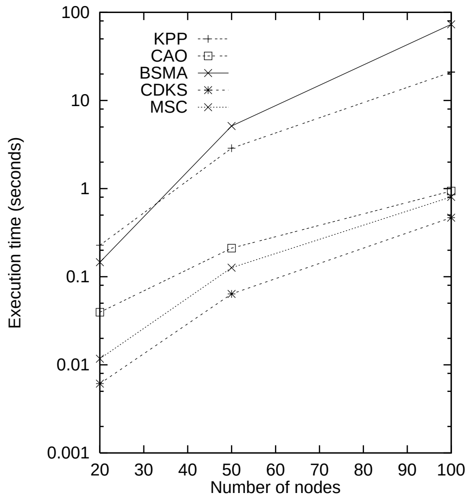
3.3 Conclusions

Distributed real-time applications have QoS requirements that must be guaranteed by the underlying network. In many cases, these applications will involve multiple users. Hence the increasing importance of multicasting. Multicast routing can be an effective tool to manage the network resources and fulfill the applications' requirements. Several multicast routing algorithms are proposed for high-speed networks carrying real-time traffic. Our work is the first detailed, quantitative evaluation of these algorithms under realistic high-speed networking environments.

We studied the performance of unconstrained multicast routing algorithms when applied to wide-area networks with asymmetric link loads. The KMB heuristic constructs low cost trees with large end-to-end delays that exceed the upper bound on delay imposed by the application. LC is also unable to satisfy the required delay



(a) Unconstrained algorithms



(b) Constrained algorithms

Figure 3.15: Execution times, Variable network size, average degree 4, 5 multicast group members, $B_{min} = 5$ Mbps, $B_{max} = 125$ Mbps, $\Delta = 0.03$ seconds.

bound. KMB is more efficient in managing the network bandwidth, than LC.

RPM performs poorly when applied to networks with asymmetric link loads. It creates expensive multicast trees and is very inefficient in managing the network bandwidth, because it results in very asymmetric link loads. Current implementations of RPM do not contain a resource reservation and admission control module. Resource reservation is a separate protocol that has minimal interaction with routing, and thus it is currently not possible to select alternate paths to replace any saturated links in the multicast tree when they get rejected by admission control. We have shown that incorporating RPM together with admission control and resource reservation in a single module dramatically improves RPM's efficiency in managing the available network bandwidth. Simulation results have also shown that, similar to the other unconstrained algorithms, RPM is not capable of satisfying the delay bounds imposed by real-time applications. Note, however that RPM is a fast distributed dynamic algorithm. Therefore, it is the simplest to implement and maintain among all algorithms studied in this chapter.

We concluded that the unconstrained multicast routing algorithms, KMB, LC, and RPM, can not be applied to real-time applications on networks spanning large areas. Then, we studied a semiconstrained algorithm and four delay-constrained algorithms: three delay-constrained minimum Steiner tree heuristics and one delay-constrained shortest path heuristic. All three delay-constrained minimum Steiner tree heuristics construct low cost trees, which satisfy the given delay bound and can manage the network resources efficiently, but BSMA is the best. The execution times of the delay-constrained minimum Steiner tree heuristics differ considerably. As the networks size increases, BSMA's average execution time grows much faster than KPP's and CAO's average execution times. CAO has fast execution times in case of small group sizes. Note however, that execution times of both KPP and CAO grow exponentially with the size of the network in the worst case, while the worst case execution times of BSMA are polynomial in the size of the network. Overall, we conclude that all three delay-constrained minimum Steiner tree heuristics, though efficient, are too complex to be applied on any real wide-area networks.

The delay-constrained shortest path heuristic, CDKS, does not perform as good as the three delay-constrained minimum Steiner tree heuristics. The semiconstrained algorithm, MSC, is always capable of constructing delay-constrained multicast trees in the scenarios we studied. However, using a strict internally computed delay bound limits MSC's ability to construct low cost multicast trees. Both CDKS and MSC have fast execution times and scale well to large network sizes.

The need for delay-constrained algorithms is evident from the experiments presented in this chapter. We suggest that any future work on delay-constrained multicast routing be focused on simple, fast algorithms similar to CDKS and MSC. Efficient, distributed, scalable implementations of such algorithms must be proposed in order for them to have the potential of being adopted by future multicast routing protocols. Allowing nodes to join/leave an existing multicast tree dynamically is another feature that should be considered in future implementations of delay-constrained multicast routing algorithms.

Finally, we are confident that the results of our simulation experiments are an indication of the expected performance of the studied algorithms when applied in real networks. However, the values we measured in our experiments are dependent on the networking environment we adopted: the size of the network, the range of the link costs, the distribution of the link costs within that range, ..., etc. Therefore the measured quantities can not be generalized to all networking environments. For example, we can not conclude from figure 3.9 that BSMA's percentage excess tree cost relative to that of COPT never exceeds 7% irrespective of the networking environment. This value may exceed 7% if, for example, a different link cost distribution is used.

Chapter 4

Delay-Constrained Broadcast Routing Problem

Broadcasting is the special case of multicasting when all nodes in the network are members of a given multicast group. The *NP*-complete unconstrained minimum Steiner tree problem used for multicast routing reduces to the less complex unconstrained minimum spanning tree problem in case of broadcast routing. Several optimal polynomial time algorithms exist for the unconstrained minimum spanning tree problem, e.g., [8, 86]. In addition, distributed algorithms for the same problem have been proposed, e.g., [87, 88]. However, the problem of constructing delay-constrained minimum spanning trees for broadcast routing purposes has not been studied previously.

In the future, many real-time applications will involve all nodes in a given network. Some distributed real-time control applications and the broadcasting of critical network state information are just a few examples. For such applications, delay-constrained minimum spanning trees are needed to broadcast the real-time traffic from the source node to all other nodes in the network. The existing delay-constrained minimum Steiner tree heuristics, which have been surveyed in chapter 2 and evaluated in chapter 3, can be used to construct delay-constrained broadcast trees, but, as we concluded at the end of chapter 3, most of them are too complex. In this chapter, we prove that the delay-constrained minimum spanning tree problem is *NP*-complete. However, we expect heuristics specifically designed for that problem to be less complex than the existing delay-constrained minimum Steiner tree heuristics.

In this chapter, we propose a delay-constrained minimum spanning tree heuristic for asymmetric networks. We call it the bounded-delay broadcast heuristic, **BDB**. We evaluate the performance of BDB in comparison to those of COPT, BSMA, and the distributed delay-constrained multicast tree heuristic, **DMCT** [40]. COPT is the same optimal algorithm used in the experiments of chapter 3. We include BSMA in the comparison, because it is the best existing delay-constrained minimum Steiner tree heuristic with respect to tree cost. DMCT is also included in the comparison, because it is a fast distributed delay-constrained minimum Steiner tree heuristic. Both BDB and DMCT are based on Prim's unconstrained minimum spanning tree algorithm [8], but DMCT was designed for symmetric networks. We will show that BDB outperforms DMCT, not only in the case of asymmetric networks, but also in the special case of symmetric networks.

This chapter is organized as follows. In section 4.1, we formulate the delay-constrained minimum spanning tree problem in asymmetric networks and prove that it is *NP*-complete. In section 4.2, we present our heuristic for solving the problem. Then in section 4.3, we evaluate the average performance of the heuristic and compare it to COPT, BSMA, and DMCT using simulation. Finally, we present our conclusions in section 4.4.

4.1 Problem Formulation

In this chapter, we use the same definitions presented in section 2.3, but we set $G = V$ to represent broadcast groups. It is sufficient to identify a spanning tree by its source since the members of a broadcast group are always fixed. Therefore, $T(s) \subseteq E$ is rooted at a source node $s \in V$ and contains a path from s to any node $v \in (V - \{s\})$.

The delay-constrained minimum spanning tree problem, DCMST, in directed networks constructs the spanning tree $T(s)$ rooted at s that has minimum total cost among all possible spanning trees rooted at s which have a maximum end-to-end delay less than or equal to a given delay constraint Δ.

Theorem 4.1 *DCMST is NP-complete.*

Proof. For the purpose of this proof, we formulate the DCMST problem as a decision problem as follows.

DCMST Decision Problem: *Given a directed network $N = (V, E)$, a nonnegative cost $C(e)$ for each $e \in E$, a nonnegative delay $D(e)$ for each $e \in E$, a source node $s \in V$, a positive delay constraint Δ, and a positive value B, is there a spanning tree $T(s)$ that satisfies:*

$$Cost(T(s)) \leq B, \tag{4.1}$$

$$Max\_Delay(T(s)) \leq \Delta? \tag{4.2}$$

Clearly DCMST is in *NP*, because a nondeterministic algorithm can guess a set of links to form the tree, then it is possible to verify in polynomial time that these links do form a tree, that this tree spans all nodes in the network, that the total cost of the tree is less than B, and that the maximum end-to-end delay from the source node s to any node $v \in V$ is $\leq \Delta$.

The next step is to transform a known *NP*-complete problem to DCMST. We will use the Exact Cover by 3-Sets (X3C) problem which has been shown to be *NP*-complete in [33]. It can be stated as follows.

X3C Decision Problem: *Given a finite set $X = \{x_1, \ldots, x_{3p}\}$ and a collection $Y = \{y_1, \ldots, y_q\}$, $q \geq p$, of 3-element subsets of X, is there a subcollection $Y' \subseteq Y$ such that every element of X occurs in exactly one member of Y', i.e., the members of Y' are pairwise disjoint, and $\bigcup_{y \in Y'} y = X$?*

Given an arbitrary instance of X3C, we construct the network $N = (V, E)$ for the corresponding instance of DCMST as follows:

- Every element of X is represented by a node in the network, and also every member of Y is represented by a node. Two additional nodes are introduced: a source node s and another node t. Therefore:

$$V = \{s\} \cup \{t\} \cup X \cup Y \tag{4.3}$$

- Add the following set of directed links E to interconnect the nodes:

$$\begin{aligned} E &= (s,t) \cup \{(s,y_i) : i = 1,\ldots,q\} \cup \{(t,y_i) : i = 1,\ldots,q\} \cup \\ &\quad \{(y_i,x_j) : x_j \in y_i, i = 1,\ldots,q, j = 1,\ldots,3p\} \end{aligned} \tag{4.4}$$

- Assign the following costs to the links:

$$C(e) = \begin{cases} 3, & e \in \{(s,y_i) : i = 1,\ldots,q\}, \\ 1, & otherwise. \end{cases} \tag{4.5}$$

- Unit delay is assigned to all links:

$$D(e) = 1, \quad \forall e \in E \tag{4.6}$$

- Set the delay constraint Δ of DCMST to:

$$\Delta = 2 \tag{4.7}$$

- Set the positive value B to:

$$B = 5p + q + 1 \tag{4.8}$$

Figure 4.1 illustrates this transformation. It is clear that it can be done in polynomial time. The final step is to show that a feasible spanning tree exists for the above instance of DCMST if and only if the 3-set collection Y has an exact cover Y'. If for a given instance of X3C, any element $x \in X$ is not in any member y of the collection Y then that instance does not have an exact cover. The above transformation, when applied to such an instance, will result in an unconnected network; thus no spanning tree can be constructed.

If, however, for a given instance of X3C, each element $x \in X$ appears in at least one member y of Y, the resulting network will be connected. Since all link delays are set to 1 and the delay constraint Δ is set to 2, the number of hops along any path starting at s in any feasible solution $T(s)$ can not exceed 2. Each node x_i will be attached to some node $y_j \in Y$ via a unit cost link. As a result of the 2- hop constraint, the y_js used to reach the x_is must be directly attached to s via the expensive links of cost 3. Lower cost but longer paths, via the node t, can be used from s to the y_js

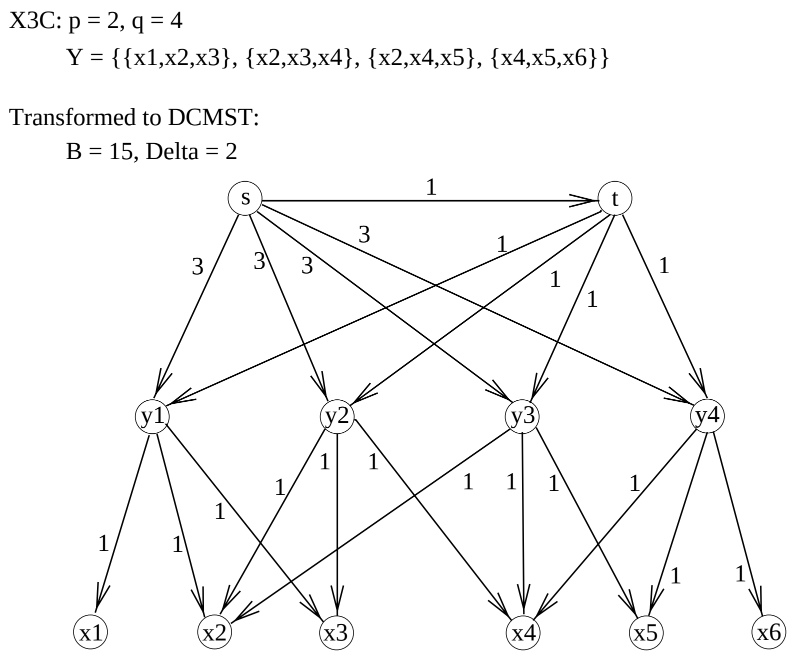


Figure 4.1: Equivalent instances of X3C and DCMST. Link costs are shown. All link delays are set to 1 (not shown).

which are not used to reach the x_is. Let there be m nonleaf y_j nodes (y_js that are used to reach the x_is) and n leaf y_j nodes, where $q = m + n$. Thus the total cost of any spanning tree, $T(s)_{sat\_\Delta}$, that satisfies the delay constraint Δ is:

$$\begin{aligned} Cost(T(s)_{sat\_\Delta}) &= \overbrace{3p * 1}^{x_i s} + \overbrace{m * 3}^{\text{nonleaf } y_j s} + \overbrace{n * 1}^{\text{leaf } y_j s} + \overbrace{1 * 1}^{t} \\ &= 3p + 3m + (q - m) + 1 = 3p + 2m + q + 1 \qquad (4.9) \end{aligned}$$

Each component in equation 4.9 represents the cost of the links used to attach a set of nodes, indicated as the label of that component, upstream towards s. From equations 4.1, 4.8 and 4.9 it follows that the condition

$$m \leq p \qquad (4.10)$$

must be satisfied for the total cost of the tree to be less than B. On the other hand, each of the m nonleaf y_js can be used to reach at most 3 x_is. Therefore

$$m \geq p \qquad (4.11)$$

nonleaf y_j nodes are needed to reach the $3p$ x_is. Combining conditions 4.10 and 4.11, we get

$$m = p \tag{4.12}$$

as the only possible number of nonleaf y_js that results in a feasible solution to that instance of DCMST. For such a feasible solution to exist, each of the p nonleaf y_js must be used to reach exactly 3 different x_is, or, in terms of the X3C problem, the members of the collection Y corresponding to the p nonleaf y_j nodes must be pairwise disjoint. p pairwise disjoint 3-sets cover $3p$ elements, and thus form an exact cover of the set X. This means that the existence of a feasible solution of an instance of DCMST implies the existence of a feasible solution for the corresponding instance of X3C.

Conversely, if an instance of X3C has an exact cover of X, that exact cover $Y' \subseteq Y$ must have exactly $|Y'| = p$ members to cover the $3p$ elements of X. The corresponding instance of DCMST will have a feasible solution with $|Y'|$ nonleaf y_js, a maximum end-to-end delay of $2 = \Delta$, and a total cost of $3p + |Y'| * 3 + |Y - Y'| * 1 + 1 = 3p + 3p + q - p + 1 = 5p + q + 1 = B$. This completes the proof. $\square$

The DCMST problem remains *NP*-hard even if all link delays are equal. When all link costs are equal, the solution to the polynomial time least-delay tree problem can be used to answer the DCMST decision problem. In the next section we propose a simple and efficient heuristic for the DCMST problem to avoid the exponentially growing execution times of the optimal solutions.

4.2 The Bounded-Delay Broadcasting Heuristic (BDB)

The BDB algorithm consists of two phases; phase 1 is executed once, followed by phase 2, which is also executed once. The result of phase 1 is a moderate-cost spanning tree which satisfies the delay constraint. Phase 2 attempts to replace expensive links in this tree with lower cost links, without violating the delay constraint, and without introducing loops. Phase 1 of BDB is outlined below:

```
1. Given $N = (V, E)$, a source node $s$, all link costs, and all link delays. The initial
   subtree contains the source $s$ only.
2. Repeat {
3.     Find the least cost link that connects an unconnected node to the already
       constructed subtree without violating the delay constraint. Add that
       link to the subtree.
4.     If step 3 can not find any link, then select a link $e = (u, v)$ which is not
       included in the subtree, and $u$ and $v$ are already in the subtree. The link $e$
       is chosen such that:
       • Replacing the link $l = (w, v)$ (this is the subtree link currently used
         to connect $v$ upstream towards $s$) with the selected link $e$ maximizes
         the delay relaxation.
       If no link that achieves positive delay relaxation can be found, the heuristic
       fails and exits, else remove $l$ and add $e$ to the subtree.
5. } until the tree spans all nodes.
```

The term delay relaxation can be defined as follows. Given are two alternate paths, P_1 and P_2, to connect the source s to some node $n \in V$, and $Delay(P_2) < Delay(P_1)$. If P_1 is the currently used path from s to n, the delay from s to n can be reduced by using the path P_2 instead of P_1. Therefore, the delay relaxation at n is the difference between $Delay(P_2)$ and $Delay(P_1)$. Phase 1 of BDB is based on Prim's unconstrained minimum spanning tree algorithm. It starts with a subtree containing the source node s only, and adds one node at a time to the subtree, without violating the delay constraint until the subtree spans all nodes in the network. If at any point during the first phase, the heuristic can not find any node that can be added without violating the delay constraint, it resorts to delay relaxation. BDB relaxes the delays by choosing a node n that is already in the subtree and replaces the subtree link e_{before} connecting it upstream towards s with another link e_{after} such that n remains connected to s and the end-to-end delay from s to n is reduced. The node n is chosen such that the delay relaxation is maximized. If no more positive delay relaxation can be achieved and BDB fails to add any remaining unconnected nodes to the subtree, the algorithm fails. It is guaranteed, however, that the first phase of BDB will find a delay-constrained tree spanning all nodes if one exists, because in the worst case the heuristic keeps applying the delay relaxation step, and thus reducing the end-to-end delays to the individual nodes until it ends up with the least-delay tree. If the least-delay tree can not span all nodes in the network without violating the delay constraint, no other tree can. The second phase of BDB takes the tree constructed by the first phase and attempts to reduce its cost. It is outlined below.

```
1. Given $N = (V, E)$, a source node $s$, all link costs, all link delays, and an initial
   delay-constrained tree spanning all nodes in $V$.
2. Repeat {
3.    Find the least cost link $e = (u, v)$, $v \neq s$, which is not included in the
      subtree, and which satisfies the following conditions.
      If adding $e$ to the tree does not create a loop:
        • $C(e) < C(l)$ where $l = (w, v)$ is the tree link currently used to connect
          $v$ upstream towards $s$.
        • Replacing $l$ with $e$ does not cause any delay constraint violations along
          the tree.
   else:
        • Call the loop breaking algorithm, LBA. LBA attempts to break the
          loop such that the cost of the resulting tree is less than the cost prior
          to creating the loop and that the resulting tree does not violate the
          delay constraint. If LBA returns successfully, memorize the values
          returned for $l_{remove}$ and $l_{add}$ (see LBA code). If LBA fails, the loop
          created by adding $e$ can not be broken. Thus $e$ can not be used to
          reduce the cost of the tree.
4.    If no such link can be found, the heuristic can not achieve further cost
      reduction and stops, else:
        • Remove $l$ and add $e$ to the tree.
        • If $e$ creates a loop, break the loop by removing the link $l_{remove}$ from
          the tree and replacing it with the link $l_{add}$.
5. } until no further cost reduction can be achieved.
```

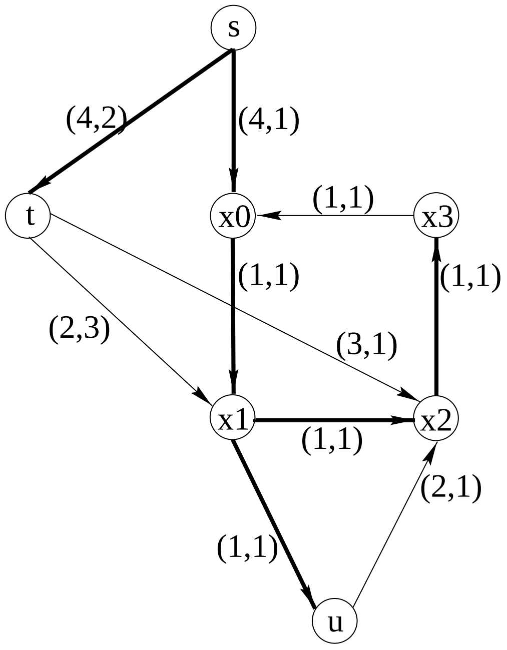
In phase 2, expensive tree links are removed and replaced with lower cost links. The algorithm scans all links not already in the tree, and chooses the least-cost link,

$e = (u, v)$ such that $C(e) < C(l)$, where $l = (w, v)$ is the link currently used to connect the node v upstream towards s. Another condition for choosing the link e is that using it to replace l in the tree does not cause any delay constraint violations. The chosen link e is then added to the tree, and link l is removed. Phase 2 repeats this operation until no more cost reduction can be achieved.

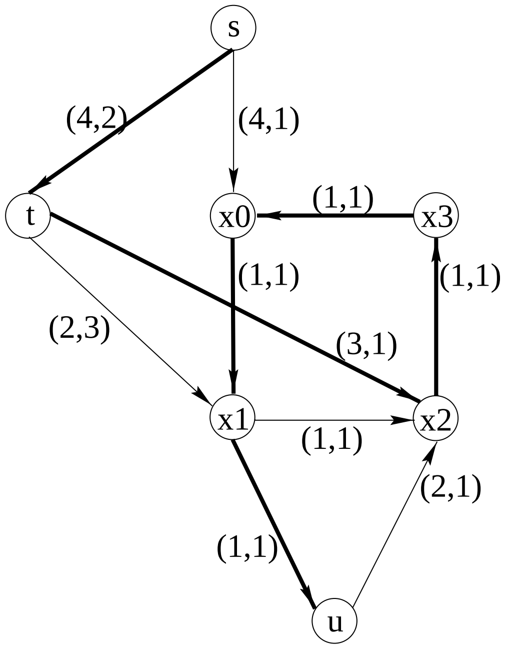
The link replacement operation of phase 2 may result in loops. Before a loop is created, the loop breaking algorithm (LBA) is applied. If LBA succeeds, the loop is created and broken according to its output, else the loop is not created in the first place. The pseudo code for LBA is given below.

```
1. Given $N = (V, E)$, a tree $T(s) \subset E$, a potential loop path consisting of the set
   of nodes $X = \{x_0, \ldots, x_k\} \subset V$, and the set of links $L = \{l_1 = (x_0, x_1), \ldots, l_k =$
   $(x_{k-1}, x_k)\} \subset T(s)$, a link $l'_0 = (x_k, x_0)$ to close the loop if added, and a link $l_0 =$
   $(y, x_0)$ linking the potential loop path upstream towards the source. $C(l'_0) <$
   $C(l_0)$.
2. $improvement := 0$, $l_{remove}$ and $l_{add}$ are not set.
3. Repeat for all nodes $x_i \in (X - x_0)$ {
4.    Find the least cost link $l'_i$, to replace $l_i$ in connecting $x_i$ upstream towards
      the source, such that:
      • $C(l_0) + C(l_i) - C(l'_0) - C(l'_i) > improvement$
      • The resulting tree after removing $l_0$ and $l_i$ and replacing them with
        $l'_0$ and $l'_i$ does not contain any loops and does not violate the delay
        constraint.
      If such a link exists: $improvement := C(l_0) + C(l_i) - C(l'_0) - C(l'_i)$,
      $l_{remove} := l_i$, $l_{add} := l'_i$.
5. }
6. If $l_{remove}$ and $l_{add}$ are not set: return failure, else return success and the values
   of $l_{remove}$ and $l_{add}$.
```

LBA attempts to select the proper location to break a given loop and to restore the tree structure without violating the delay constraint, and such that the cost of the resulting tree after creating and breaking the loop is lower. Using the same terminology of LBA's pseudo code, if link $l'_0 = (x_k, x_0)$ is added, a loop results.



(a) Cost(T(s)) = 12, Max_Delay(T(s)) = 4.



(b) Cost(T(s)) = 11, Max_Delay(T(s)) = 7.

Figure 4.2: Example of the operation of the loop breaking algorithm. The delay constraint $\Delta = 7$. Link costs and delays are shown in brackets, $(cost, delay)$.

Removing link $l_0 = (y, x_0)$ disconnects the loop nodes, x_i, $i = 1, \ldots, k$, from the upstream tree. LBA scans each loop node and attempts to remove the link $l_i = (x_{i-1}, x_i)$ and to replace it with another link $l'_i = (z, x_i)$, $z \in V$. Removing l_i breaks the loop. The link l'_i is selected such that it reattaches the nodes of the broken loop upstream towards the source s. LBA attempts to select l_i and l'_i such that the resulting tree does not violate the delay constraint, and such that its cost is lower than the cost of the initial tree given to LBA. If there are multiple alternatives to break the loop, LBA chooses the alternative which results in the lowest cost tree. If LBA can not find any links l_i and l'_i to break the loop, it fails, else it outputs the values of l_i and l'_i.

Example: Figure 4.2 shows an example illustrating the loop breaking operation. Figure 4.2(a) shows the broadcast tree constructed by the first phase of BDB. Its cost is 12. In phase 2, BDB attempts to remove the link $l_0 = (s, x_0)$ and to replace it with the link $l'_0 = (x_3, x_0)$ in order to reduce the tree cost. However, this results in a loop $\{l_1 = (x_0, x_1), l_2 = (x_1, x_2), l_3 = (x_2, x_3), l'_0 = (x_3, x_0)\}$. LBA is called to determine how to break that loop. For the given case, LBA has only three alternatives. The first alternative is to remove the link l_1 and replace it with the link (t, x_1). This breaks the loop and results in a tree cost of 11, but the resulting tree would have a maximum end-to-end delay of $8 > \Delta$. Thus it is rejected. The second alternative is to remove the link l_2 and replace it with the link (u, x_2), thus breaking the current loop, but creating another one consisting of $\{(x_0, x_1), (x_1, u), (u, x_2), x_2, x_3), (x_3, x_0)\}$. This is also unacceptable. The only remaining alternative for the given example is to remove l_2 and replace it with (t, x_2) which results in an acceptable tree with a total cost of 11 and a maximum end-to-end delay of $7 = \Delta$ as shown in figure 4.2(b).

The DMCT algorithm [40] resembles phase 1 of BDB, but its approach to relax the delays is different from BDB's. However, DMCT does not have an equivalent to phase 2 of BDB. DMCT's execution stops as soon as all destination nodes are included in the tree being constructed, because its aim is to construct a Steiner tree and not a spanning tree.

In the next section we compare the performance of BDB to the performance of COPT, BSMA, and DMCT.

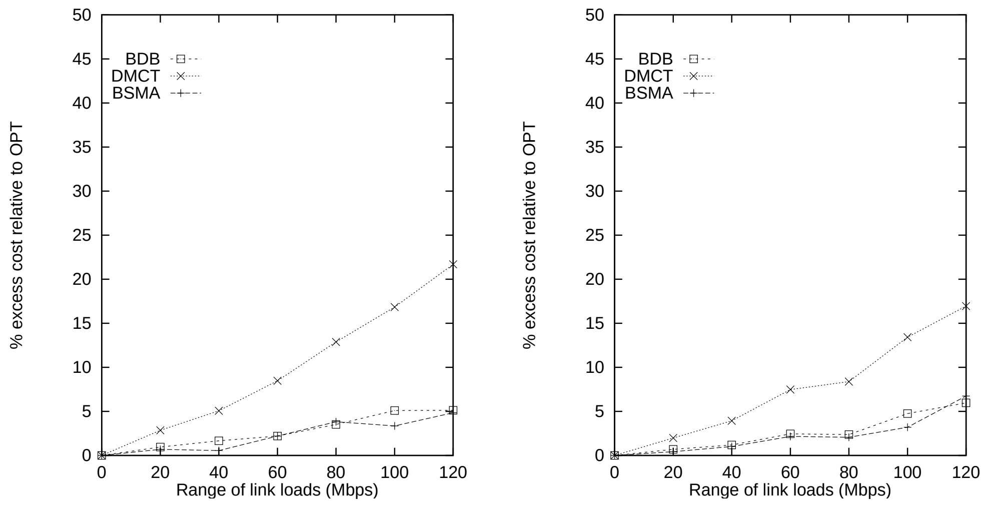
4.3 Simulation Results

For our experimental investigation, we used the same network topologies, cost and delay functions, and traffic sources which we used in the experiments of chapter 3. Therefore, a brief summary of the experimental setup suffices. The random generator described in section 3.1.1 was used to create random networks. Networks ranging in size from 20 nodes up to 200 nodes with an average node degree of 4 were simulated. The networks spanned an area of $4000 * 2400$ Km^2, and 155 Mbps links were used. Under these assumptions, the propagation components dominated the link delays. A link's cost was set equal to the sum of the equivalent bandwidths of the traffic streams traversing that link.

The experiment we ran, compares the different algorithms when each of them is applied to create a broadcast tree for a given source node generating VBR traffic with an equivalent bandwidth of 0.5 Mbps, under given network loading conditions. For each run of the experiment, we generated a random set of links to interconnect the fixed nodes, we selected a random source node, and we generated random background traffic for each link. The equivalent bandwidth of each link's background traffic was a random variable uniformly distributed between B_{min} and B_{max}. This represented the initial cost of each link. We simulated both symmetric and asymmetric networks. For networks with symmetric link loads, the initial cost of a link $e = (u, v)$ and the initial cost of the reverse link $e' = (v, u)$ were set to the same random value. For networks with asymmetric link loads, however, the initial cost of $e = (u, v)$ and $e' = (v, u)$ were independent random variables. In that case, the asymmetry of the link loads increased as the range of the link loads, i.e., the difference between B_{max} and B_{min}, increased.

The experiment was repeated with different link loading conditions, different delay constraints, and different network sizes. For each setting of these parameters, we measured the total cost of the broadcast tree. We ran the algorithms repeatedly until confidence intervals of less than 5%, using 95% confidence level, were achieved. On the average, 300 different networks were simulated in each experiment, in order to reach such confidence levels. At least 250 networks were simulated in each case. We simulated the following algorithms: BDB, DMCT, BSMA, and COPT. We could not apply COPT to networks with more than 20 nodes due to its excessive execution time. The other algorithms were applied to networks with up to 200 nodes. Therefore, we show the percentage excess cost of each algorithm relative to COPT in case of 20-node networks only. When discussing results for networks with more than 20-nodes, we show the percentage excess cost of each algorithm relative to BSMA which was previously shown in chapter 3 to be the best performing delay-constrained minimum Steiner tree heuristic.

Figure 4.3 shows the cost of a broadcast tree versus the range of link loads, $B_{max} - B_{min}$, for 20-node networks and a tight delay constraint of 0.03 seconds. We increased the range of link loads, $B_{max} - B_{min}$, from 0 to 120 Mbps while keeping the average link load, $(B_{max} + B_{min})/2$, fixed at 65 Mbps at all times. All algorithms perform equally well in case of zero range of link loads, because all link costs are equal and

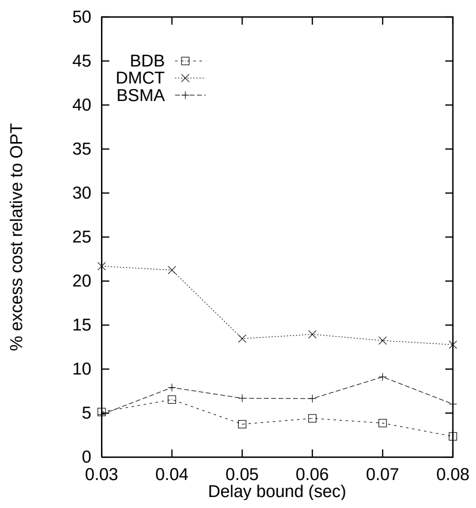


(a) Asymmetric load (b) Symmetric load

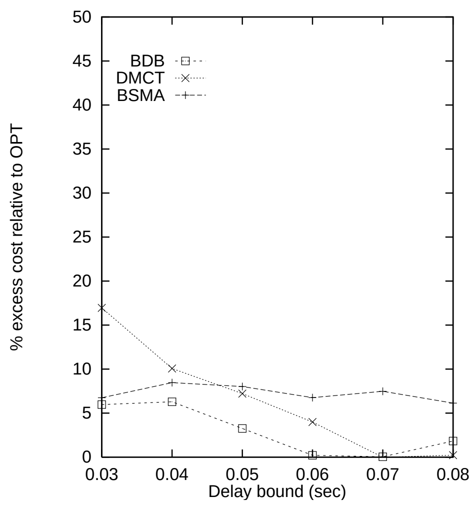
Figure 4.3: Percentage increase in the cost of a broadcast tree relative to optimal, variable range of link loads, 20 nodes, delay constraint $\Delta = 0.03$ seconds.

there is nothing to be minimized in that case. DMCT's costs deviate further from optimal as the range of link loads increases. It is up to 23% worse than COPT when the link loads are asymmetric, but it performs slightly better in case of symmetric link loads and yields tree costs that are within 15% from COPT. BDB and BSMA remain close to optimal even when the range of link loads is large. The tree costs of BDB and BSMA are equal throughout the entire range of link loads simulated in this experiment. BDB's tree costs are within 7.5% from COPT when asymmetric link loads are used and only 5% off optimal in case of using symmetric link loads. We found that DMCT's tree costs are comparable to the costs of the intermediate trees resulting from phase 1 of BDB. Thus the phase 2 of BDB is capable of reducing the cost of its initial tree by up to 15% in some cases.

We repeated the experiment with a fixed range of link loads and a variable delay constraint. The results are shown in figure 4.4 for 20-node networks. When the delay constraint is tight, it limits the algorithms' ability to construct low cost trees, because there aren't many possible solutions for the problem. As the delay constraint increases, its effect on restricting the algorithms' efficiency in constructing low cost

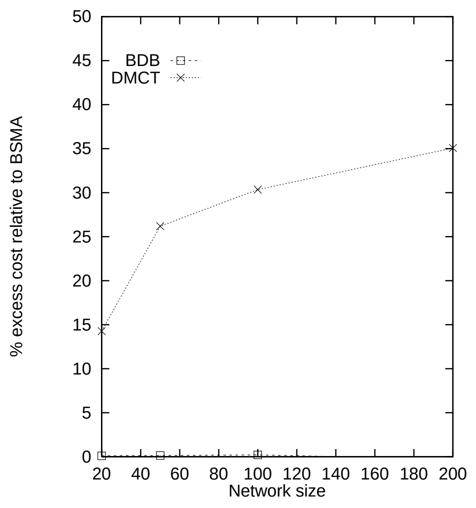


(a) Asymmetric load

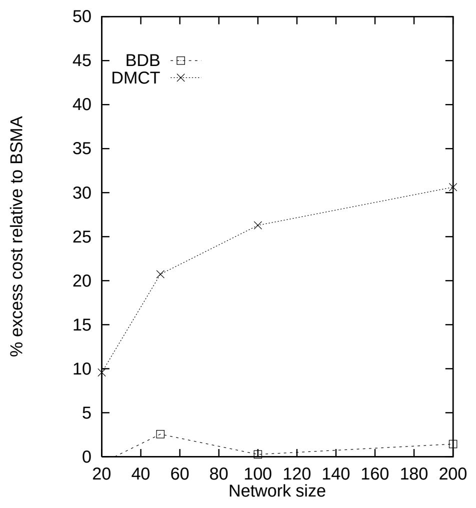


(b) Symmetric load

Figure 4.4: Percentage increase in the cost of a broadcast tree relative to optimal, variable delay constraint Δ, 20 nodes, $B_{min} = 5$ Mbps, $B_{max} = 125$ Mbps.



(a) Asymmetric load



(b) Symmetric load

Figure 4.5: Percentage increase in the cost of a broadcast tree relative to BSMA's tree cost, variable network size, delay constraint $\Delta = 0.03$ seconds, $B_{min} = 5$ Mbps, $B_{max} = 125$ Mbps.

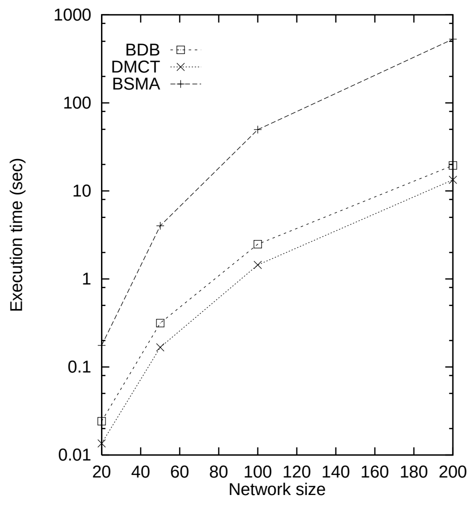
trees diminishes, and the algorithms are capable of constructing lower cost trees. When the delay constraint increases further, the effect on the resulting tree costs is minimal, because the solution of the delay-constrained minimum spanning tree problem approaches the solution of the unconstrained minimum spanning tree problem. This is evident in figure 4.4(b), where BDB and DMCT are almost optimal when the delay constraint exceeds 0.06 seconds. This happens because DMCT and phase 1 of BDB are based on Prim's algorithm, which is optimal for the unconstrained minimum spanning tree problem in undirected networks. BDB performs slightly better than BSMA when the delay constraint is relaxed, both when the link loads are asymmetric and when they are symmetric.

Figure 4.5 shows the percentage excess costs of BDB and DMCT relative to BSMA when the network size varies from 20 nodes to 200 nodes while keeping the delay constraint and the range of link loads fixed. BDB performs as well as BSMA does throughout the entire range of network sizes, while DMCT's performance relative to BSMA deteriorates as the network size increases. For 200-node networks, DMCT's trees are 30% and 35% more expensive than BSMA for symmetric and asymmetric link loads respectively.

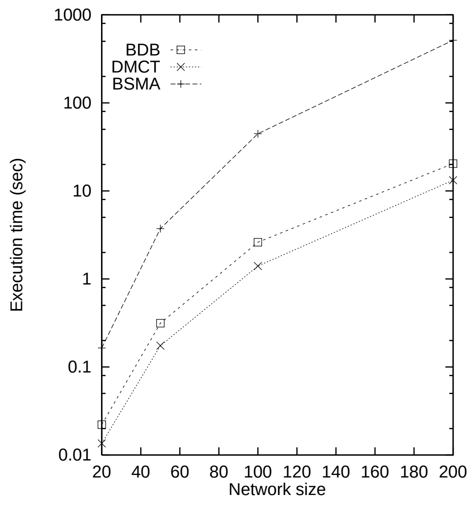
Finally, in figure 4.6, we present the average execution times of BDB, DMCT, and BSMA versus the network size. Note, however, that our code for these algorithms was not optimized for speed. Figure 4.6 shows that the average execution times of both heuristics grow at the same rate and are always within the same order of magnitude, with BDB being constantly slower than DMCT by approximately 60%. BSMA is at least one order of magnitude slower than BDB and DMCT, and its execution time grows at a faster rate.

4.4 Conclusions

We studied the problem of constructing broadcast trees for real-time traffic with delay constraints in networks with asymmetric link loads. We formulated the problem as a delay-constrained minimum spanning tree problem in directed networks, and then we proved that this problem is *NP*-complete. We proposed a bounded delay broadcast heuristic (BDB) to solve the delay-constrained minimum spanning tree



(a) Asymmetric load



(b) Symmetric load

Figure 4.6: Average Execution time, variable network size, delay constraint $\Delta = 0.03$ seconds, $B_{min} = 5$ Mbps, $B_{max} = 125$ Mbps.

problem. The heuristic consists of two phases. The first phase is based on Prim's algorithm [8] and constructs a moderate-cost delay-constrained spanning tree. The second phase reduces the cost of that tree by replacing tree link with lower cost links not in the tree without violating the imposed delay constraint. Simulation results show that BDB's performance is close to optimal in case of networks with asymmetric link loads as well as in the special case of networks with symmetric link loads. Delay-constrained minimum Steiner tree heuristics can be used to solve the delay-constrained minimum spanning tree problem. We compared BDB to the best delay-constrained Steiner heuristic with respect to tree cost, BSMA, and to a fast delay-constrained Steiner heuristic, DMCT. BDB is as efficient as BSMA in constructing low cost broadcast trees. DMCT is not as efficient as BDB. As the network size increases the difference between BDB and DMCT increases. DMCT's trees are up to 35% more expensive than BDB's trees when the network size is 200 nodes. DMCT was designed for networks with symmetric link loads, but even in that case BDB performs better. The average execution times of both BDB and DMCT grow at the same rate with DMCT being faster than BDB. BSMA's execution times are larger than BDB's execution times and grow at a faster rate. Unfortunately, however, we could not perform a conclusive analysis of the worst case time complexity of BDB. In summary, our experimental results indicate that BDB is a fast and efficient delay-constrained minimum spanning tree heuristic.

Chapter 5

Delay-Constrained Unicast Routing Problem

Routing problems can be divided into unicast problems, multicast problems, and broadcast problems. In this chapter, we only consider the unicast routing problem. Unicast routing protocols can be classified into two categories: distance-vector protocols, e.g., the routing information protocol (RIP) [65, 66], and link-state protocols, e.g., the open shortest path first protocol (OSPF) [71]. Distance-vector routing protocols are based on a distributed version of Bellman-Ford shortest path algorithm [11]. Considering the message complexity, distance-vector routing protocols scale well to large network sizes, because each node sends periodical topology update messages only to its direct neighbors. No flooding or broadcasting operations are involved. Each node maintains only limited information about the shortest paths to all other nodes in the network. Due to their distributed nature, distance-vector protocols may suffer from looping problems when the network is not in steady state. In link-state routing protocols, on the other hand, each node maintains complete information about the network topology and uses this information to compute the shortest path to a given destination centrally using Dijkstra's algorithm [12]. Link-state protocols have limited scalability, because flooding is used to update the nodes' topology information. They do not suffer from looping problems, however, because of their centralized nature. Recently, Garcia-Luna-Aceves and Behrens [89] proposed a distributed protocol, based on link vectors, that avoids looping problems and scales well to large networks.

Bellman-Ford and Dijkstra algorithms are the best known shortest path algorithms. As has been discussed in section 2.4.1, the properties of the shortest path depend on the metric the link length represents. If the length of a link is a measure of the delay on that link, then a shortest path algorithm computes the least-delay path, **LD**, and if the link length is set equal to the link cost, then a shortest path algorithm computes the least-cost path, **LC**. Many variations of the shortest path problem have been studied over the years. For example, Simha and Narahari [90] studied the case where queueing delay is the dominant component of a link's delay, and Aida et. al [91] proposed an optimal shortest path algorithm that takes into account both the mean and the variance of the link delays. Rampal and Reeves [83] investigated the interaction between routing and call admission control for multimedia traffic. Plotkin [92] proposed a strategy that unifies the routing and call admission decisions. Chen and Liu [93] showed that the problem of routing a multimedia connection subject to a cell loss constraint is *NP*-complete and proposed heuristic solutions for that problem. Fratta, Gerla, and Kleinrock [94] studied the routing and link capacity assignment problem when given a set of source-destination pairs, and a constraint on the average delay per packet from source to destination. Their method is known as the flow deviation method. It permits packets belonging to the same stream to be forwarded over different paths from source to destination. Thus packets may need to be reordered upon their arrival at the destination. In addition, forwarding packets of the same real-time traffic stream over different paths may cause complicated synchronization problems at the destination. The flow deviation method is, therefore, not suitable for real-time applications.

We study the problem of unicast routing of real-time traffic subject to an end-to-end delay constraint in connection-oriented networks. We formulate the problem as a Delay-Constrained LC (DCLC) path problem. This problem has been shown to be *NP*-complete [33]. Therefore, we propose a distributed heuristic solution: the Delay-Constrained Unicast Routing (DCUR) algorithm. Widyono [34] proposed an optimal centralized delay-constrained algorithm to solve the DCLC problem. His algorithm, called the Constrained Bellman-Ford (CBF) algorithm, performs a breadth-first search to find the DCLC path. Unfortunately, due to its optimality, CBF's worst case running times grow exponentially with the size of the network. Jaffe [95] studied a variation of the problem in which the path cost and the path delay are defined as two constraints, and he proposed a pseudo-polynomial-time heuristic and a polynomial-time heuristic for solving the problem. The path cost (and similarly the path delay) is an additive metric, i.e, it is equal to the sum of the costs of the links on the path. Wang and Crowcroft [96] investigated the routing problem subject to multiple quality of service constraints in datagram networks. They considered multiplicative and concave constraints in addition to additive constraints.

In this chapter, we propose a distributed heuristic to solve the DCLC problem: the delay-constrained unicast routing algorithm, **DCUR**. DCUR requires a limited amount of computation at any node, and its message complexity is $O(|V|^3)$ in the worst case. On the average, however, DCUR requires much fewer messages, and therefore, it scales well to large network sizes. DCUR requires only a limited amount of information at each node. This information is stored in a delay vector and a cost vector that are similar to the distance vectors of some existing routing protocols [65]. The basic idea of DCUR is that when a node receives a request to construct a delay-constrained path to a given destination, that node is given the choice between two alternatives only. The node can either follow the direction of the LC path or the direction of the LD path. After deciding which direction to follow, the node sends a request to the next hop node in that direction to take over responsibility for the rest of the path construction operation. When the next hop node receives a request to construct a delay-constrained path, it follows the same procedure that has been explained for the previous node. Each node hands over the responsibility for path construction to the next hop node in the direction of the destination until the destination itself is reached. Limiting the number of paths to choose from at any node to only two restricts the amount of computation DCUR requires considerably.

Establishing a connection that provides guaranteed service involves routing, signaling, call admission control, and resource reservation. In our work, we only consider the routing aspect of the problem and leave the other aspects for future investigation. The remainder of this chapter is organized as follows. In section 5.1, we formulate the DCLC problem. In section 5.2, we describe the routing information needed at each node for successful execution of DCUR. Then, in section 5.3, we present the distributed DCUR algorithm. In section 5.4, we prove its correctness, and then we study its complexity in section 5.5. In section 5.6, we evaluate DCUR's performance using simulation. Finally, we present our conclusions in section 5.7.

5.1 Problem Formulation

In this chapter, we use the same definitions presented in section 2.3. However, there is no need to define a multicast group G in this chapter, because we consider only one destination node d. For a given source node $s \in V$ and destination node $d \in V$, we define $\mathcal{P}(s,d) = \{P_1, \ldots, P_m\}$ as the set of all possible alternative paths from s to d.

The DCLC problem finds the least-cost path from a source node s to a destination node d such that the delay along that path does not exceed a delay constraint Δ. It is a constrained minimization problem that can be formulated as follows.

Delay-Constrained Least-Cost (DCLC) Path Problem: *Given a directed network $N = (V, E)$, a nonnegative cost $C(e)$ for each $e \in E$, a nonnegative delay $D(e)$ for each $e \in E$, a source node $s \in V$, a destination node $d \in V$, and a positive delay constraint Δ, the constrained minimization problem is:*

$$\min_{P_i \in \mathcal{P}'(s,d)} \quad Cost(P_i) \tag{5.1}$$

where $\mathcal{P}'(s,d)$ is the set of paths from s to d for which the end-to-end delay is bounded by Δ. Therefore $\mathcal{P}'(s,d) \subseteq \mathcal{P}(s,d)$. If $P_i \in \mathcal{P}(s,d)$ then $P_i \in \mathcal{P}'(s,d)$ if and only if

$$Delay(P_i) \leq \Delta. \tag{5.2}$$

The decision version of the DCLC problem is *NP*-complete [33]. It remains *NP*-complete in the case of undirected networks. However, it is solvable in polynomial time if all link costs are equal or all link delays are equal. To avoid the excessive complexity of the optimal solutions for the DCLC problem, we propose a simple distributed heuristic, DCUR.

In the next section, we discuss the routing information which needs to be present at any node in the network to assure successful execution of DCUR. Then, in section 5.3, we describe the operation of DCUR.

5.2 Routing Information

Every node $v \in V$ must have the following information available during the computation of the delay-constrained path: the costs of all outgoing links, the delays of all outgoing links, a cost vector, a delay vector, and a routing table.

The cost vector at node v consists of $|V|$ entries, one entry for each node w in the network. Each entry in the cost vector holds the following information:

- the destination node address, w,
- the cost of the LC path from v to w, $least\_cost\_value(v, w)$, and
- the address of the next hop node on the LC path from v to w, $least\_cost\_nhop(v, w)$.

Similarly, the delay vector at node v has one entry for each node w in the network. Each entry in the delay vector holds:

- the destination node address, w,
- the total end-to-end delay of the LD path from v to w, $least\_delay\_value(v, w)$, and
- the address of the next hop node on the LD path from v to w, $least\_delay\_nhop(v, w)$.

The cost vectors and delay vectors are similar to the distance vectors of some existing routing protocols [65]. Distance-vector based protocols discuss in detail how to update the distance vectors in response to topology changes, and how to prevent instability. These procedures are simple and require the contents of the distance vector at each node to be periodically transmitted to direct neighbors of that node only. They do not involve any flooding or broadcasting operations. The same procedures used for maintaining the distance vectors can be used for maintaining the cost vectors and delay vectors. We will not discuss these procedures, because our focus is on a routing algorithm that uses the cost vectors and delay vectors as input information. Thus, we assume that the cost vectors and delay vectors at all nodes are up-to-date. We also assume that the link costs, the link delays, the contents of the cost vectors, and the contents of the delay vectors do not change during the execution of the routing algorithm.

In addition to the cost vector and delay vector, each node v maintains a routing table. Each entry in the routing table corresponds to an established path from a source node s to a destination d that passes through v. A routing table entry will be described in the next section. Routing table entries are created during the connection establishment phase for a session involving real-time traffic that flows from a source s to a destination d. In addition to the routing information, a routing table entry could also be used to hold information about the resources reserved for the connection from s to d. When a real-time session terminates, the corresponding path is torn down, and all routing table entries corresponding to that path are deleted.

5.3 The Delay-Constrained Unicast Routing Algorithm (DCUR)

We start by presenting a simple version of DCUR, and explaining how to implement it in a distributed fashion. Then we discuss how loops may be created, and how DCUR detects them and eliminates them. After completing the description of DCUR, we prove its correctness and study its complexity in sections 5.4 and 5.5 respectively.

DCUR is a source-initiated algorithm that constructs a delay-constrained path connecting source node s to destination node d. The path is constructed one node at a time, from the source to the destination. Any node v at the end of the partially-constructed path can choose to add one of only two alternative outgoing links. One link is on the LC path from v to the destination, while the other link is on the LD path from v to the destination. This limitation restricts DCUR's ability to construct the optimal path, but it considerably reduces the amount of computation required at any node.

In the following, we describe a simple version of DCUR which assumes that no routing loops can occur. The source node s initiates path construction by looking up the $least\_delay\_value(s, d)$ from its delay vector. If this value is greater than the delay constraint Δ, then no delay-constrained paths exist between s and d, and DCUR fails and stops. If, however, delay-constrained paths do exist, i.e.,

$$least\_delay\_value(s,d) \leq \Delta, \tag{5.3}$$

the algorithm proceeds. The source s becomes the active node, denoted *active_node*. At all times there is only one active node, at the end of the partially-constructed path. The variable *delay_so_far* is set to 0, and the variable *previous_active_node* is set to *null*.

The *active_node* reads the address of the next hop node on the LC path towards d, $least\_cost\_nhop(active\_node, d)$, from its cost vector. For convenience, we denote $least\_cost\_nhop(active\_node, d)$ as *lc_nhop*. Then *active_node* sends a *Query* message to *lc_nhop*, inquiring about the LD value from *lc_nhop* to d. *lc_nhop* looks up the requested value $least\_delay\_value(lc\_nhop, d)$ from its delay vector, and sends a *Response* message back to *active_node* with this information. After *active_node* receives the *Response* message, it checks if

$$delay\_so\_far + D(active\_node, lc\_nhop) + least\_delay\_value(lc\_nhop, d) \leq \Delta. \tag{5.4}$$

If the inequality is satisfied, then there exist delay-constrained paths from *active_node* to d which use the link $(active\_node, lc\_nhop)$, and *active_node* selects the direction of the LC path towards d. If the inequality is not satisfied, then *active_node* selects the direction of the LD path towards d. The LD path from *active_node* to d is guaranteed to be part of at least one delay-constrained path from s to d; otherwise, *active_node* could not have been selected in a previous step (a proof is provided in section 5.4). After deciding which direction to follow, *active_node* creates a routing table entry with the following information:

- $source$ = the address of s,
- $destination$ = the address of d,
- $previous\_node$ = address of the $previous\_active\_node$,
- $next\_node = \begin{cases} lc\_nhop & \text{if the LC path direction is chosen,} \\ least\_delay\_nhop(active\_node, d) & \\ & \text{if the LD path direction is chosen,} \end{cases}$
- $previous\_delay = delay\_so\_far$, and

- $flag = \begin{cases} LCPATH & \text{if the LC path direction is chosen,} \\ LDPATH & \text{if the LD path direction is chosen.} \end{cases}$

Then $active\_node$ adds $D(active\_node, next\_node)$ to the variable $delay\_so\_far$. Finally the $active\_node$ sends a *Construct_Path* message to $next\_node$ that contains: the address of the source s, the address of the destination d, the value of the delay constraint Δ, and the updated value of $delay\_so\_far$ which represents the delay along the already constructed path from s to $next\_node$. After sending out the *Construct_Path* message, $active\_node$ becomes inactive.

When a node $v \neq d$ receives a *Construct_Path* message, it becomes the new $active\_node$. The new $active\_node$ sets $previous\_active\_node$ to be the address of the node which sent it a *Construct_Path* message. Then the new $active\_node$ executes the same procedure just described.

When the destination node d receives a *Construct_Path* message, it records the address of the node which sent the message. d creates a routing table entry, with the following values: address of the source s, address of the destination d, $previous\_node = previous\_active\_node$, $next\_node = null$, and $previous\_delay = delay\_so\_far$. Then the destination sends an *Acknowledgement* back to the source. The *Acknowledgement* message travels the constructed path backwards hop by hop until it reaches the source. When the source receives the *Acknowledgment* message, it signals to the application that the path construction has been successfully completed, and traffic can be transmitted along that path.

An $active\_node$ does not send a *Query* message if the next hop node is the same on both the LC path and the LD path from $active\_node$ to the destination, i.e., $least\_cost\_nhop(active\_node, d) = least\_delay\_nhop(active\_node, d)$. It is known in advance that the LD direction satisfies the delay constraint, so there is no need for the *Query* message. In this case, $active\_node$ sets the $flag$ in the routing table entry to $LDPATH$. The reason for that particular setting will be explained later in this section, when routing loops are discussed.

The need for the *Query* and *Response* messages can be completely eliminated by making use of the fact that a node transmits the contents of its cost vector and delay vector periodically to all its neighbors. Thus if a node saves a copy of the delay vector from each of its neighbor nodes, then there is no need for the *Query* and *Response* messages. However, this increases the storage requirements at each node.

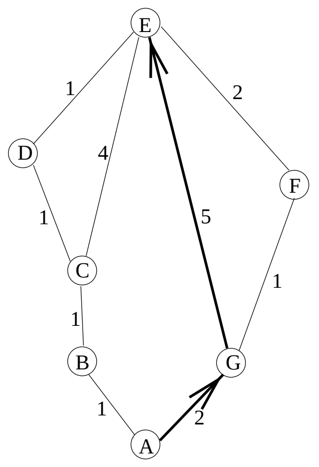
Figure 5.1 shows an example[1] of the paths obtained by different routing algorithms to connect source node A to destination node E, with a delay constraint of 3. Subfigure 5.1(d) shows the path DCUR constructs. DCUR proceeds as follows. The source A adds the first link on the LC path towards E, link (A, B), after checking that there exist delay-constrained paths from A to E that utilize (A, B). Then node B adds the first link on its LC path towards E, link (B, C), after checking that there exist delay-constrained paths from A to E that utilize (A, B) and (B, C). Node C next determines that the first link on its LC path towards E, link (C, D), cannot be used. This is because the subpath $\{A \rightarrow B \rightarrow C \rightarrow D\}$ is not part of any delay constrained path from A to E. Thus C decides to continue via the LD path direction. It adds the first link in that direction, link (C, E), which leads directly to the destination.

The paths constructed by existing distance-vector protocols are guaranteed to be loop-free if the contents of the distance vectors at all nodes are up-to-date and the network is in stable condition. However, up-to-date cost vectors and delay vectors contents and stable network condition are not sufficient to guarantee loop-free operation for DCUR. In DCUR, each node involved in the path construction operation selects either the LC path direction or the LD path direction, as has been explained before. If all nodes choose the LC path direction, or all nodes choose the LD path direction, then no loops can occur, because the resulting paths are the LC path or LD path respectively. However, if some nodes choose the LC path direction while others choose the LD path direction, loops may occur. In the following subsection, we discuss how DCUR detects and eliminates loops. We will prove in section 5.4 that, although loops may occur during the execution of DCUR, the final path DCUR constructs is guaranteed to be loop-free.

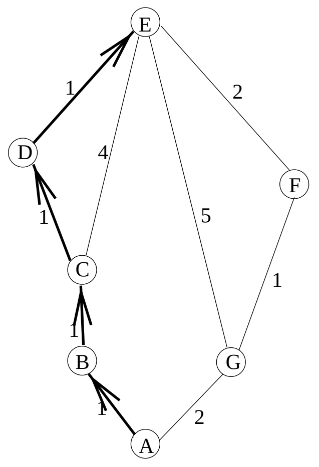
5.3.1 Loop Removal

Figure 5.2 shows a scenario that results in a loop. The source node A initiates the construction of a path towards the destination node D with an imposed delay constraint value of 8. Subfigures 5.2(a), 5.2(b), and 5.2(c) show successive stages

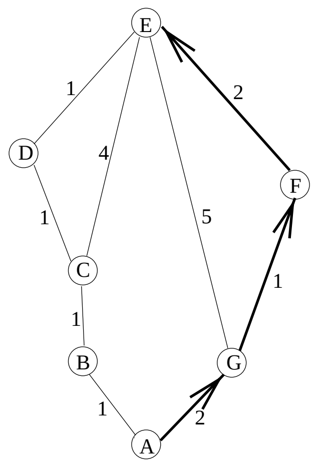
[1]Figures 5.1 and 5.2 show examples of undirected networks for simplicity. DCUR can be applied to both directed and undirected networks.



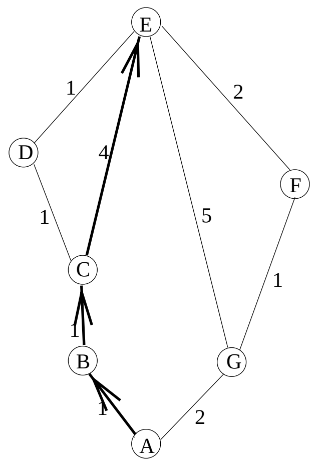
(a) LD path, cost = 7, delay = 2



(b) Unconstrained LC path, cost = 4, delay = 4



(c) Optimal DCLC path, cost = 5, delay = 3



(d) DCUR path, cost = 6, delay = 3

Figure 5.1: Paths constructed by different algorithms from source node A to destination node E. All link delays are equal to 1, and are not shown in the figure. Link costs are shown next to each link. The delay constraint, Δ, is equal to 3.

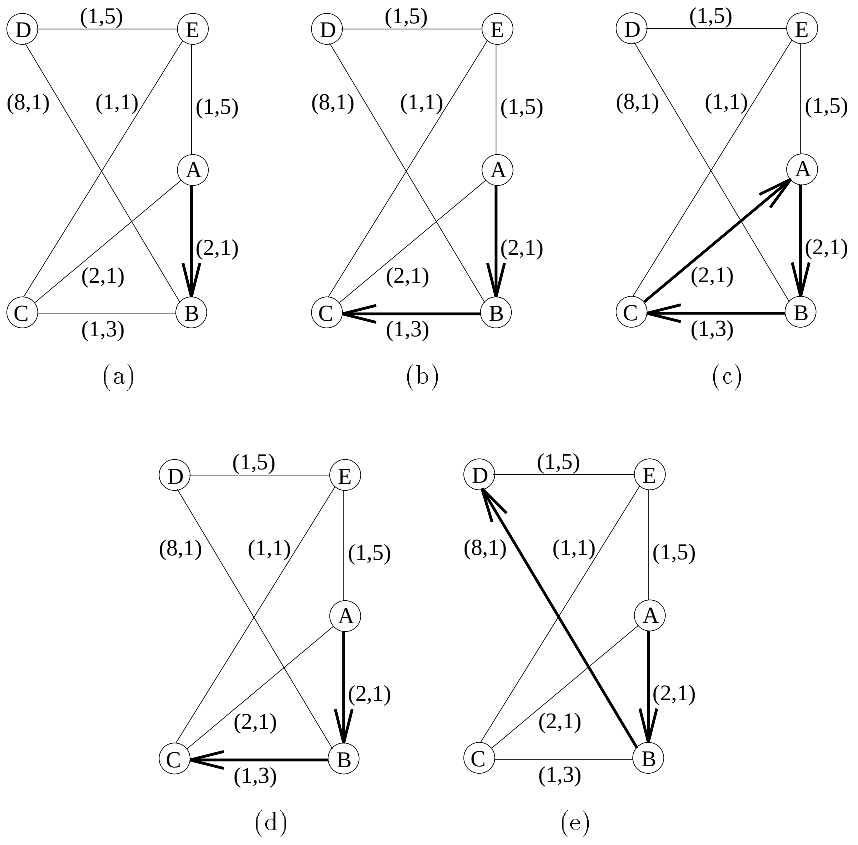


Figure 5.2: Example of a loop scenario. A is the source and D the destination. Link costs and link delays are shown next to each link as (cost,delay). The delay constraint, Δ, is equal to 8.

of path construction until a loop is created. The source A follows the LD path direction towards the destination D and starts the path construction by adding link (A, B). Node B follows the LC path direction towards D and adds link (B, C) to the path. Node C also attempts to follow the LC path direction towards D by using link (C, E). This is not possible however, because C's calculations reveal that if (C, E) is added, DCUR will not be able to construct a delay-constrained path to D (the least total delay that can be achieved in this case is 10). Thus C follows the LD path direction and adds link (C, A) to the path being constructed. This creates the loop $\{A \rightarrow B \rightarrow C \rightarrow A\}$, as shown in subfigure 5.2(c).

DCUR detects loops as follows. When a node receives a $Construct\_Path$ message, it searches its routing table. A loop is detected if a routing table entry already exists for the source-destination pair specified in the $Construct\_Path$ message.

The active node, *active_node*, that detects a loop initiates the loop removal operation. The contents of *active_node*'s routing table entry are left unchanged. *active_node* sends a *Remove_Loop* message to the previous node on the loop, *previous_active_node* (the node from which *active_node* received the last $Construct\_Path$ message), and then *active_node* becomes inactive. The addresses of the source and destination nodes are all that needs to be included in the *Remove_Loop* message. The *Remove_Loop* message traverses the loop backwards, removing routing table entries, until it finds a node w whose routing table entry's $flag$ is set to $LCPATH$ indicating that this node is following the LC path direction towards the destination. There must be at least one node on the loop that follows the LC path direction, because, as we mentioned before, loops can not be created if all nodes follow the LD path direction. The *Remove_Loop* message is not sent any further backwards along the loop, after it arrives at w. Node w then decides to follow the LD path direction, instead of the LC path direction, in order to avoid the conditions that caused the loop. This decision can never lead to any delay constraint violations. Thus w adjusts the contents of its routing table entry so that $next\_node = least\_delay\_nhop(w, d)$ and $flag = LDPATH$. The variables *previous_node* and *previous_delay* remain unchanged. Then w sends a $Construct\_Path$ message to *next_node*, and path construction continues.

For the example of figure 5.2, node A detects the existence of a loop. A reacts by sending a *Remove_Loop* message that traverses the loop backwards. Node C receives the *Remove_Loop* message from A, but C is already following the LD path direction towards the destination, so all it does is to send the *Remove_Loop* message further backwards to B, and to delete its routing table entry, thereby removing link (C, A) from the path (subfigure 5.2(d)). Node B receives the *Remove_Loop* message. It is following the LC path direction towards the destination, so it decides to follow the LD path direction instead, and modifies its routing table entry accordingly, thus removing link (B, C) from the path and adding link (B, D) instead. Then B continues constructing the path by sending a $Construct\_Path$ message to D, which is the destination. The final delay-constrained path from A to D is the one shown in subfigure 5.2(e).

It was mentioned above that, at a node w, the routing table entry's $flag$ is set to $LDPATH$ when both the LC path direction and the LD path direction share the same link to the next hop. The reason is that if the $flag$ were set to $LCPATH$, and then w received a *Remove_Loop* message, it would have removed the link leading to the next node in the LC path direction, and then it would have added the same link to the path again, because that link leads also to the next node in the LD path direction. The result would have been the same loop occurring twice.

The description of DCUR is now complete. Pseudo code for the algorithm is given in appendix B. In the next two sections, we prove the correctness of DCUR and study its complexity.

5.4 Correctness of DCUR

We verify the correctness of DCUR by proving that it can always construct a loop-free delay-constrained path within finite time, if such a path exists. If no feasible path exists for a given source-destination pair, DCUR fails immediately at the source node after it determines that inequality 5.3 is not satisfied. Thus there is no unnecessary overhead if no solution exists.

Theorem 5.1 *DCUR always constructs a delay-constrained path for a given source s and destination d, if such a path exists.*

Proof. If no feasible path exists for a given source-destination pair, DCUR fails immediately at the source node after checking that the delay along the LD path exceeds the delay constraint, i.e., inequality 5.3 is not satisfied. If the LD path can not satisfy the delay constraint, no other path can. If at least one delay-constrained path from s to d exists, then inequality 5.3 will be satisfied, and path construction can start. Initially, the source s is the only member in the path. The rest of this proof is done by induction on j, where $P_j = \{v_0 \rightarrow \ldots \rightarrow v_j\}$ denotes the subpath constructed starting at the source, $s = v_0$, and ending at the current active node, $active\_node = v_j$, and j denotes the length of the path in hops. The basis for induction is $P_0 = \{v_0\}$.

Since inequality 5.3 is satisfied, and $Delay(P_0) = 0$, it follows that

$$Delay(P_0) + least\_delay\_value(v_0, d) \leq \Delta. \tag{5.5}$$

Assume that

$$Delay(P_j) + least\_delay\_value(v_j, d) \leq \Delta. \tag{5.6}$$

Inequality 5.6 guarantees that the subpath P_j is part of at least one delay-constrained path from s to d. DCUR proceeds by adding either the first link along the LC path from v_j to d or the first link along the LD path from v_j to d. If DCUR adds the first link along the LC path, i.e., $v_{j+1} = lc\_nhop = least\_cost\_nhop(v_j, d)$, then inequality 5.4 must be satisfied. We restate that inequality for the sake of clarity.

$$delay\_so\_far + D(active\_node, lc\_nhop) + least\_delay\_value(lc\_nhop, d) \leq \Delta$$

Note that $delay\_so\_far = Delay(P_j)$ and $active\_node = v_j$ and $lc\_nhop = v_{j+1}$. Therefore, inequality 5.4 can be rephrased as:

$$\begin{aligned} Delay(P_j) + D(v_j, v_{j+1}) + least\_delay\_value(v_{j+1}, d) &= \\ Delay(P_{j+1}) + least\_delay\_value(v_{j+1}, d) &\leq \Delta \end{aligned} \tag{5.7}$$

The other alternative for DCUR is to proceed from v_j by adding the first link along the LD path, i.e., $v_{j+1} = least\_delay\_nhop(v_j, d)$. In this case,

$$least\_delay\_value(v_j, d) = D(v_j, v_{j+1}) + least\_delay\_value(v_{j+1}, d), \tag{5.8}$$

and we can restate inequality 5.6 as:

$$\begin{aligned} Delay(P_j) + D(v_j, v_{j+1}) + least\_delay\_value(v_{j+1}, d) &= \\ Delay(P_{j+1}) + least\_delay\_value(v_{j+1}, d) &\leq \Delta. \end{aligned} \tag{5.9}$$

In both cases, v_{j+1} becomes the next *active_node*. It follows from inequalities 5.7 and 5.9 that the subpath from s to *active_node* is part of at least one delay-constrained path towards d. DCUR stops only when $active\_node = d$. □

Theorem 5.2 *The final path constructed by DCUR for a given source s and destination d does not contain any loops.*

Proof. We use the same notation used in the proof of theorem 5.1. Let $V_j = \{v_0, \ldots, v_j\}$ be the set of nodes in the subpath P_j. All nodes in V_j have a routing table entry for the source-destination pair, s and d. The active node, v_j, adds a link (v_j, v_{j+1}). If $v_{j+1} \in V_j$, a loop is created. Node v_{j+1} becomes the next active node. Node v_{j+1} searches its routing table for an entry corresponding to s and d. If $v_{j+1} \in V_j$, it will find such an entry, thus detecting a loop. We proved that when a link (v_j, v_{j+1}) is added that creates a loop, node v_{j+1} will always detect that loop.

Next we prove that when node v_{j+1} detects a loop, it calls a process that correctly breaks that loop. When v_{j+1} detects a loop, it sends a *Remove_Loop* message back to v_j. Node v_j's reaction to the receipt of the *Remove_Loop* message depends on the $flag$ in the routing table entry corresponding to s and d. In all cases, node v_j removes the link (v_j, v_{j+1}) from the path being constructed. This is sufficient to correctly break the detected loop. $\Box$

Theorem 5.3 *The execution time of DCUR for a given source s and destination d is always finite.*

Proof. If no delay-constrained paths exist, then DCUR fails immediately at the source after determining that inequality 5.3 is not satisfied. If inequality 5.3 is satisfied, then DCUR proceeds. If no loops occur, then, after adding at most $(|V|-1)$ links, DCUR reaches the destination d. It remains to prove that even if loops occur, DCUR will still reach d within finite time. A subpath $P_j = \{v_0 \rightarrow \ldots \rightarrow v_i \rightarrow \ldots \rightarrow v_j\}$ ends with a loop if $v_j = v_i$ where $0 \leq i < j$ and $v_0 = s$. When the size of the network, $|V|$, is finite, the maximum number of distinct subpaths starting at s and ending with a loop is finite. Therefore, it is sufficient to prove that DCUR never attempts to construct the same subpath ending with a loop twice. When node $v_j = v_i$ detects a loop at the head of a subpath P_j^{LOOP}, it calls the loop removal procedure which traverses the path P_j^{LOOP} backwards removing links until a link $e_k^{LC} = (v_k, v_{k+1})$ is reached that is on the LC path direction from v_k towards d, where $i \leq k < j$. Link e_k^{LC} is removed from the path and path construction resumes by adding the link on the LD path direction from v_k towards d, link e_k^{LD}. One necessary condition to reconstruct P_j^{LOOP} is to readd link e_k^{LC} to the path being constructed. This means that a loop must occur, and to remove that loop DCUR removes link e_k^{LD}. However loop removal can not stop immediately after removing e_k^{LD}, because it is on the LD path direction towards d. Therefore loop removal must continue backwards until a link e_l^{LC} on the LC path direction from node v_l towards d is reached, where $0 \leq l < k$. DCUR removes link e_l^{LC}. Then path construction resumes and link e_k^{LC} may be readded to the subpath being constructed. Therefore, after a link e_k^{LC}, originally on a path P_j^{LOOP}, is removed from the path, it can be readded to the path only if a link e_l^{LC} is removed, where $0 \leq l < k < j$. The same holds for link e_l^{LC}. It follows that, the exact same subpath P_j^{LOOP} can not be reconstructed twice during the execution of DCUR. $\square$

5.5 Complexity of DCUR

Each time a node receives a *Construct_Path* message or a *Remove_Loop* message, it performs a fixed amount of computations, irrespective of the size of the network. Therefore, the computational complexity of the proposed distributed algorithm at any node is $O(1)$ time[2].

We now consider the worst case message complexity of DCUR, i.e., the number of messages needed in the worst case, in order to construct a path for a given source-destination pair. If no loops occur, then the number of messages needed to construct a path is proportional to the number of links in the path. This is because a node running DCUR exchanges at most three messages (one *Query* message, one *Response* message, and one *Construct_Path* message) to add one link. For a network size of $|V|$ nodes, the longest possible loop-free path from source to destination consists of $|V|$ nodes and $(|V|-1)$ links. Therefore the number of messages needed in the worst case is $O(|V|)$, if it is guaranteed that no loops will occur. Unfortunately, the occurrence of loops complicates the analysis.

The tree of the LC paths from any node in the network to the destination node d, denoted *LCTREE*, consists of $(|V|-1)$ links. Similarly, the tree of the LD paths from any node in the network to the destination d, denoted *LDTREE*, also consists of $(|V|-1)$ links. The union of these two trees is a subnetwork $N' = (V, E')$, where $(|V|-1) \leq |E'| \leq 2 * (|V|-1)$, because some links may be members of both trees.

[2] In this analysis we consider the complexity of DCUR only. We do not consider the complexity of the underlying mechanisms for maintaining and updating the cost vectors and delay vectors.

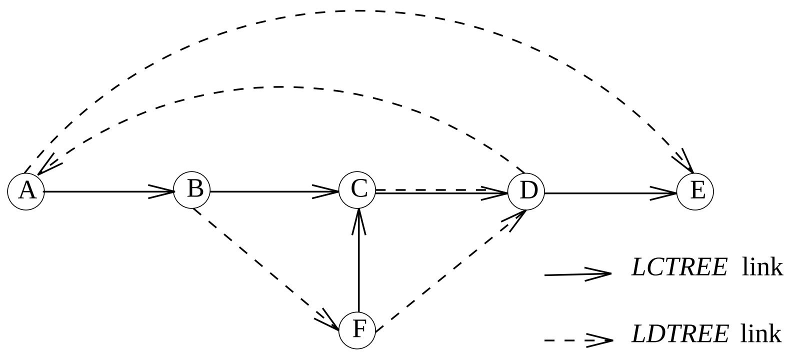


Figure 5.3: Example of a subnetwork constructed by taking the union of the *LCTREE* and the *LDTREE*. The destination is node E.

Figure 5.3 shows an example of the union of an *LCTREE* and an *LDTREE*. In this example, the link (C, D) is a member of both trees. The $|E'|$ links are the only links considered by DCUR when constructing a path from a source s to the destination d, because, as has been explained before, at any node DCUR considers only the LC path direction and the LD path direction towards the destination.

Let the links of the *LCTREE* be called tree links. We add the links of the *LDTREE* to the *LCTREE* to obtain the subnetwork N'. The links of the *LDTREE* which are not already in the *LCTREE* will be classified into one of the following three link types.

- A back link which is traversed from a node to one of its ancestors[3]. A back link may result in a loop.
- A descendent link goes from a node to one of its descendants other than its child. A descendent link may provide one or more nodes with two alternate paths towards the destination.
- A cross link connects two nodes such that neither is a descendant of the other. A cross link may provide one or more nodes with two alternate paths towards the destination[4].

[3]A node v is an ancestor of a node w in the *LCTREE* if w is on the path from v to the destination d. If v is an ancestor of w then w is a descendant of v. If the link (v, w) is a tree link, then w is v's child. In the *LCTREE* each node, other than d, has only one child.

[4]See, for example [97], for more detailed discussion of the terms: back link, descendent link, and cross link.

In the example of figure 5.3, links (A, B), (B, C), (C, D), (D, E), and (F, C) are tree links. The link (D, A) is a back link. Links (A, E) and (F, D) are descendent links, and the link (B, F) is a cross link.

A subnetwork N' has X back links, Y descendent links, and Z cross links where $0 \leq X, Y, Z \leq (|V| - 1)$ and $(X + Y + Z) \leq (|V| - 1)$. Adding a back link to a path under construction may or may not result in a loop. Since we are studying the worst case, we assume that adding a back link to a path always results in a loop. Consider a back link, e. Link e may be added and removed from the path being constructed several times, if it is reachable via multiple alternate paths from the source node. A loop results each time e is added. The back link e is reachable via $(Y + Z)$ alternate paths in the worst case. This happens when the $(Y + Z)$ descendent links and cross links are upstream (closer to the source node) from the back link e. In this case, each time DCUR attempts to use one of the $(Y + Z)$ resulting alternate paths, it may continue downstream (towards the destination) and add the link e, thus creating a loop. If DCUR attempts to use all $(Y + Z)$ alternate paths while constructing the delay-constrained path, the link e will be added and removed $(Y + Z)$ times, which means that $(Y + Z)$ loops will be created and removed during path construction. The example of figure 5.3 is not a worst case scenario. However, it shows how the back link (D, A) can be reached via three alternate paths when node A is the source. The first alternative is the original path along the *LCTREE*: $\{A \rightarrow B \rightarrow C \rightarrow D\}$. The second alternative was created due to the addition of the cross link (B, F), and it is $\{A \rightarrow B \rightarrow F \rightarrow C \rightarrow D\}$. The final alternative is $\{A \rightarrow B \rightarrow F \rightarrow D\}$. This path was brought to existence by the descendent link (F, D).

So far we considered only one back link. However, the subnetwork N' contains X back links. In the worst case, each of the X back links is reachable via $(Y + Z)$ alternate paths. In this case we may end up with $X * (Y + Z)$ loops. Since $(X + Y + Z) \leq (|V| - 1)$, it follows that, in the worst case, DCUR may create and remove $O(|V|^2)$ loops before completing the construction of the delay-constrained path.

The largest possible loop consists of $(|V| - 1)$ nodes and $(|V| - 1)$ links (the destination can not be part of a loop in DCUR). A maximum of three messages are needed to add one loop link. Thus it takes $O(|V|)$ messages to create the largest loop. One message is needed for removing one loop link, which means that at most $O(|V|)$

messages are needed if all loop links have to be removed before path construction resumes. Therefore, $O(|V|)$ messages are needed, to create and remove the largest loop. In the worst case $O(|V|^2)$ loops may be created and removed. This means that, in the worst case, DCUR needs $O(|V|^3)$ messages to handle loops. Compared to the $O(|V|)$ messages required to add the permanent links that constitute the final loop-free path, it is obvious that, in the worst case, loop handling dominates the operation of DCUR, and that the overall worst case message complexity of DCUR is $O(|V|^3)$. Fortunately, our simulation results show that DCUR's average performance is much better than the worst case just studied. These results will be presented in the next section.

5.6 Simulation Results

We used simulation for our evaluation of the average performance of DCUR. Full duplex, directed, simple, connected, random networks identical to the random networks described in section 3.1 were used in the experiments. We varied the size of the networks from 20 nodes up to 200 nodes and used a fixed average node degree of 4. The networks spanned an area of $4000 * 2400$ Km^2, and 155 Mbps links were used. Under these assumptions, the propagation components dominated the link delays. A link's cost was defined as a function of that link's utilization. It was set equal to the sum of the equivalent bandwidths of the traffic streams traversing that link. Link costs were asymmetric, because $C(u, v)$ and $C(v, u)$ were independent. However the link delays were symmetric, because the propagation delay in both directions of a full duplex link are equal. Two experiments were conducted over these networks.

5.6.1 The Average Message Complexity of DCUR

In the first experiment, we measured the average number of messages DCUR requires to establish a delay-constrained path. For each run of the experiment, we generated a random set of links to interconnect the fixed nodes, we selected a random source node, we selected a random destination node, and we generated random background traffic to utilize each link. The cost (utilization) of a link was a random variable

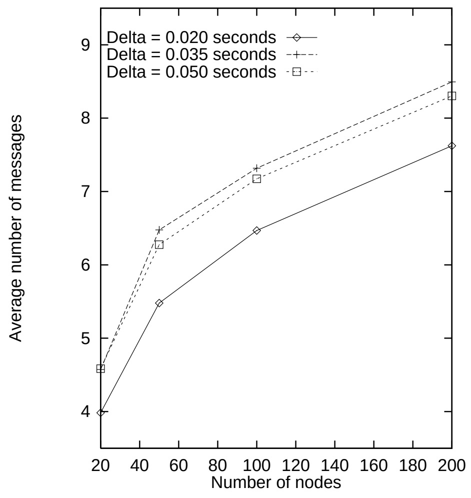


Figure 5.4: Average number of messages, variable network size, average node degree 4, three delay constraint settings: 0.02 seconds, 0.035 seconds, and 0.05 seconds.

uniformly distributed between 5 Mbps and 125 Mbps. The experiment was repeated with network sizes ranging from 20 nodes up to 200 nodes. We also varied the delay constraint value from 0.015 seconds to 0.055 seconds. The delay constraint represents only an upper bound on the end-to-end propagation time across the network. Relatively small values were chosen for the delay constraint in order to allow the higher level end-to-end protocols, at both the source and the destination, enough time to process the transmitted information without affecting the quality of the real-time session. In this experiment, we measured the average number of messages exchanged between the nodes which execute the distributed DCUR algorithm to construct a delay-constrained path. Note that any message generated by DCUR at some node is always destined to an immediate neighbor of that node. Therefore, any DCUR generated message travels a distance of one only hop. Unless otherwise stated, DCUR was run repeatedly until confidence intervals of less than 5% of the mean value, using 95% confidence level, were achieved for all measured values presented in this subsection and in the next subsection. At least 500 different networks were simulated for each measured value.

Figure 5.4 shows the average number of messages versus the size of the network for three different values of the delay constraint: a strict delay constraint of 0.02

seconds, a moderate delay constraint of 0.035 seconds, and a lenient delay constraint of 0.05 seconds. All three curves of figure 5.4 indicate clearly that the average number of messages grows very slowly with the size of the network. For any of the delay constraint values shown in the figure, doubling the size of the network increases the average number of DCUR's messages by roughly one message only. Thus the average growth rate of the number of messages is roughly logarithmic in the network size, for the experiments we ran.

The number of messages exchanged while constructing a path is smallest when the delay constraint value is small. This is due to the following:

- A path that satisfies the strict delay constraint consists on the average of fewer links than a path that satisfies a lenient delay constraint.
- DCUR is forced to follow the LD path direction most of the time in order to satisfy the strict delay constraint. Therefore, the probability of the occurrence of a loop is small. As has been discussed in the previous section, the occurrence of loops increases the number of messages.

When the delay constraint is increased to 0.035 seconds, the number of messages is largest. One reason is that the average number of links on the solution path is greater than it is when the delay constraint is only 0.02 seconds[5]. The other reason is that 0.035 seconds is a moderately strict delay constraint, and DCUR may be able to follow the LC path direction at some nodes and to follow the LD path direction at others. This toggling between LC path direction and LD path direction increases the probability of loop occurrence, and hence increases the average number of messages exchanged.

Increasing the delay constraint further, from 0.035 seconds to 0.05 seconds, leads to a reduction in the average number of messages. 0.05 seconds is a lenient delay constraint. Thus DCUR is able to follow the LC path direction most of the time without violating the delay constraint, and therefore it no longer toggles between the LC path direction and the LD path direction. The consequence is that loops occur rarely. Of course, when the delay constraint is increased to 0.05 seconds, the average

[5]For a 200-node network the average number of links per path is 4.28 for a 0.02 seconds delay constraint, 4.72 for a 0.035 seconds delay constraint, and 5.12 for a 50 seconds delay constraint.

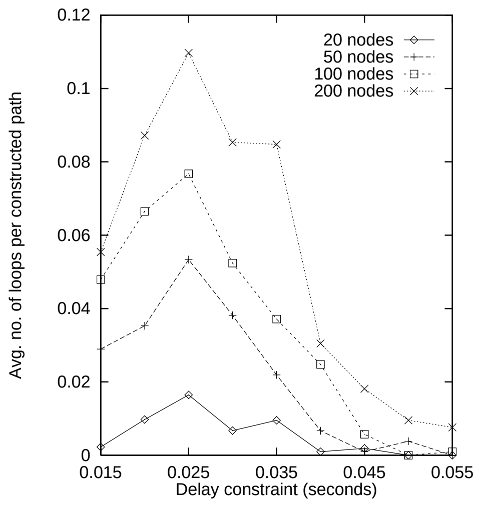


Figure 5.5: Average number of loops occurring while constructing a single delay-constrained path, network sizes of 20 nodes, 50 nodes, 100 nodes, and 200 nodes, average node degree 4, variable delay constraint.

number of links per path also increases, but figure 5.4 indicates that the lack of loop occurrence is the dominant effect in this case.

It has been mentioned in the previous section that when the LC path direction and the LD path direction from a node towards the destination coincide, then there is no need for sending the *Query* message and waiting for the *Response* message. Both messages can be eliminated, and only one message is needed to add the next link, namely the *Construct_Path* message. Otherwise, three messages are needed to add one link. Our simulation results show that, at many nodes, the LC path direction and the LD path direction towards the destination are identical, and thus eliminating the *Query* and *Response* messages at such nodes, results in a significant reduction in the number of messages per added link. For example, figure 5.4 shows that for a 200-node network and a delay constraint of 0.05 seconds, on the average 8.3 messages are needed to construct the path. We also measured the average number of links on the path, and it is 5.12 links. Thus $(8.3/5.12) = 1.62$ messages are needed per added link assuming that no loops occur. This is more than a 45% reduction when compared to the need for 3 messages per added link if the *Query* and *Response* messages were always exchanged.

In order to verify our assumption, that loops occur most frequently when the delay constraint is moderately strict, we measured the average number of loop occurrences during one successful run of DCUR, i.e., a run that successfully constructs a delay-constrained path for a given source-destination pair. We found that loops do not occur frequently (less than 12 loops every 100 successful runs of DCUR). Therefore, it was not possible (due to the excessive simulation times) to repeat the experiment until small enough confidence intervals were achieved for the measured values of the average number of loop occurrences. 1,000 successful runs of DCUR were simulated for each point in figure 5.5. Figure 5.5 shows the average number of loop occurrences per successful run of DCUR versus the delay constraint for different network sizes. Figure 5.5 shows that loops occur most frequently when the delay constraint value ranges from 0.02 seconds to 0.045 seconds. When the delay constraint is lenient (larger than 0.045 seconds) loop occurrences are very infrequent, less than one loop every 100 successful runs of DCUR. The average number of loop occurrences also decreases when strict delay constraint values of less than 0.02 seconds are used. It is obvious from figure 5.5 that loops do not occur frequently for any of the network sizes we simulated. However, the figure also indicates that loops occur more frequently as the size of the network increases.

5.6.2 Comparison to Other Algorithms

In this subsection, we show the results of the second experiment which compares DCUR with two algorithms that are also suitable for delay-sensitive applications. The first algorithm is the LD path algorithm, LD. LD is optimal with respect to the end-to-end delay, but it does not attempt to minimize the cost of the constructed path. Therefore, it may result in inefficient utilization of the link bandwidth. The other algorithm is CBF which was briefly described in section 1. CBF constructs the optimal DCLC path, but it is centralized and its execution time grows exponentially with the network size. Note that CBF and LD are the only existing unicast routing algorithms capable of satisfying the delay requirements of real-time applications.

The structure of the second experiment is similar to that of the first experiment. The only difference is that for each randomly selected source-destination pair we applied DCUR, LD, and CBF, one at a time, to construct the delay-constrained path

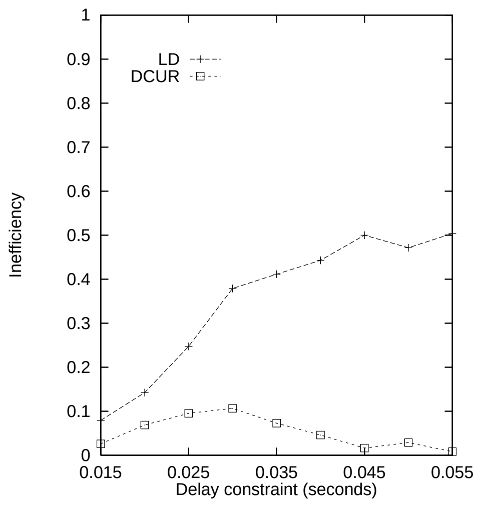


Figure 5.6: Inefficiency, 200-node networks, average node degree 4, variable delay constraint.

from source to destination. For each algorithm, we measured the average path cost, the average inefficiency relative to CBF, the average end-to-end delay, and the success rate (how frequently the algorithm succeeds in finding a delay-constrained path). The average inefficiency of an algorithm x is defined as:

$$inefficiency_x = \frac{(cost_x - cost_{CBF})}{cost_{CBF}} \tag{5.10}$$

We show the results of the experiment for 200-node networks and a variable delay constraint. Figure 5.6 shows the average inefficiency of LD and DCUR relative to CBF. When the delay constraint is small (0.015 seconds), the number of alternate delay-constrained paths, available for the algorithms to choose from, is small, and therefore the differences between the algorithms are also small. For delay constraint values between 0.02 seconds and 0.045 seconds, DCUR is up to 10% worse than the optimal CBF. The reason is that, because of the tight delay constraint, DCUR can not always follow the unconstrained LC path direction. In some cases, it has to follow the LD path direction instead. The toggling between these two directions affects DCUR's ability to create low-cost paths. However, DCUR remains on the average more efficient than LD. When the value of the delay constraint exceeds 0.045 seconds, its effect on the constructed path is minimal. In that range, DCUR's inefficiency approaches zero, because it almost exclusively elects to follow the LC path direction.

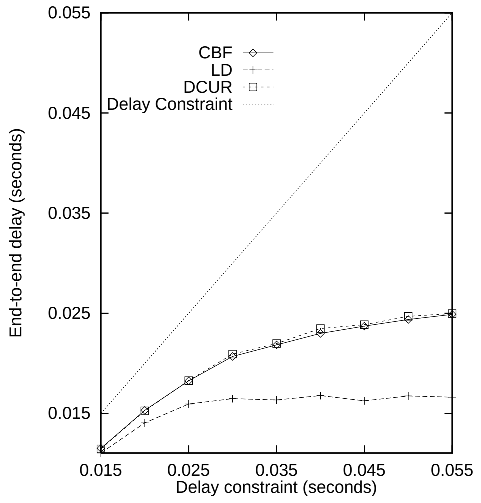


Figure 5.7: Average end-to-end delay, 200-node networks, average node degree 4, variable delay constraint.

LD does not attempt to minimize the path cost at all. That's why its inefficiency is up to 50% when the delay constraint value is large.

Figure 5.6 indicates that the path cost of DCUR is always within 10% from the path cost of the optimal CBF. Thus DCUR's cost performance is quite satisfactory, especially when considering that CBF is a centralized algorithm that requires global information about the network topology, while DCUR is a distributed heuristic that requires only limited information to be maintained at each node (one cost vector and one delay vector).

The average end-to-end delays of DCUR and CBF are considerably larger than the minimal delays achieved by LD as shown in figure 5.7. This is not a big advantage for LD, though. More important is that all three algorithms have the same success rate in satisfying the imposed delay constraint. The success rates of the three different algorithms are identical, because all of them are always capable of constructing a delay-constrained path, if one exists. All three algorithms achieve a 100% success rate when the delay constraint value is greater than 0.03 seconds. As the delay constraint value decreases below 0.03 seconds the success rate decreases indicating that delay-constrained solutions do not always exist for networks spanning large areas when the delay constraint value is sufficiently small.

In addition to the 200-node networks, we simulated 20-node, 50-node, and 100-node networks as well. The results for the different network sizes are similar to results shown above for the 200 nodes case. This indicates that the performance of the different algorithms relative to each other does not depend significantly on the network size.

5.7 Conclusions

We studied the delay-constrained unicast routing problem in point-to-point connection-oriented networks. Our work was motivated by the fast evolution of delay-sensitive distributed applications. We formulated the problem as a delay-constrained least-cost path problem, which is known to be *NP*-complete. Therefore we proposed a distributed, source-initiated heuristic solution, the delay-constrained unicast routing (DCUR) algorithm, to avoid the excessive complexity of the optimal solutions. DCUR is always capable of constructing a delay-constrained path within a finite time, if one exists, for a given source-destination pair. DCUR requires only a limited amount of information at each node. The information at each node is stored in a cost vector and a delay vector. These vectors are constructed and maintained in exactly the same manner as the distance vectors which are widely deployed over current networks. DCUR is capable of detecting and eliminating any loops that may occur while it constructs a delay-constrained path. We proved the correctness of DCUR by showing that it is always capable of constructing a loop-free delay-constrained path within a finite time, if such a path exists. The number of computations at each node participating in the path construction process is fixed, irrespective of the network size. The worst case message complexity of DCUR is dominated by the occurrence and removal of loops. It requires $O(|V|^3)$ messages to construct a single path in the worst case. Fortunately, however, our simulation results show that DCUR requires much fewer messages on the average, because loop occurrence is rare in realistic networks. We compared the performance of DCUR to CBF, which is an optimal DCLC algorithm with running times that grow exponentially with the size of the network. We also compared DCUR to LD, a shortest path algorithm that minimizes the end-to-end delay, but does not attempt to minimize the path cost. Our results indicated that

DCUR yields satisfactory performance with respect to both path cost and path delay. Our evaluation of the cost performance of the algorithms showed that DCUR is always within 10% from the optimal CBF, while LD is up to 50% worse than optimal in some cases. In summary, DCUR is a simple, efficient, distributed algorithm that scales well to large network sizes.

Chapter 6

Shared Multicast Trees and the Center Selection Problem

So far in our study of multicast routing algorithms, we considered only one source per multicast session (see the definitions in section 2.3). Many network applications involve multiple sources and multiple receivers, e.g., videoconferencing. The multicast routing problem for these applications is known as the many-to-many problem. There are two alternate approaches to address this problem. One approach is to consider the many-to-many problem as multiple one-to-many multicast routing problems, and to construct a source-specific multicast tree for each source. The other approach is to use a single multicast tree (or only a few multicast trees, as will be discussed later in this chapter). In this approach, traffic streams from multiple sources share the links of the same tree and hence the name shared multicast trees.

6.1 Shared Multicast Trees versus Source-Specific Multicast Trees

Both source-specific multicast trees and shared multicast trees have their advantages and disadvantages. Shared multicast trees have the following advantages over source-specific multicast trees.

- It takes less overhead to construct and maintain one shared tree per multicast session than to construct a source-specific multicast tree for every source transmitting to that session. With shared multicast trees, when a new source starts transmitting to an already existing multicast session, it does not have to construct an entire source-specific tree. It merely has to find a path to connect itself to the existing shared tree of that session. Similarly, when a node joins an existing multicast session as a receiver, it merely has to find a path to connect itself to that session's shared tree. If source-specific trees are used instead, the new receiver will have to join the source-specific trees of each source transmitting to the already existing multicast session.

- When using shared multicast trees, a source node does not have to keep an explicit list of the members of the multicast session. Similarly, when a node is a member in a multicast session, it does not have to keep an explicit list of all sources.

On the other hand, shared multicast trees suffer from the following disadvantages as compared to source-specific multicast trees.

- High traffic concentration. Traffic streams from different sources share the links of the same tree, which results in high traffic concentration on these links. The traffic of a given multicast session will be concentrated on the links of the shared tree.
- End-to-end delays along shared trees are longer than the corresponding delays when source-specific trees are used [23]. An example, comparing the delays along a shared tree to the delays if source-specific trees are used instead, is given in figure 6.1.

We have already studied the problem of constructing source-specific multicast trees in previous chapters. In the remainder of this dissertation, we study the problem of constructing shared multicast trees.

6.2 The Center Selection Problem

All multicast routing protocols which permit the use of shared trees designate a node for each shared tree to be its center. In PIM-SM [74, 72], the center of a shared tree is denoted as the RP (rendezvous point), while in CBT [78, 75], the center is denoted

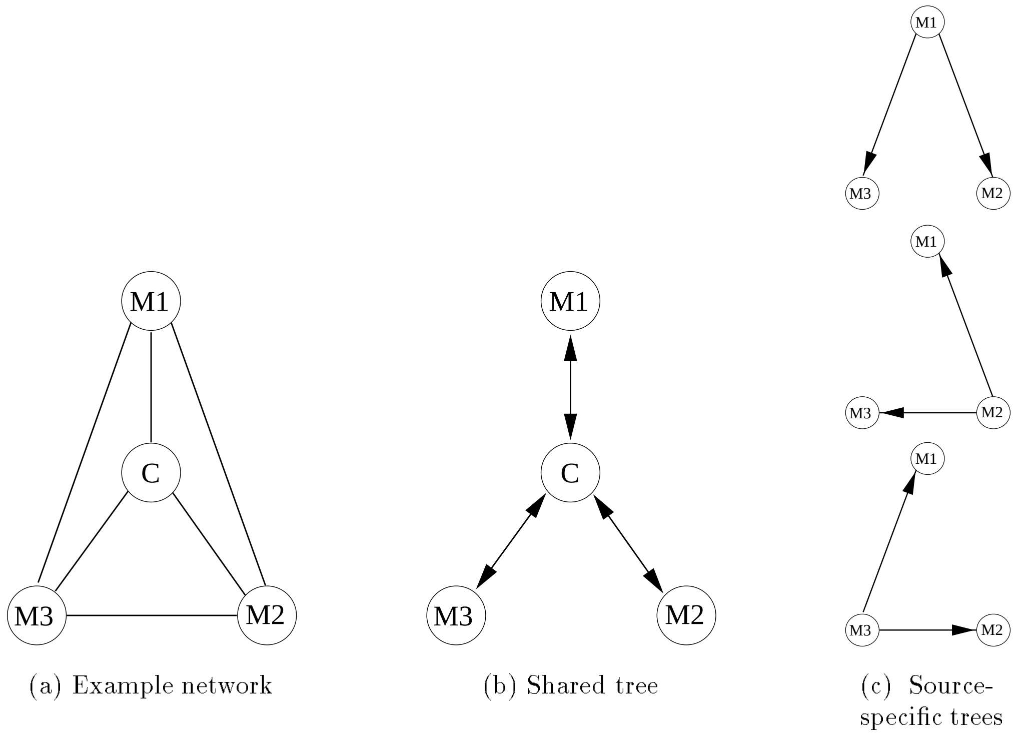


Figure 6.1: Comparison of the end-to-end delays along shared trees and source-specific trees. Unit delays are assigned to all links. The multicast group $G = \{M_1, M_2, M_3\}$. Multicast sources M_1, M_2, and M_3. Maximum delay along the shared tree is 2. Maximum delay along the source-specific trees is 1.

as the core. In general, a shared tree does not have to have a center. However, designating a node to be its center facilitates several session management operations. If a node is selected to be the center of a shared tree, it performs all the operations performed by the other nodes in the tree, but it may have additional responsibilities. For example, it may be responsible for expanding the tree when new receivers join the multicast session, or it may be responsible for collecting traffic from all sources and processing that traffic before multicasting it to all destinations.

Finding the optimal shared tree is difficult. Therefore, most previous work on shared multicast trees in a communication networking context has partitioned the problem into two separate subproblems: the center selection problem and the route selection problem. Most heuristics for shared multicast tree construction start by selecting a node to be the center, and then they select routes to construct a shared tree around that center.

The center selection problem and the route selection problem can be addressed independently. The same routing algorithms we considered when studying the source-specific multicast routing problem are used to select routes for the construction of shared multicast trees. Therefore, our work on shared multicast trees will focus on the center selection problem only. The performance of a center selection algorithm is evaluated based on the quality of the resulting shared multicast tree. Since our focus is on the center selection problem, we use a fixed algorithm for the route selection problem (a variation of the delay-constrained unicast routing algorithm presented in chapter 5), in order to guarantee a fair comparison of the different center selection algorithms we study.

In their specification of shared multicast trees, PIM-SM and CBT propose the use of more than one center per multicast session. Using multiple centers per multicast session may serve one or more of the following purposes.

- Fault tolerance. If one center fails, nodes connected to that center may be able to connect themselves to another center.
- Multiple centers may be used to achieve different QoS levels. In this case, each center administers a shared tree that guarantees a different QoS level.
- In some cases, a single center, and its shared tree, can not satisfy the delay constraint (the upper bound on the end-to-end delay from any source to any destination) of a multicast session while multiple centers can.

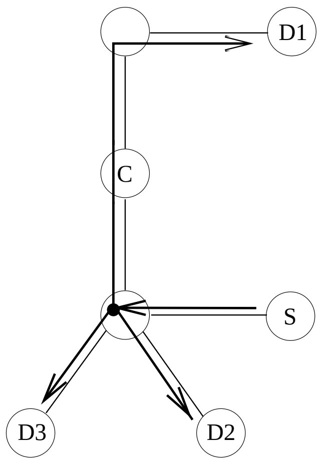
Our investigation of the center selection problems will be subject to the delay constraint, since it is the main QoS parameter we consider in this dissertation. In the remainder of this chapter, we present a classification of multicast centers, and survey related work on shared multicast trees and the center selection problem. In chapter 7, we propose algorithms for selecting a single center, and evaluate the resulting shared multicast trees based on their cost performance and their ability to satisfy the delay constraints imposed by real-time applications. In chapter 8, we study the case when a single center can not satisfy the delay constraint, and we propose algorithms to compute the minimum number of centers needed to satisfy the delay constraint and to determine the location of these centers.

6.3 Classification of Multicast Centers

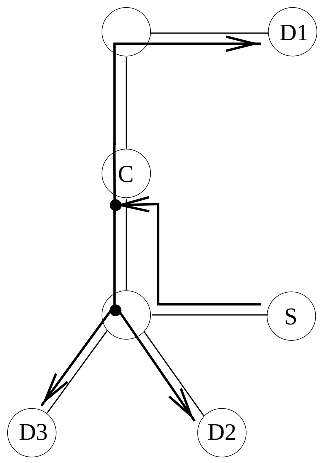
Two types of multicast centers have been proposed in the literature: administrative centers and distribution centers. An administrative center is responsible for maintaining the shared multicast tree. Its responsibilities include managing the join and leave operations of multicast group members and multicast sources and advertising the status of the multicast session to all nodes in the network. A distribution center, on the other hand, is responsible for collecting traffic streams from all sources and then multicasting them to all destinations. A distribution center may process the source's packets before multicasting them to the destinations. For example, in some videoconferencing applications, the distribution center selects only the video stream from one source and multicasts it to all destinations, and it discards all other streams.

Construction of the multicast connections is straightforward in case of distribution centers. The shortest path (or a variation of the shortest path) is used to transmit the traffic from each source to the center and then the minimum Steiner tree (or a variation of the minimum Steiner tree) is used to multicast traffic from the center to all group members. This scheme was proposed in [98] for a class of videoconferences. The authors selected the node with the minimum sum of shortest paths costs to all participants in the videoconference to be the distribution center.

In shared multicast trees with administrative centers, the traffic from a source is forwarded to each multicast group member over the shortest path to that member, along the shared tree. Such a path does not necessarily have to pass through the center. Figure 6.2 illustrates the difference between an administrative center-based tree and a distribution center-based tree. Figure 6.2(a) shows a shared multicast tree with an administrative center. The source S multicasts its packets to the destinations directly. The packets traverse the tree from S to D_2 and D_3 directly without passing through the center first. Figure 6.2(b) shows a shared multicast tree with a distribution center. The source S unicasts its packets to the center C. C may perform some application-specific processing on the packets it receives. Then it multicasts them to all destinations. It is obvious from this example that administrative centers achieve more efficient utilization of the network bandwidth, since no packet is ever forwarded over both directions of a full duplex link as may happen in the case of distribution



(a) C is an administrative center.



(b) C is a distribution center.

Figure 6.2: Example illustrating the difference between an administrative center and a distribution center. Node S is a source, nodes D_1, D_2, and D_3 are destinations, and node C is the center of the shared multicast tree.

centers. In addition, distribution centers may perform application-specific processing at the application layer. Therefore distribution centers are suitable only for a specific class of applications. In our work on center selection, we consider only administrative centers. Both the RPs of PIM-SM [74] and the cores of CBT [78] are administrative centers.

6.4 Related Work

First, we review related work from graph theory and operations research. Then, we survey work proposed specifically for shared multicast trees and the center selection problem in communication networks.

6.4.1 Graph Theory

Graph theory provides the following useful definitions that can be used when investigating the center selection problem [99].

- The eccentricity of a node is defined as the shortest distance to the node farthest away from that node. The center of a network is the node with minimum eccentricity. A network may have multiple centers. The radius of a network is the minimum eccentricity at any node in the graph. It occurs at the center nodes. The diameter of a network is the maximum eccentricity at any node in the graph.
- The median of a network is a node for which the sum of the distances to all other nodes in the network is minimum. A network may have multiple medians.

The above definitions apply to both directed and undirected networks. Breadth-first search can be used to compute the centers and medians of a network if all links span unit distance, i.e., the network is unweighted. For weighted networks, shortest path algorithms can be used for the same purpose.

Finding the center is more suitable for our work than finding the median, since we use an upper bound on end-to-end delay as a QoS constraint, and therefore our objective is to limit the maximum end-to-end delay and not to minimize its average value.

6.4.2 Operations Research

Many variations of the center selection problem for networks have been investigated in operations research. Problems considering only one center and problems for selecting multiple centers were studied [100]. Emergency facility location [101], hazardous facility location [102], and defensive facility location to minimize competition [103] are just a few examples of the problems studied. There is an entire research area called “Location Research”. Location problems in networks can be classified into two main categories. One category is the minisum location problems which find one (or more than one) median for a given network. The other category is the minimax location problems which find one (or more than one) center for a given network.

All operations research variations of the center location problem, which we are aware of, consider situations in which all services either originate from the center or terminate at the center. This means that the center is either the only source or the only destination. Our many-to-many multicasting problems are different. Traffic may originate from multiple source and it must be delivered to multiple destinations. In addition, the center does not necessarily have to be a source or a destination.

An interesting version of the multiple center selection problem was presented by Toregas and et al. [104]. Their problem was to locate the minimum number of emergency facilities such that the maximum delay from the closest emergency facility to any node in the network is less than a given value. The authors solved this minimax problem using zero-one integer programming.

6.4.3 Communication Networks

Intensive development efforts are currently under way in the standard bodies to evolve efficient scalable multicast routing protocols, and shared multicast trees are included in the specifications of these protocols. The shared tree modes of PIM-SM [74] and CBT [78] have already been described in section 2.6. In both protocols, receivers join the shared multicast tree via the forward shortest paths towards the center, and sources transmit to the shared tree via the forward shortest paths towards the center. Thus receivers receive the sources' traffic streams via the shortest reverse paths from the center. The difference between the shared modes of PIM-SM and CBT is mainly in the mechanisms used to maintain the tree.

CBT permits the use of multiple cores. A receiver has to join only one core, and a source unicasts its packets towards one core only. As soon as the packets arrive at any node in the shared tree, they are multicast towards all destinations. In CBT, there is one primary core. The other cores join the primary core via the shortest paths to construct a core backbone. The result is a single shared tree with multiple cores.

An earlier draft of PIM-SM [105] permitted the use of multiple RPs per group. Each source sends packets towards each of the RPs, but receivers only join towards a single RP. This results in each RP having its own shared tree that spans only a subset of the multicast group members. The motivation for having multiple RPs was to achieve fault tolerant operation in case of RP failures. The most recent draft of PIM-SM [74], however, no longer permits the use of multiple RPs. Currently, a multicast session selects an RP list from a prespecified RP set. Only one node from the RP list is the active RP. Not much robustness is sacrificed, since the other nodes in the RP list act as multiple backup RPs, and recovery time after an RP failure is small.

In PIM-SM an RP is chosen randomly from the candidate RPs list, and in CBT cores are placed by hand based on the topological distribution of the group membership at session initiation time. We are not aware of any current efforts at the protocol development level aimed at proposing center selection protocols. However, there has been some previous work on center selection at the research level. We will review this work in the remainder of this section.

Wall's work. David Wall investigated the problems of broadcasting and multicasting (selective broadcasting) in his PhD dissertation [23]. His main focus was on broadcasting and not multicasting. The only cost function he considered was link delays. Therefore his shortest path trees were least-delay trees. He dedicated two chapters for broadcasting and multicasting over shared trees. Wall proposed different criteria to be optimized in a shared tree, e.g., the average end-to-end delay or the maximum end-to-end delay. He also described the optimum shared tree corresponding to each criterion he proposed. The optimum solutions were complicated and in some cases even *NP*-complete. To avoid this excessive complexity, Wall proposed three heuristic algorithms to select the center of a broadcast (or a multicast) tree.

- The first heuristic selects the node to be the center whose shortest path tree spanning all members has the least maximum delay to any member. This is the graph theoretic center.
- The second heuristic selects the node to be the center whose shortest path tree spanning all members has the least average delay to all members. This is the graph theoretic median.
- The third heuristic selects the node to be the center whose shortest path tree spanning all members has the least diameter. Wall defined the diameter of a tree as the maximum delay between any two nodes in the tree.

In his dissertation, Wall also compared the delays along shared trees to the delays if source-specific shortest path trees were used instead. He established several upper bounds on the average and maximum delays that can be achieved by shared trees. His most important result is that if the shortest path tree of a randomly chosen center node is used as a shared tree, then the maximum delay between any two nodes along that tree is at most two times the maximum delay achieved if source-specific trees were used instead.

Finally, Wall proposed a distributed implementation scheme for his three heuristics. This scheme consists of two phases. First, each node in the network computes its criterion (maximum delay, average delay, or diameter) based on the information it has. Then each node broadcasts its criterion value to all other nodes. The values are compared distributedly and the node with best criterion is selected to be the center. To implement his scheme efficiently, Wall proposed that when a node receives the criterion values from two or more other nodes, that node should compare all these values and only continue broadcasting the best value it received so far.

Wei and Estrin's work. Wei and Estrin [57] studied the tradeoffs between source-specific multicast trees and shared multicast trees using simulation. A shared tree was compared to the corresponding source-specific trees based on the maximum end-to-end delay, the average end-to-end delay, the tree costs, and the resulting traffic concentration. They defined maximum traffic concentration as the number of traffic streams from different sources which traverse the same link. The authors used both real networks and random networks of up to 200 nodes in their simulations. Each link had a delay and a cost. Several shared tree algorithms and source-specific algorithms were studied. Some algorithms optimized cost while others optimized delay.

Wei and Estrin's simulation results showed that source-specific least-delay trees achieve, on the average, up to 20% smaller maximum delays and up to 30% less traffic concentration than shared trees. However, the cost of a shared trees was approximately 10% less than that of a source-specific least-delay tree.

The authors also proposed algorithms that restrict the choice of the multicast centers to the set of multicast group members, and they showed that this restriction does not significantly affect the performance of the resulting shared trees.

Shukla, Boyer, and Klinker's work. Shukla et al. [106] proposed a protocol for constructing multicast trees in asymmetrically loaded networks. Their protocol allows the use of either source-specific trees or shared trees. The authors defined the cost of a link as a function of both the link's delay and its utilization. They proposed a center selection protocol for shared multicast trees based on a tournament. A simplified version of the tournament works as follows. Each receiver is paired with a source in decreasing order of distance. The node at the middle of the shortest path between each pair is the winner of that pair. All winners are then paired together, and the next group of winners is computed, and so on until only one winning node remains. This node is the center of the shared tree. The authors also proposed a shared tree routing algorithm to be applied after selecting the center. Unlike PIM and CBT, the resulting shared trees are completely forward shortest path trees. However, the term "tree" is not appropriate to describe the output of the proposed routing algorithm. Suppose we are given a node n that is both a source and a receiver and also given a center c. In an asymmetric network the forward shortest path from the source n to the center c is not necessarily the same as the forward shortest path from the center c to the receiver n. Since the two paths are not necessarily identical, but their end points are the same, their superposition may result in a loop. The authors attempt to include both paths in their routing structure, and therefore the result may contain loops, and hence it may not always be a tree. The authors' experience indicates that constructing a shared tree in asymmetric networks and trying to optimize it in both directions, from all sources towards the center and from the center towards all receivers, is very complicated.

Calvert, Zegura, and Donahoo's work. Calvert, Zegura, and Donahoo [107] compared different algorithms for selecting a center with respect to their effect on bandwidth and delay. They used simulation over random networks in their study. They concluded that there is no single best algorithm for selecting a center. Trade-offs between performance (bandwidth and delay) and the required information must be considered when choosing a center selection algorithm. The authors also studied the effects of center selection algorithms on traffic concentration. They showed that when center selection algorithms, which distribute the centers uniformly over all nodes in the network, are used, the traffic concentration resulting from shared trees is as good as the traffic concentration resulting from source-specific trees. In another paper, Donahoo and Zegura [108] proposed an algorithm that permits dynamic center migration in order to efficiently support multicast sessions in which sources and destinations are allowed to join and leave dynamically.

Thaler and Ravishankar's work. Thaler and Ravishankar [109] proposed two distributed center selection protocols and two versions of each protocol. The two versions differ in the amount of information each of them uses for its computation. Either one of the proposed protocols can be applied to select more than one center for a given multicast session. However, these protocols do not attempt to distribute the centers evenly throughout the network. The proposed protocols allow the centers to migrate from one node to another dynamically as the group membership changes or the load on the network changes. The authors evaluated their algorithms and most of the previously proposed algorithms using simulation over random networks. In their simulations, they selected only one center for each multicast session. The authors also compared the amount of information required at each node for distributed implementation of the previously proposed algorithms.

6.5 Conclusions

In this chapter, we presented the motivation for using shared multicast trees, and introduced the related center selection problem. We discussed the tradeoffs between shared trees and source-specific trees. Researchers have already compared the performance of both approaches using simulation. We studied source-specific trees in the first part of this dissertation. Starting from this chapter, we turned our attention to shared multicast trees. We reviewed previous work on shared trees and center selection. Simple, distributed algorithms and protocols for center selection already exist. However, none of the existing algorithms attempts to optimize the utilization of the network bandwidth subject to delay constraints imposed by real-time applications. Most previous work focused on selecting a single center. The problem of selecting multiple centers has received inadequate consideration from communication network researchers. However, variations of this problem have been investigated in operations research. In the next two chapters, we investigate the problems of selecting a single center and selecting multiple centers for real-time applications with delay constraints in a high-speed networking environment.

In chapter 7, we formulate the optimal single shared multicast tree problem subject to delay constraints and prove that it is *NP*-complete. We then propose heuristics for selecting a single center and evaluate the performance of the resulting shared trees using simulations.

In chapter 8, we show that a single shared multicast tree with a single center may not always be able to satisfy the delay requirements of real-time applications. Then we study the problem of finding the minimum number centers (with each center administering a separate shared tree that spans a subset of the multicast group members) capable of satisfying these delay requirements. Once again, the optimal solution is *NP*-complete, so we propose heuristic solutions, and we evaluate their performance using simulation.

Chapter 7

Single Shared Multicast Tree Problem with Delay Constraint

In this chapter, we investigate the use of a single shared multicast tree per multicast session. We consider resource-extensive, real-time applications with end-to-end delay constraints. Our objectives are to manage the network resources efficiently and to satisfy the applications' delay constraints. In section 7.1, we present our assumptions and formulate the problem as a diameter-constrained minimum Steiner tree problem. The diameter of a shared tree is the maximum delay along the tree from any source to any destination. Existing heuristics for constructing unconstrained shared multicast trees start by selecting a center, then apply a routing algorithm to construct a tree around that center. We propose heuristics that follow the same steps. Our focus is on center selection. Therefore, we study different center selection algorithms, but we always use the same routing algorithm.

The routing algorithm we use is a variation of the delay-constrained unicast routing algorithm, **DCUR**, we proposed in chapter 5. We allow group members to join and leave the shared tree dynamically. A new member joins the existing tree by constructing a delay-constrained path towards the center such that the diameter of the resulting tree does not violate the delay constraint. We describe the routing algorithm in section 7.2.

Different center selection heuristics are discussed in section 7.3. In addition to the previously known heuristics, which perform center selection independent of and preceding routing, we propose an approach that starts by routing an initial tree, then selects a center from the nodes on that initial tree, and finally continues routing the shared tree. All center selection heuristics and the routing algorithm considered in this chapter are distributed. In section 7.4, we evaluate the performance of the different center selection heuristics using simulation. Finally, we give concluding remarks in section 7.5.

7.1 Problem Formulation

We use the same definitions of a network, link costs, link delays, and multicast group previously given in section 2.3. However, we limit our investigation of shared multicast trees to symmetric networks only to avoid the difficulties which faced Shukla et al. [106], when studying shared multicast trees in asymmetric networks (see section 6.4.3). The shared trees they proposed contain loops, and thus require complicated routing table structures and complicated forwarding mechanisms. We, therefore, restrict ourselves to symmetric networks and postpone generalizing our results to asymmetric networks for future work.

Figure 7.1 shows an example of a shared multicast tree in a symmetric network. When a packet arrives at node in the tree, this node multicasts the packet on all links of the tree except the incoming link. Each link on the shared tree represents a symmetric full duplex link.

We take the following two measures to preserve the symmetricity of the network as shared multicast trees are dynamically established and torn down.

- The number of sources and destinations varies from one many-to-many multicast application to another. For our investigations, we consider a conferencing application in which all multicast group members are both sources and destinations. The multicast group is given as $G = \{g_1, g_2, \ldots, g_n\}$ where $n = |G| \leq |V|$ is the size of the multicast group.
- We use a dynamic link cost metric that is a function of the link utilization. When reserving resources for a multicast session, we reserve the same amount of resources on both directions of each full duplex link in the shared tree carrying that session's traffic streams. These resources are to

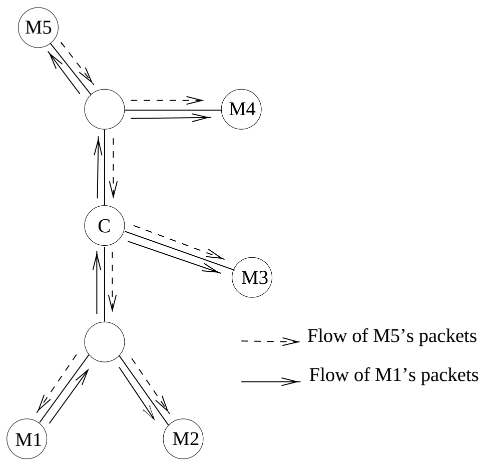


Figure 7.1: Example of the packet flow on a shared tree. C is a control center. All group members are both sources and destinations. The multicast group is $G = \{M_1, M_2, M_3, M_4, M_5\}$. Only the flow of M_1's packets and M_5's packets is shown.

be shared among the traffic streams from all sources[1].

A shared multicast tree $T(G) \subseteq E$ is a tree spanning all members of the group G such that any leaf node of $T(G)$ is a member of G. The total cost of a shared tree $T(G)$ is simply the sum of the costs of all links in that tree.

$$Cost(T(G)) = \sum_{e \in T(G)} C(e) \tag{7.1}$$

The diameter of the shared tree $T(G)$ is the maximum delay along the tree links between any two multicast group members.

$$Diam(T(G)) = \max_{g_i \in G}(\max_{g_j \in G}(\sum_{e \in P_T(g_i,g_j)} D(e))) \tag{7.2}$$

where $P_T(g_i, g_j)$ is the path from g_i to g_j along the tree $T(G)$.

Our objective is to minimize the total cost of the shared tree such that its diameter does not violate the end-to-end delay constraint, Δ, imposed by the application. The optimal problem can be formulated as a diameter-constrained minimum Steiner tree problem. For a given undirected network $N = (V, E)$, a nonnegative cost $C(e)$ for

[1]This resource reservation scheme resembles the shared reservation style of RSVP [110]. It is suitable for applications involving voice traffic only. Other reservation styles are better suited for video traffic and other types of traffic, but they do not preserve the symmetricity of the link loads.

each $e \in E$, a nonnegative delay $D(e)$ for each $e \in E$, a multicast group $G = \{g_1, \ldots, g_n\} \subseteq V$, and a positive delay constraint Δ, the constrained minimization problem is:

$$\min_{T(G) \in \mathcal{T}'(G)} \quad Cost(T(G)) \tag{7.3}$$

where $\mathcal{T}'(G)$ is the set of trees spanning all nodes in G for which the diameter is bounded by Δ. $\mathcal{T}'(G)$ is a subset of $\mathcal{T}(G)$, $\mathcal{T}'(G) \subseteq \mathcal{T}(G)$, where $\mathcal{T}(G)$ is the set of trees spanning all nodes in G. If $T(G) \in \mathcal{T}(G)$ then $T(G) \in \mathcal{T}'(G)$ if and only if

$$Diam(T(G)) \leq \Delta. \tag{7.4}$$

Theorem 7.1 *The diameter-constrained minimum Steiner tree problem is NP-complete.*

Proof. We prove that a decision version of the problem is *NP*-complete.

The Diameter-Constrained Minimum Steiner Tree Decision Problem: *Given an undirected network $N = (V, E)$, a nonnegative cost $C(e)$ for each $e \in E$, a nonnegative delay $D(e)$ for each $e \in E$, a multicast group $G = \{g_1, \ldots, g_n\} \subseteq V$, a positive delay constraint Δ, and a positive real value B, is there a tree spanning all group members such that $Cost(T(G)) \leq B$ and $Diam(T(G)) \leq \Delta$?*

The problem is clearly in *NP*, because a nondeterministic algorithm can guess a set of links to form the tree. Then it is possible to verify in polynomial time that these links do form a tree, that this tree spans all nodes in the group G, that the total cost of the tree is $\leq B$, and that the diameter of the tree is $\leq \Delta$.

Next we restrict the diameter-constrained minimum Steiner tree decision problem to a known *NP*-complete problem, the unconstrained minimum Steiner tree decision problem. This can be done by setting the delay constraint to a value greater than or equal to the largest simple path delay between any two nodes in the network. For the restricted case, a solution to the unconstrained minimum Steiner tree decision problem is a solution to the diameter-constrained minimum Steiner tree decision problem. Since the restricted problem is *NP*-complete, it follows that the general diameter-constrained minimum Steiner tree decision problem is also *NP*-complete. □

We implemented an optimal algorithm for the diameter-constrained minimum Steiner tree problem using a branch and bound technique. We denote this optimal algorithm **DiamOPT**. However, this algorithm has excessive execution times, and we could apply it only to small networks. Therefore, it is useful only for the purpose of benchmarking heuristics.

As we mentioned at the beginning of this chapter, we study shared tree construction heuristics which consist of two phases: a center selection phase followed by a route selection phase. Our focus is on center selection. Therefore, we use a fixed routing algorithm, which is a variation of the delay-constrained unicast routing algorithm presented in chapter 5.

7.2 The Routing Algorithm

We study dynamic shared trees, i.e., multicast group members are allowed to join or leave the multicast tree at any time. The routing algorithm we use is the delay-constrained shared tree routing algorithm, **DCSHARED**, which consists of two parts: **DCJOIN** which is called when a new member joins an existing shared multicast tree and **DCLEAVE** which is called when a node leave an ongoing multicast session. DCJOIN is a member-initiated variation of DCUR. Similar to DCUR, DCJOIN is based on delay vectors and cost vectors. These two data structures have already been described in section 5.2. We assume that these vectors are always up to date, and do not change during the operation of DCJOIN. Both DCUR and DCJOIN consist of a forward phase (path construction phase) and a reverse phase (acknowledgment phase). The path construction phases of both algorithms are almost identical. The difference between the two algorithms is that, in DCJOIN, a node m joins an already existing shared multicast tree by constructing a delay-constrained path towards its center c using a DCJOIN delay constraint value of $\Delta' \leq \Delta/2$, where Δ is the application's delay constraint[2]. We assume that the address of the center c and the value of the application's delay constraint are known at all nodes. The path construction phase terminates when DCJOIN reaches a node that is already a member in the shared tree such that the delay from c to m does not exceed Δ'. Figure 7.2(a)

[2]Reasons for this particular choice of Δ' will be discussed in section 7.3.

gives an example of the operation of the path construction phase of DCJOIN. Node m attempts to join an already existing shared multicast tree with center c. The path construction phase does not stop at node y, because the delay from c to m via y exceeds Δ'. However, path construction stops when g_2 is reached, because the delay from c to m via g_2 does not exceed Δ'. Note that the resulting tree at the end of the path construction phase contains a loop, $\{c \leftrightarrow z \leftrightarrow g_2 \leftrightarrow y \leftrightarrow v \leftrightarrow c\}$. Any loops existing at the end of the path construction phase are removed during the acknowledgment phase. The final shared tree after node m joins the session is shown in figure 7.2(b). Similar to DCUR, DCJOIN takes $O(|V|^3)$ messages[3] in the worst case. This brief description of DCJOIN is sufficient for our purposes.

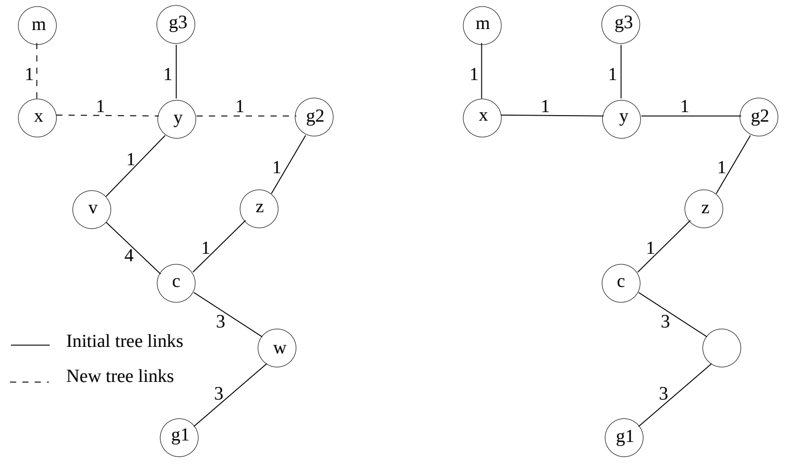
When a node leaves a multicast group, the path leading to that node in the shared multicast tree is pruned using DCLEAVE. The operation of DCLEAVE is straightforward. Complete pseudo code of DCSHARED and the routing table structures it uses is given in appendix C.

7.3 Heuristics for Center Selection

All heuristics presented in this chapter are distributed and their implementation is based on the information available in the delay vectors and cost vectors existing at all nodes. Thus the center selection algorithms we study do not require more network state information than the routing algorithm discussed in the previous section.

The first center selection heuristic, simply chooses a random node to be the center of the multicast session. We call this heuristic **RAND**. Of course implementation of RAND does not even require the information available in the delay vectors and cost vectors. One way to implement RAND distributedly is to install the same random center generator at all nodes. When a node receives the group address and the initial multicast group members of a new multicast session, it applies this information to the random center generator which computes the address of the center node. This approach does not require any message overhead, because each node selects the center independently. A similar approach is used to choose the RPs in PIM-SM [74].

[3]We define a message as traversing only a single link. If it is forwarded over another link after that, it is a counted as another message.



(a) At the end of the path construction phase

(b) At the end of the acknowledgment phase

Figure 7.2: Example of the operation of DCJOIN. Node c is the center. The application's delay constraint $\Delta = 12$. DCJOIN's delay constraint $\Delta' = \Delta/2 = 6$. The initial tree (shown in solid lines in subfigure (a)) spans three multicast group members: g_1, g_2, and g_3. Only link delays are shown. Node m joins the shared tree. The final tree is shown in subfigure (b).

The second heuristic for selecting a center is the **MinMaxD** heuristic. Each node v computes a criterion value. In this case the criterion is the maximum end-to-end delay from v to any multicast group member. This value can be obtained by comparing the entries of the delay vector corresponding to the group members. This requires $O(|G|)$ time in the worst case, where $|G|$ is the size of the multicast group. After computing its criterion value, v broadcasts this value to all other nodes. When a node w receives node v's criterion value, it continues forwarding it downstream along the broadcast tree only if node v's criterion value is smaller than the criterion values node w has received from any other nodes so far. Each node saves the lowest criterion value it received so far and the address of the node advertizing that value. In the worst case, each node's broadcast message will reach all other nodes in the network. Therefore $O(|V|^2)$ messages are required in the worst case to select the center. After all broadcast operations are complete, all nodes will have the same minimum criterion value saved. The node which advertized that value, i.e., the node with minimum maximum delay to any multicast group member, is the selected center. This approach for distributed center selection has been proposed by Wall [23]. The same heuristic can be implemented using different criteria such as the average delay to the multicast group members, the average path cost to the group members, or the maximum path cost to any group member. However, the maximum delay to any multicast group member is the best suited criterion for our work, because we study delay-constrained shared trees, and one of our objectives is to limit the maximum delay.

After selecting a center for a new multicast session using either RAND or MinMaxD, that center is initially the only node in the shared multicast tree. Other group members can then join the shared tree using DCSHARED with a delay constraint of $\Delta' = \Delta/2$. By enforcing a maximum delay of $\Delta/2$ from the center to any node in the shared tree, it can be guaranteed that the diameter of the shared tree will never exceed Δ. When RAND is used together with DCSHARED to construct delay-constrained shared multicast trees, we denote the overall heuristic **RAND-DCSHARED**. Similarly, when combining MinMaxD and DCSHARED we denote the resulting heuristic **MinMaxD-DCSHARED**.

To avoid the use of a separate center selection phase preceding the routing phase, we propose to construct an initial shared multicast tree that spans a subset of the group members, then select one of the nodes in that tree to be the multicast center, and finally permit the remaining multicast group members to join the shared tree using the DCSHARED heuristic. We propose a heuristic called **DCINITIAL** to construct the initial delay-constrained shared tree and to select its center, and we call the overall shared tree construction heuristic **DCINITIAL-DCSHARED**. DCINITIAL starts by selecting a random multicast group member. That random member searches its delay vector to find the group member farthest away in terms of delay. A delay-constrained least-cost path is constructed between these two nodes using a variation of DCUR. This path constitutes the initial shared tree. Finally, the node on that path closest to its middle is selected to be the center of the shared tree, and the address of the selected center is advertized to all nodes in the network. The operation of DCINITIAL is very similar to the operation of DCUR. Constructing the initial delay-constrained path takes $O(|V|^3)$ messages in the worst case. Selecting the

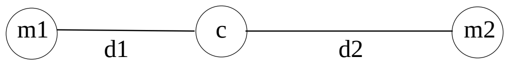


Figure 7.3: The initial shared multicast tree constructed by DCINITIAL between two multicast group members m_1 and m_2. The initial tree is a delay-constrained path $\{m_1 \leftrightarrow c \leftrightarrow m_2\}$. Links (m_1, c) and (m_2, c) may represent multihop paths. The values d_1 and d_2 represent the delay along the constructed path.

center requires traversing the constructed path backwards which takes $O(|V|)$ messages. Broadcasting the address of the selected center requires $(|V| - 1)$ messages. Thus, overall DCINITIAL requires $O(|V|^3)$ messages in the worst case to construct an initial shared tree that spans at least two multicast group members and to select the multicast center, and it eliminates the need for a separate center selection phase.

While selecting the center, DCINITIAL also computes the value of the constraint Δ' to be used by multicast group members not already in the shared tree to join the shared tree using DCJOIN. The computed value of Δ' is advertized to all nodes in the network together with the address of the center. Figure 7.3 shows an example of an initial delay-constrained path between two group members. Because of the delay constraint, $d_1 + d_2 \leq \Delta$, and hence $0 \leq d_1, d_2 \leq \Delta$. We can not use $\Delta' = \Delta/2$, because, if $d_1 > \Delta/2$ or $d_2 > \Delta/2$, we may end up with a tree diameter that violates the application's delay constraint Δ. Therefore, Δ' is computed as follows.

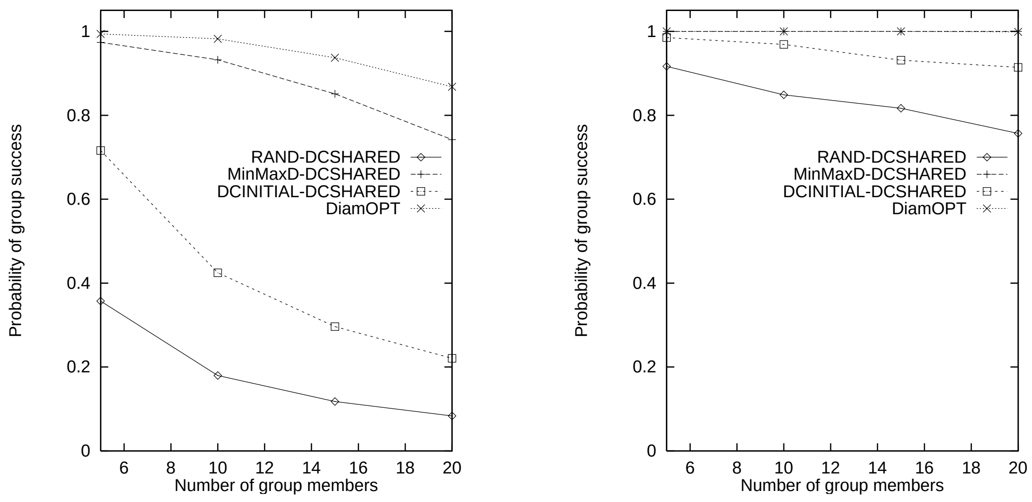
$$\Delta' = \min(\frac{\Delta}{2}, \Delta - \max(d_1, d_2)) \qquad (7.5)$$

Complete pseudo code of DCINITIAL is given in appendix D. In the previous part of this chapter, we formulated the delay-constrained shared multicast tree problem as a diameter-constrained minimum Steiner tree problem which is *NP*-complete. Therefore, its optimal solution, DiamOPT, is not suitable for large networks. Then we proposed three heuristics to solve the same problem: RAND-DCSHARED, MinMaxD-DCSHARED, and DCINITIAL-DCSHARED. In the next section, we evaluate the performance of all these algorithms using simulation.

7.4 Simulation Results

The objective of this section is to evaluate the performance of the different delay-constrained shared multicast tree construction algorithms. We ran several simulation experiments. Full duplex, simple, connected, random networks similar to the random networks described in section 3.1 were used in the experiments. We varied the size of the networks from 20 nodes up to 100 nodes and used a fixed average node degree of 4. The node positions were fixed for each network size and the links interconnecting these nodes were generated using our random link generator described in section 3.1.1. The networks spanned an area of $4000 * 2400$ Km^2 (0.02*0.012 seconds^2 in terms of propagation delay), and 155 Mbps links were used. Under these assumptions, the propagation components dominated the link delays, and hence the link delays were symmetric, i.e., $D(u, v) = D(v, u)$. The link cost was defined as a function of the link utilization. It was set equal to the sum of the equivalent bandwidths of the traffic streams traversing a link. Symmetric link costs were used, i.e, $C(u, v) = C(v, u)$. Thus the resulting networks were undirected.

The first experiment compares the different algorithms when each of them is applied to create a delay-constrained shared multicast tree for a given multicast group. Recall that we only study the scenario in which all multicast group members are both sources and receivers. The optimal algorithm, DiamOPT, is static. It fails if it does not succeed in constructing a delay-constrained tree spanning all group members. On the other hand, the three heuristics, RAND-DCSHARED, MinMaxD-DCSHARED, and DCINITIAL-DCSHARED, are dynamic. They can construct delay-constrained shared trees which span only a subset of the multicast group. For a fair comparison between DiamOPT and the three heuristics, we considered a run of an algorithm successful only if it results in a tree spanning all multicast group members. For each run of the experiment we generated a random set of links to interconnect the fixed nodes, random background traffic for each link, and we selected a random multicast group. The equivalent bandwidth of each link's background traffic was a random variable uniformly distributed between B_{min} and B_{max}. The experiment was repeated with different multicast group sizes and different delay constraint values. We measured the probability of group success of an algorithm and the cost of the shared tree. The

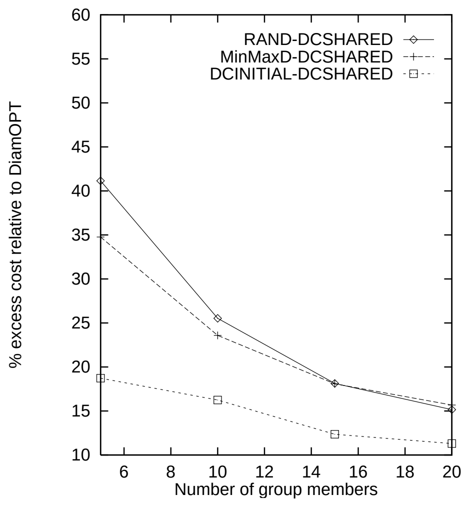


(a) $\Delta = 0.035$ seconds (b) $\Delta = 0.05$ seconds

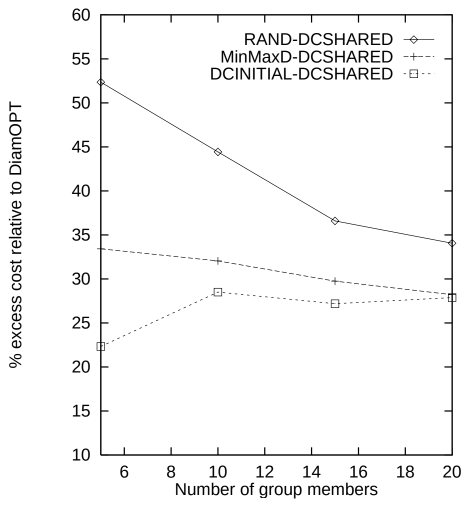
Figure 7.4: Probability of group success of shared multicast tree construction algorithms, 20 nodes, average degree 4, $B_{min} = 5$ Mbps, $B_{max} = 125$ Mbps .

probability of group success is defined as the probability that an algorithm succeeds in constructing a delay-constrained shared tree spanning all group members. The experiment was run repeatedly until confidence intervals of less than 5%, using 95% confidence level, were achieved for all measured quantities. At least 500 networks were simulated for each measurement.

Figure 7.4 shows the probability of group success of the different algorithms versus the group size for 20-node networks for two settings of the delay constraint Δ, 0.035 seconds and 0.05 seconds. The probability of group success decreases as the multicast group size increases. DiamOPT always succeeds in finding a solution if one exists. Unfortunately, none of the three heuristics is guaranteed to find a solution if one exists. MinMaxD-DCSHARED achieves probabilities of group success within 10% from DiamOPT when Δ is set to 0.035 seconds. Both DiamOPT and MinMaxD-DCSHARED are always successful when Δ is relaxed to 0.05 seconds. RAND-DCSHARED and DCINITIAL-DCSHARED have very low probabilities of group success when Δ is 0.035 seconds. We tried to repeat this experiment using a delay constraint value of 0.02 seconds, but the probabilities of group success of all algorithms were very low,



(a) $\Delta = 0.035$ seconds



(b) $\Delta = 0.05$ seconds

Figure 7.5: Total cost of a shared multicast tree relative to DiamOPT, 20 nodes, average degree 4, $B_{min} = 5$ Mbps, $B_{max} = 125$ Mbps.

so we could not obtain good confidence intervals. MinMaxD-DCSHARED's good performance indicates clearly the advantage of using delay as a criterion for selecting the multicast center, particularly for small delay constraint values.

The percentage excess cost of a shared multicast tree relative to the cost of the optimal tree is given in figure 7.5. The costs of all three heuristics increase relative to optimal when the delay constraint is relaxed from 0.035 seconds to 0.05 seconds. DCINITIAL-DCSHARED has the best cost performance of the three heuristics, but the difference between it and MinMaxD-DCSHARED is not more than 15%.

We repeated the first experiment using 100-node networks. Unfortunately, DiamOPT could not be applied to networks of that size due to its excessive running times. DCINITIAL constructs an initial shared tree, and RAND and MinMaxD also construct an initial shared tree consisting of the center node only. After that multicast group members, not already in the initial tree, attempt to join the tree one member at a time. When running the first experiment on 100-node networks, we measured the probability that a multicast group member succeeds in joining the shared tree. We call this the probability of member success. Figure 7.6 shows the probability of

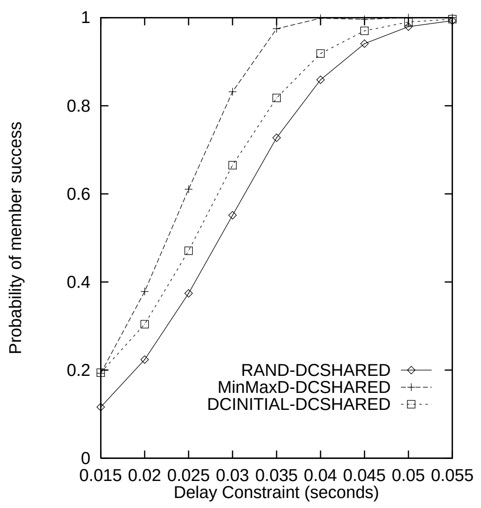


Figure 7.6: Probability of member success of shared multicast tree construction algorithms, 100 nodes, average degree 4, 25 multicast group members, $B_{min} = 5$ Mbps, $B_{max} = 125$ Mbps.

member success versus the delay constraint for a multicast group of 25 members. MinMaxD-DCSHARED's performs better than the other two heuristics. The probability of member success is independent of the group size, because the success of a member's attempt to join a shared tree, i.e., the success of the DCSHARED routing algorithm, depends only on the least-delay value between that member and the center. MinMaxD chooses the node with least maximum delay to any group member to be the center. Thus increasing the probability that DCSHARED succeeds. MinMaxD-DCSHARED's good performance with respect to enabling individual members to join the shared tree leads to its superior performance relative to the other heuristics with respects to successfully constructing a shared tree that spans all group members, as has been shown already in figure 7.4.

In the first experiment, we studied a single multicast group, and constructed a single shared multicast tree. MinMaxD-DCSHARED has higher probability of success than DCINITIAL-DCSHARED. However, the shared trees DCINITIAL-DCSHARED constructs are of lower costs. The performance of RAND-DCSHARED is poor with respect to both the probability of success and the tree cost.

We repeated the first experiment using a different version of the DCSHARED routing algorithm. In that version, the delay constraint Δ' varies dynamically as nodes join and leave the shared multicast tree. We denote this version of the algorithm as DCSHARED-V to distinguish it from the original DCSHARED algorithm in which a fixed delay constraint $\Delta' = \Delta/2$ is used. Of course, permitting Δ' to vary dynamically complicates the operation of the routing algorithm and requires additional information to be stored in the routing table entries. However, using a variable Δ' permits nodes that are farther away from the center than $\Delta/2$ to join the tree. Assume that a node joins an existing shared tree via a path with delay $3\Delta/4$ from the center. Δ' must be set to $(\Delta - 3\Delta/4)$ after that to make sure that any subsequent join operation will not result in delay constraint violation. Therefore, DCSHARED-V may permit one far away node to join the shared tree, but as a result Δ' will be set to a very small value such that even nodes that are close to the center may fail to join the tree. Using a fixed Δ' prevents far away nodes from joining the tree, but, on the other hand, ensures that all nodes within $\Delta/2$ delay from the center can join the shared tree at any time. Figure 7.7 compares the probabilities of member success of the two versions of DCSHARED when used in conjunction with DCINITIAL. Using a variable Δ' does not improve the performance of the routing algorithm. In fact the original algorithm DCSHARED has slightly higher probability of member success than DCSHARED-V. We also measured the probability of group success and the tree cost for the two versions of DCSHARED. All results indicate that there is no improvement in performance that justifies the added complexity of DCSHARED-V. So we decided to use the simpler DCSHARED throughout the experiments.

We conducted a second experiment in which we started with an unloaded networks, and kept adding multicast groups and constructing the corresponding shared trees until the cumulative tree failure rate exceeded 15% (when this point was reached, we considered the network saturated). We considered a shared tree construction operation, and the corresponding multicast session, a failure if, after all multicast group members attempt to join the tree, the final tree spans less than two group members. The reason is that any useful communication activity requires at least two participants. A member node may fail to join the shared tree either because no delay-constrained path exists from that member to the center or because of link saturation. Bandwidth could be reserved on a link until its cost, i.e., the sum of the equivalent bandwidths of the sessions traversing that link, exceeded 85% of the link's

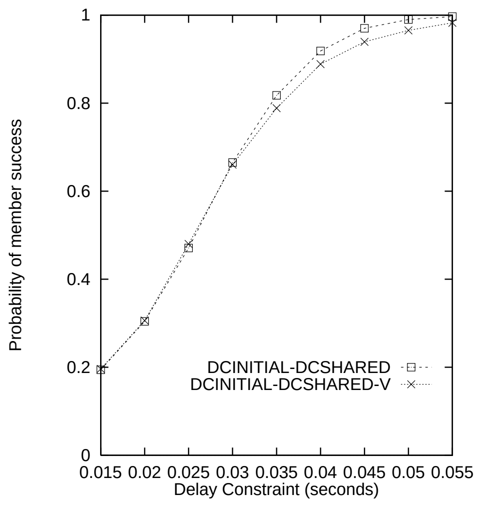


Figure 7.7: Probability of member success for two versions of the routing algorithm, fixed $\Delta' = \Delta/2$ (DCSHARED) versus variable Δ' (DCSHARED-V), 100 nodes, average degree 4, 25 multicast group members, $B_{min} = 5$ Mbps, $B_{max} = 125$ Mbps.

capacity, then the link got saturated. Before the routing algorithm, DCSHARED, added a link to the shared tree, it verified that sufficient bandwidth is available on that link to carry the traffic traversing that tree. When a link was added to a shared tree, a bandwidth of 5 Mbps was reserved on that link. We repeated the experiment with randomly chosen multicast groups of different sizes. Our objective was to determine how efficiently the shared tree construction algorithms manage the network bandwidth. The experiment was repeated, until the confidence intervals for all measured quantities were $< 5\%$ using the 95% confidence level. Similar to the first experiment, a random network was generated before each run of the experiment. This experiment could not be applied to the optimal algorithm, DiamOPT, because of its large execution times.

The first quantity we measured in the second experiment was the number of successfully admitted multicast sessions at the end of each run of the experiment. Figure 7.8 shows the measured values versus the multicast group size for 20-node networks and a delay constraint value of 0.035 seconds. As the multicast group size increases, the size of the shared tree increases, and hence the total amount of network bandwidth reserved for that tree also increases. Therefore the network saturates after admitting

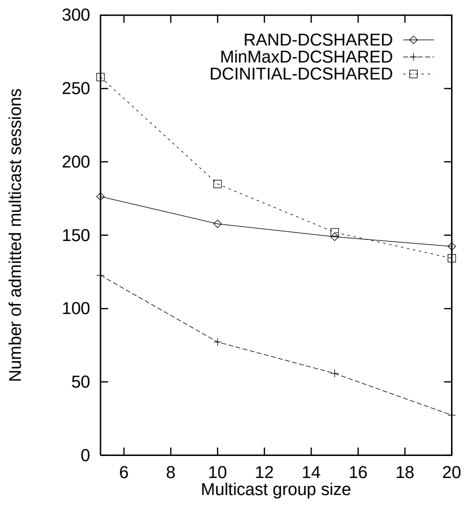


Figure 7.8: Number of admitted multicast sessions, 20 nodes, average degree 4, $\Delta = 0.035$ seconds.

less sessions. When the multicast group size is small, DCINITIAL-DCSHARED admits the largest number of sessions followed by RAND-DCSHARED. For large group sizes, both algorithms have comparable performance. MinMaxD-DCSHARED's performance is approximately 25% worse than DCINITIAL-DCSHARED's performance when the group size is small and its performance deteriorates fast as the group size increases. Measuring the number of successfully admitted sessions was not sufficient, since the size of the corresponding multicast trees may vary largely, because a multicast session was considered successful if the corresponding tree spanned at least two of its group members. Therefore, we also measured the number of admitted group members for all successfully admitted sessions. The number of admitted group members is shown in figure 7.9.

To analyze the behavior of the different algorithms in the second experiment, we measured the number of centers allocated to each node at the end of each run of the experiment. Note that the positions of the nodes are fixed, and only random links are created for each run. The fixed node positions are given in figure 3.1. Each node has an integer valued address as shown in the figure. Figure 7.10 shows the average number of centers allocated at each node.

Figure 7.10(a) shows that, as expected, the random center selection algorithm,

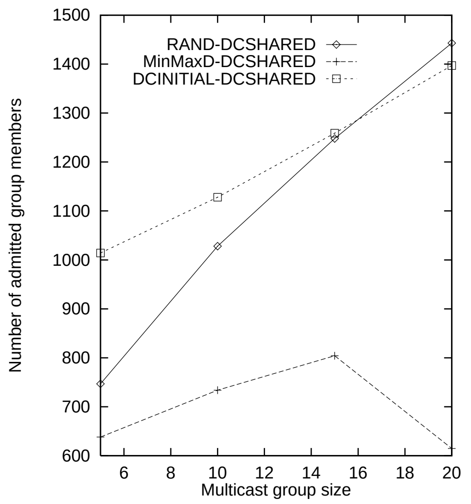
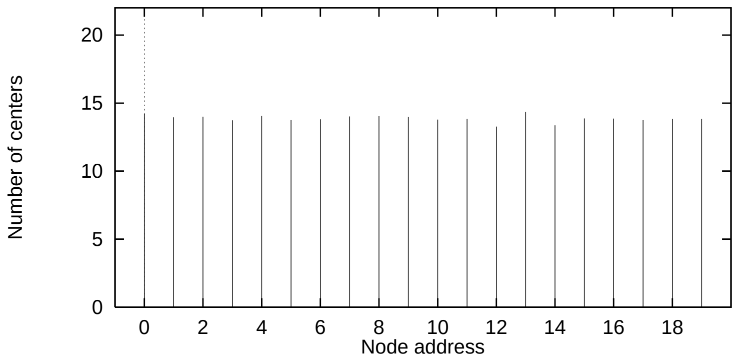


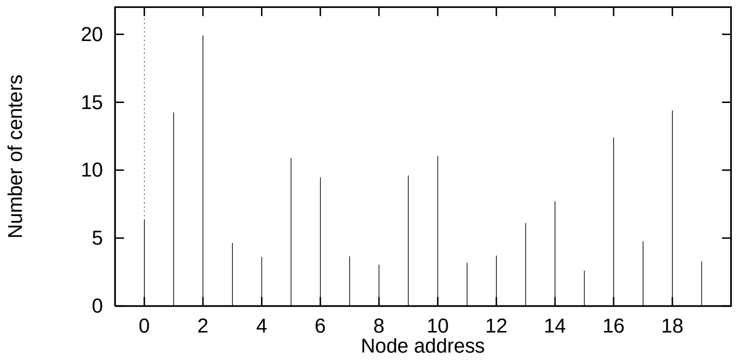
Figure 7.9: Number of admitted multicast group members, 20 nodes, average degree 4, $\Delta = 0.035$ seconds.

RAND, results in a uniform distribution of the centers over all nodes in the network. The uniform distribution of the centers leads to uniform load distribution over the links, and therefore the probability of traffic concentration at any individual links is very small. However, selecting the centers randomly may result in cases where the center is too far from the multicast group members and no delay-constrained paths exist from the center to some, or even any, group members. These cases occur more frequently when the multicast group size is small. That is why RAND-DCSHARED's performance is not as good as DCINITIAL-DCSHARED's performance for small group sizes.

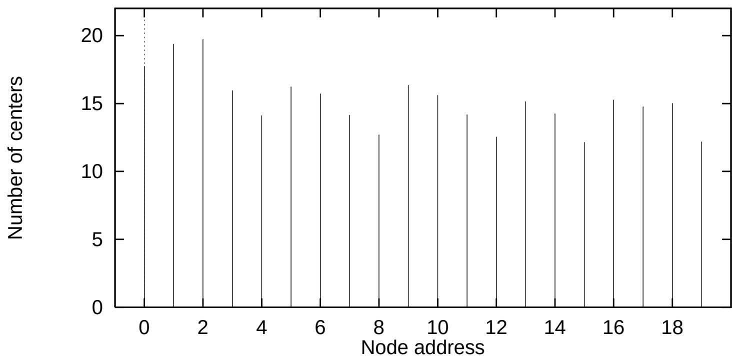
On the contrary to RAND, MinMaxD favors to select the center from among the nodes that are near the geometric center of the network, e.g., nodes 2 and 18, as can be seen from figure 7.10(b). The reason is that MinMaxD's center selection criterion is the maximum delay to any multicast group member. Since the group members are randomly selected using a uniform probability function, it is more probable that the nodes close to the geometric center will minimize the selection criterion. Therefore, MinMaxD-DCSHARED keeps allocating centers at these nodes. The results are heavy traffic concentration at the links attached to these nodes and unbalanced load distribution, and the network saturates after admitting much fewer sessions than the



(a) RAND-DCSHARED



(b) MinMaxD-DCSHARED



(c) DCINITIAL-DCSHARED

Figure 7.10: The average number of centers at each node at network saturation time for the 20-node network of figure 3.1, 5 multicast group members, $\Delta = 0.035$ seconds.

other algorithms.

Naturally, the number of admitted group members increases as the multicast group size increases as can be seen from figure 7.9. However, for MinMaxD-DCSHARED, this value drops when the group size approaches the network size, i.e., in the broadcast case. In the broadcast case, the group members are the same for all sessions. MinMaxD uses the static delay vectors for center selection. It keeps selecting the same node to be the center irrespective of the costs of the links attached to that node. This results in extremely heavy traffic concentration on these links, and very early network saturation. Thus MinMaxD-DCSHARED's performance in the broadcasting case is even worse than its performance in the general multicast cases.

DCINITIAL constructs an initial delay-constrained path and selects the node closest to the middle of that path to be the center. DCINITIAL is a delay-constrained least-cost routing algorithm. Therefore it avoids using heavily loaded, high-cost links. This is good for load balancing purposes. In addition, the success of DCINITIAL in constructing an initial path ensures that the final shared tree constructed by DCINITIAL-DCSHARED spans at least two group members. No such guarantee can be provided in case of RAND and MinMaxD. For these reasons, DCINITIAL-DCSHARED performs better than the other algorithms with respect to the number of admitted sessions (figure 7.8) and the total number of admitted group members (figure 7.9) in case of small group sizes. However, figure 7.10(c) shows that selecting the node at the middle of an initial path to be the multicast center does not result in perfectly uniform distribution of the multicast centers. When the group size is large, the advantage of having at least two group members in the initial path is no longer significant. More important in that case is the uniform distribution of the multicast centers, which results in uniform distribution of the network load on all links. That is why RAND-DCSHARED catches up with DCINITIAL-DCSHARED, as the group size increases. In the broadcast case, RAND-DCSHARED performance is even better than DCINITIAL-DCSHARED's performance.

In figure 7.11, we show the average number of group members which successfully join the shared tree per multicast session. MinMaxD-DCSHARED performs much better than the other algorithms with that respect, which confirms the results of the

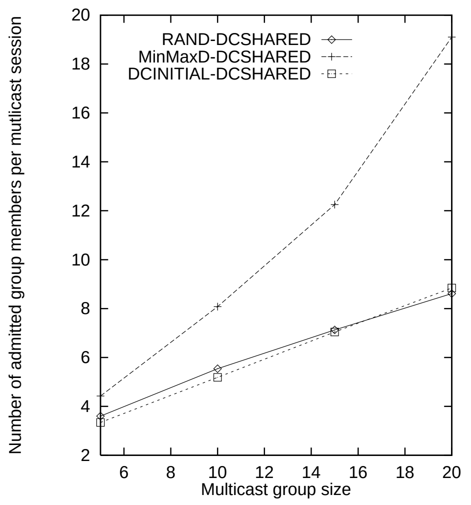


Figure 7.11: Number of admitted group members per multicast sessions, 20 nodes, average degree 4, $\Delta = 0.035$ seconds.

first experiment. Thus, we conclude from the second experiment that MinMaxD-DCSHARED's only weakness lies in its tendency to concentrate the centers of multicast sessions at a few nodes, and hence its inability to distribute the load uniformly over all network links. RAND-DCSHARED and DCINITIAL-DCSHARED distribute the centers more uniformly throughout the network and therefore they can manage the network bandwidth more efficiently.

7.5 Conclusions

In this chapter, we studied the problem of constructing shared multicast trees subject to a delay constraint in connection-oriented networks. We formulated the problem as a diameter-constrained minimum Steiner tree problem (DCMST), and proved that this problem is *NP*-complete. There has been no previous work on this problem. Previous work on shared multicast trees targeted datagram networks and considered only unconstrained cases. We implemented an optimal algorithm for the DCMST problem, DiamOPT. However it has excessive execution times. To avoid this excessive complexity, we proposed three distributed, dynamic heuristics for constructing delay-constrained shared multicast trees.

We divided the problem into two phases. During the first phase, we constructed an initial shared tree and selected one of its nodes to be the multicast center. Then, during the second phase, nodes are permitted to join and leave the shared tree dynamically. We proposed a distributed routing algorithm, DCSHARED, for the second phase. DCSHARED is $O(|V|^3)$ messages in the worst case. Our work focused on the first phase. We studied three distributed heuristics for the first phase of the problem: RAND, MinMaxD, and DCINITIAL. RAND chooses a random node to be the center. It has no message overhead. MinMaxD chooses the node with the least maximum delay to any multicast group member to be the center. Its worst case message complexity is $O(|V|^2)$. DCINITIAL is a heuristic which we proposed to avoid dedicating the first phase of the problem for center selection only. DCINITIAL does both routing and center selection. It constructs an initial shared tree consisting of a delay-constrained path between two group members. Then it selects the node closest to the middle of that path to be the multicast center and advertizes its address to all nodes. DCINITIAL's worst case complexity is $O(|V|^3)$ messages. DCINITIAL makes better use of the first phase of the problem than RAND and MinMaxD, because the initial shared tree it constructs includes at least two group members. On the other hand, the initial trees constructed by RAND and MinMaxD consist of the center only.

A first phase heuristic followed by the route selection algorithm, DCSHARED, constitutes a complete delay-constraint shared tree construction heuristic. We evaluated three such algorithms, RAND-DCSHARED, MinMaxD-DCSHARED, and DC-INITIAL-DCSHARED, in addition to the optimal algorithm DiamOPT. We ran two simulation experiments. DiamOPT had excessive execution times, so it could not be applied in some experiments. It is useful only to benchmark other algorithms. Simulation results showed that MinMaxD-DCSHARED has higher probability of success in constructing delay-constrained shared trees and permitting nodes to join those trees than the other two heuristics. DCINITIAL-DCSHARED constructs shared trees of lower cost than the other two heuristics. RAND-DCSHARED and DCINITIAL-DCSHARED are more efficient in managing the network resources than MinMaxD-DCSHARED. MinMaxD-DCSHARED causes high traffic concentration at a few links, and hence it results in unbalanced network loading.

Overall, we conclude that both MinMaxD-DCSHARED and DCINITIAL-DC-SHARED perform better than RAND-DCSHARED. We suggest to use MinMaxD-DCSHARED, if the primary objective is to maximize the probability that a new member succeeds in joining a shared trees. However, if the primary objective is to construct low-cost shared trees and to manage the network bandwidth efficiently, then we suggest to use DCINITIAL-DCSHARED.

Chapter 8

Multiple Shared Multicast Trees Problem with Delay Constraint

In this chapter, we continue our investigation of multicast routing problems involving multiple sources and multiple receivers in the presence of a delay constraint. We discussed the advantages and disadvantages of shared multicast trees as opposed to source-specific trees in chapter 6. One disadvantage of using a single shared tree is that delays along that tree are larger than the delays if source-specific trees are used instead. David Wall [23] proved that, when the least-delay (LD) tree rooted at the center node of a multicast session is used as a shared multicast tree, the maximum end-to-end delay from any source to any destination along that tree is at most two times the maximum end-to-end if source-specific LD trees are used. The maximum end-to-end delay from any source to any destination along a multicast tree is the diameter of the tree. On the other hand, one advantage of using a single shared multicast tree is that constructing and maintaining a single tree requires less overhead (space and time) than the overhead required to construct and maintain a source-specific tree for each multicast source.

The focus of our work is providing satisfactory performance to real-time applications with delay constraints. In some cases, when the delay constraint value of a multicast session is sufficiently small, a single shared multicast tree may not be able to satisfy this constraint. At the same time, the overhead required to construct and maintain a separate multicast tree for each source node may not be acceptable. Our objective in this chapter is to determine, and construct, the minimum number of shared trees necessary to satisfy the delay constraint of a multicast session.

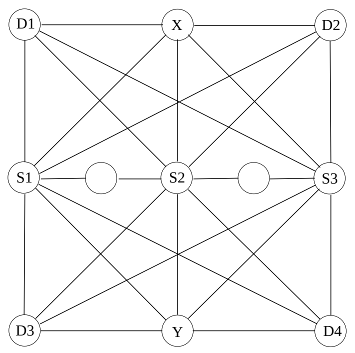
As we mentioned in section 6.4.3 already, the use of multiple shared trees was defined in an expired draft of the sparse mode of the protocol independent multicasting protocol (PIM-SM) [105]. That draft permitted the use of multiple centers, called rendezvous points (RP). Each source sends packets towards each of the RPs, but a destination only joins the tree of a single RP. This results in each RP having its own shared tree that spans all sources but only a subset of the destinations. We follow a similar approach. We select a minimum number of centers and construct a shared multicast tree around each center, such that each tree is used to carry the traffic streams from all sources to a subset of the destinations without violating the delay constraint. Figure 8.1 shows an example of a multicast session for which a single shared tree can not satisfy the imposed delay-constraint while multiple shared trees and source-specific trees can.

The remainder of this chapter is organized as follows. In section 8.1, we formulate the delay-constrained minimum number of centers problem, prove its *NP*-completeness, and propose an optimal solution. Then in section 8.2, we propose heuristic solutions for the same problem. We present simulation results for the proposed algorithms in section 8.3. Finally, we draw our conclusions in section 8.4.

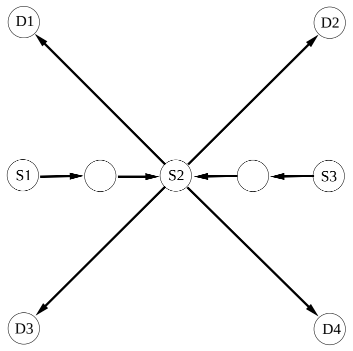
8.1 Problem Formulation and the Optimal Solution

We limit our investigation to symmetric networks only, for the same reasons previously discussed in chapters 6 and 7. A point-to-point communication network is represented as an undirected connected simple network $N = (V, E)$, where $V = \{1, 2, \ldots, |V|\}$ is a set of nodes and E is a set of undirected links. Each link $e = (u, v) \in E$ has a nonnegative real delay $D(u, v)$. $D(u, v) = D(v, u)$ because the network is undirected.

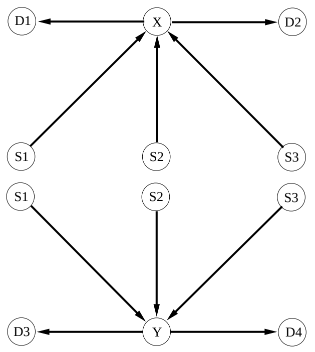
A multicast session consists of a group of multicast destinations $G = \{g_1, \ldots, g_k\}$ and a set of multicast sources $S = \{s_1, \ldots, s_l\}$. Each session imposes a positive real delay constraint Δ. For a given multicast session, determining the minimum number of centers necessary to satisfy the delay constraint and locating their positions requires



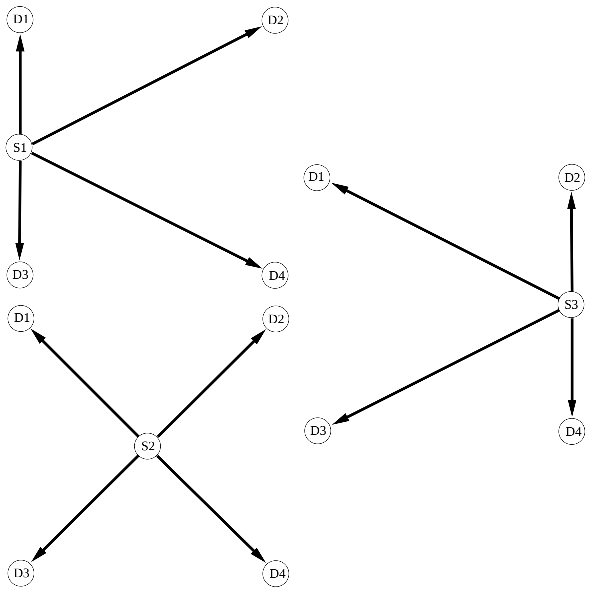
(a) Example network.



(b) A single shared tree. Diameter = 3. Node S_2 is the center.



(c) Two shared trees. Diameters of both trees = 2. Nodes X and Y are the centers.



(d) Three source-specific trees. Diameters of the three trees = 1.

Figure 8.1: Example illustrating the need for multiple shared multicast trees. Unit delays are assigned to all links. The multicast group $G = \{D_1, D_2, D_3, D_4\}$. Multicast sources S_1, S_2, and S_3. Delay constraint $\Delta = 2$. No single shared tree can satisfy this constraint. The best single shared tree is shown in subfigure (b). Two shared tree can satisfy the constraint as shown in subfigure (c). Even better delay performance can be achieved by using source-specific trees as shown in subfigure (d).

only the knowledge of link and path delays. Information about link costs is of no significance to this problem.

The delay-constrained minimum number of centers problem is to select a minimum number of centers, $C_{min} = \{c_1, \ldots, c_m\} \subseteq V$. Each center $c \in C_{min}$ administers its own shared tree $T(c)$. The shared tree $T(c)$ of some multicast center c spans all source nodes S, but only a subset of the destination nodes, $G_c \subseteq G$. G_cs are disjoint sets. The union of the destination nodes spanned by each shared tree is the multicast group, i.e., $\bigcup_{c \in C_{min}} G_c = G$. The constraint is for the diameter of any shared tree to be $\leq \Delta$, i.e., $Diam(T(c)) \leq \Delta$ for all $c \in C_{min}$. $Diam(T(c))$ is the maximum delay along the tree links from any source to any destination. It is given as:

$$Diam(T(c)) = \max_{s_i \in S}(\max_{g_j \in G_c}(\sum_{e \in P_{T(c)}(s_i, g_j)} D(e))) \tag{8.1}$$

where $P_{T(c)}(s_i, g_j)$ is the path from s_i to g_j along the tree $T(c)$. The shared tree constructed around a multicast center is an LD tree rooted at that center. Any shortest path algorithm, e.g., Dijkstra algorithm [12] or Bellman-Ford algorithm [11], can be used to construct that tree.

Theorem 8.1 *The delay-constrained minimum number of centers problem is NP-complete.*

Proof. We prove that a decision version of the problem is *NP*-complete.

The Delay-Constrained Minimum Number of Centers Decision Problem: *Given an undirected network $N = (V, E)$, a nonnegative delay $D(e)$ for each $e \in E$, a multicast group $G = \{g_1, \ldots, g_k\} \subseteq V$, a set of source nodes $S = \{s_1, \ldots, s_l\}$, a positive delay constraint Δ, and a positive integer B, is there a set of nodes C_{min} with each node $c \in C_{min}$ being the center of a shared tree, $T(c)$, that spans all source nodes, S, and a subset of the destinations nodes, $G_c \subseteq G$, such that $|C_{min}| \leq B$ and $\bigcup_{c \in C_{min}} G_c = G$ and $Diam(T(c)) \leq \Delta$ for all $c \in C_{min}$?*

The problem is clearly in *NP*, because a nondeterministic algorithm can guess a set of center nodes, $C_{min} \subseteq V$, and guess a set of nodes, $G_c \subseteq G$, for each $c \in C_{min}$. Then a polynomial time algorithm can be used to construct for each node $c \in C_{min}$ an

LD tree rooted at c, denoted as $T(c)$, such that $T(c)$ spans all nodes in $S \cup G_c$. Finally, it is possible in polynomial time to verify that $|C_{min}| \leq B$ and that $\bigcup_{c \in C_{min}} G_c = G$ and that $Diam(T(c)) \leq \Delta$ for all $c \in C_{min}$.

The next step is to restrict the delay-constrained minimum number of centers problem to a known *NP*-complete problem. When restricting the input value B to $B = 1$, the problem reduces to constructing a single diameter-constrained shared multicast tree that spans all sources and all destinations. Any node on that tree can be selected as the center. Wall studied an even more restricted version of this problem in which all link delays a equal to one and $S = G = V$, i.e., all network nodes are both sources and destinations. He named this problem the Bounded-MaxA Tree problem.

The Bounded-MaxA Tree Problem: *Given an undirected network $N = (V, E)$, a unit delay $D(e) = 1$ for each $e \in E$, and a positive delay constraint Δ, is there a tree, T, spanning all nodes in V such that $Diam(T) \leq \Delta$?*

Wall proved that this problem is *NP*-complete [23]. Therefore, the delay-constrained minimum number of centers problem is also *NP*-complete. □

In the following, we describe an optimal algorithm for solving the delay-constrained minimum number of centers problem. We transform the problem into a set cover problem, and then solve it. For each source-destination pair, s and d, we define the set $PC_{s,d}$ of potential center nodes. A node w is in the set $PC_{s,d}$ if and only if the delay from s to d along the LD tree rooted at w is $\leq \Delta$. Let the LD path between s and w be $\{s \leftrightarrow \ldots \leftrightarrow v \leftrightarrow \ldots \leftrightarrow w\}$, and let the LD path between d and w be $\{d \leftrightarrow \ldots \leftrightarrow v \leftrightarrow \ldots \leftrightarrow w\}$. v may be any node in the network including w, s, and d themselves. Thus the segment $\{v \leftrightarrow \ldots \leftrightarrow w\}$ is common to both paths. And consequently, the delay from s to d along the LD tree rooted at w is given as:

$$Tree\_Delay(s, d, w) = LD\_Path\_Delay(s, v) + LD\_Path\_Delay(v, d) \tag{8.2}$$

where

$$LD\_Path\_Delay(u, v) = \sum_{e \in LD(u,v)} D(e) \tag{8.3}$$

and $LD(u, v) \subseteq E$ is the set of links on the LD path between u and v. Therefore, $PC_{s,d} = \{w | Tree\_Delay(s, d, w) < \Delta\}$. A set PC_d is defined for each destination node

$d \in G$, where $PC_d = \bigcap_{i=1}^{l} PC_{s_i,d}$. PC_d is the set of all possible centers for which the corresponding shared trees may span the destination node d without violating the delay constraint.

Finally, we define the following sets: $G_n = \{d \in V | n \in PC_d\}$, $n = 1, \ldots, |V|$. G_n is the set of all destination nodes that can be reached via the shared tree centered at node n without violating the delay constraint. The transformation is now complete, and the set cover problem is to find a minimal set $C_{min} \subseteq V$ such that $\bigcup_{c \in C_{min}} G_c = G$. We solve the set cover problem using a branch and bound technique to obtain the set C_{min}.

If $PC_{s,d} = \emptyset$, then the LD path between s and d does not satisfy the delay constraint and therefore no delay-constrained shared tree can connect s to d. The corresponding PC_d will also be $\emptyset$. In this case, or any other case that results in $PC_d = \emptyset$, the delay-constrained minimum number of centers problem does not have a solution and the optimal algorithm fails.

The computation of $PC_{s,d}$ for all source-destination pairs dominates the transformation from delay-constrained minimum number of centers problem to set cover problem. It requires calculating the least-delay paths between all nodes. This can be done in $O(|V|^3)$ using Kruskal's all shortest paths algorithm [97] or by applying Dijkstra shortest path algorithm $|V|$ times. After that, the optimal algorithm computes $Tree\_Delay(s, d, w)$ for all sources $s \in |S|$ and all destinations $d \in |G|$ and all $w \in |V|$. Each computation of a $Tree\_Delay(s, d, w)$ takes $O(|V|)$ time in the worst case to find the node v that has to be used in equation 8.2. Thus it takes $O(|S||G||V|^2)$ time in the worst case to compute all $Tree\_Delay(s, d, w)$ values. After that, computing $PC_{s,d}$ for all source-destination pairs, $s \in S$ and $d \in G$, computing P_d for all destinations $d \in G$, and computing G_n for all nodes $n \in V$ is straightforward. Therefore, the worst case complexity of the transformation phase of the optimal algorithm is $O(|V|^3 + |S||G||V|^2)$ time. Of course, the other phase of the optimal algorithm, which involves solving the set cover problem, may have exponentially growing execution times in the worst case. Therefore, we resort to heuristics for the delay-constrained minimum number of centers problem to avoid the excessive complexity of the optimal algorithm. We denote the optimal algorithm presented above **OPT**.

8.2 Heuristic Solutions

In this section, we present four different heuristics for solving the delay-constrained minimum number of centers problem. An LD routing algorithm is used to construct the shared trees corresponding to the selected centers. We take appropriate measures to ensure that the constructed shared trees are delay-constrained.

The first heuristic consists of two phases. The first phase is identical to the first phase of the optimal algorithm, thus transforming the problem into a set cover problem. The second phase is a greedy heuristic for the set cover problem [111]. It starts with all multicast destinations being uncovered and the set of centers being empty. At each step, it adds to the set of centers the node n, whose set G_n covers the most remaining uncovered destinations. Pseudo code of this greedy heuristic is given below.

```
1. input: $G = \{g_1, \ldots, g_k\}$ and a set $G_n \subseteq G$ for all $n \in V$;
2. $U := G$;
3. $C_{min} := \emptyset$;
4. while $U \neq \emptyset$ {
5.     select $n \in V$ that maximizes $|G_n \cap U|$;
6.     $U := U - G_n$;
7.     $C_{min} := C_{min} \cup n$;
8. };
9. output: $C_{min}$;
```

We call the heuristic **GREEDY**. The second phase of the heuristic takes $O(|G||V|)$ time in the worst case. Therefore, GREEDY's overall complexity is dominated by the transformation phase which takes $O(|V|^3 + |S||G||V|^2)$ time. A distributed implementation of this heuristic would be rather complicated because of the complexity of the transformation phase. Most difficult is finding the appropriate node v to be plugged in equation 8.2.

For both OPT and GREEDY, after the set C_{min} is computed, an LD routing algorithm is applied to construct the shared trees. For each $c \in C_{min}$, the LD routing algorithm constructs an LD tree rooted at c and spanning all source nodes in S and the destination nodes in the set G_c. It may happen that a destination node d is covered by two or more of the selected centers in the set C_{min}, e.g., $d \in G_{c_1}$ and $d \in G_{c_2}$ for $c_1 \in C_{min}$ and $c_2 \in C_{min}$. To ensure that this destination node does not receive multiple copies of each packet, only one center includes it in its shared tree. This center is chosen arbitrarily from the members of C_{min} covering the destination node.

The heuristics, we propose in the remainder of this section, solve the delay-constrained minimum number of centers problem directly without transforming it to a set cover problem. We propose a simple heuristic that starts with an empty set of centers. Then the destination nodes attempt, one at a time, to join one of the shared trees of the selected centers. If a destination node fails to join any of the already existing shared trees, without violating the delay constraint, it becomes a new center, and constructs a shared tree for itself. The heuristic works as follows.

```
1.  input: $G = \{g_1, \ldots, g_k\}$ and $\{S = \{s_1, \ldots, s_l\}$;
2.  $C_{min} := \emptyset$;
3.  for each destination node $g \in G$ {
4.      for each center $c \in C_{min}$ {
5.          (comment: destination $g$ attempts to join $T(c)$ via the LD path from $g$ to
            $c$ without causing delay constraint violation)
            if $Diam(T(c) \cup LD(g, c) \leq \Delta$ {
6.              $T(c) := T(c) \cup LD(g, c)$;
7.              exit the inner for loop;
8.          };
9.      };
10.     if destination $g$ failed to join any existing shared tree at selects itself as a
        new center {
11.         $C_{min} := C_{min} \cup g$;
12.         $T(g) := \emptyset$;
13.         for each source node $s \in S$ {
14.             if $LD\_Path\_Delay(s, g) \leq \Delta$ then $T(g) := T(g) \cup LD(s, g)$;
15.         else the algorithm fails and stops;
16.         };
17.     };
18. };
19. output: $C_{min}$;
```

The heuristic's approach to selecting multicast centers is naive, so we call it **NAIVE**. Assuming the LD routing algorithm is implemented using Dijkstra shortest path algorithm, then its worst case complexity is $O(|V|^2)$. NAIVE's complexity can be easily derived from the above pseudo code. It is $O(|G||C_{min}||V|^2 + |G||S||V|^2))$ time. In the worst case, NAIVE's output would be $C_{min} = G$. Therefore, NAIVE's worst case complexity would be $O(|G||V|^2(|G| + |S|))$ time.

As we just mentioned, in the worst case, NAIVE's output would be $C_{min} = G$, which means using $|G|$ shared trees. If $|G| > |S|$, the number of shared trees is greater than the number of trees that would have been constructed if each source uses its own multicast tree. This does not make sense. So we implemented the algorithm such that whenever $|C_{min}|$ exceeds $|S|$, we stop executing the algorithm and opt for source-specific trees instead of shared trees.

We propose two more heuristics for selecting a minimum number of multicast centers. Both heuristics follow very similar steps to NAIVE. Unlike NAIVE, however, both heuristics use more sophisticated center selection mechanisms. They keep an ordered list of candidate centers. If a destination node fails to join any of the already existing shared trees without violating the delay constraint, it selects a new center. The new center is the first entry in the list of candidate centers which is not already a center. The list of candidate centers contains all network nodes in ascending order of a certain criterion. One heuristic uses, as a criterion, the maximum delay from a node to any multicast source or multicast destination. This heuristic is called **MAXD**. The other heuristic is called **AVGD**. It uses, as a criterion, the average delay from a node to all multicast sources and destinations. Both MAXD and AVGD start by constructing the ordered list of nodes. This is dominated by computing the least delays from all nodes in the network to all multicast sources and destinations. It can be done in $O(|V|^2(|S \cup G|))$ time by applying Dijkstra shortest path algorithm $|S \cup G|$ times. Constructing the ordered lists, either according to maximum delay or average delay, from the computed least delay values is straightforward. After constructing the ordered lists, both MAXD and AVGD proceed similar to NAIVE except that node g

in steps 11 through 15 of NAIVE's pseudo code is replaced with the first node in the ordered list which is not already a center. We see no need to provide separate pseudo code for MAXD and AVGD. The worst case complexity of both MAXD and AVGD is $O(|G||V|^2(|G| + |S|))$ time, the same as NAIVE.

Computing the criterion values for the ordered lists of MAXD and AVGD can be done in a distributed fashion. Each node computes its criterion value then transmits it to a central node which orders the nodes into a list and advertizes the contents of this list to all nodes in the network. The rest of the MAXD and AVGD heuristics, as well as the entire NAIVE heuristic, can also be implemented distributedly. A node's (either a source node or a destination node) attempt to join an existing shared tree can be executed using a distributed LD routing algorithm. If a destination node fails to join any of the existing shared trees, it selects a new center locally, then broadcasts its selection to all network nodes.

In their work on center selection protocols, Thaler and Ravishankar [109] proposed to use ordered lists of candidate centers, similar to MAXD and AVGD, and they allowed the use of multiple centers for the same multicast session. However, in their experimental evaluation of the center selection problem, the authors considered only a single center per multicast session. In the next section, we use simulation to evaluate the multiple centers selection algorithms presented in the previous part of this chapter.

8.3 Simulation Results

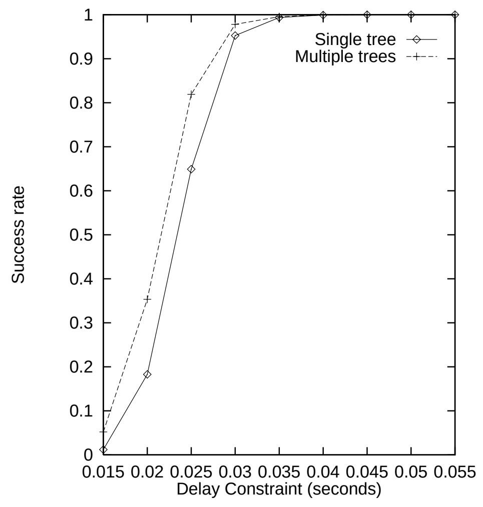
Our objective in this section is to evaluate the performance of the different center selection algorithms with respect to their ability to minimize the number of centers, and hence the number of shared trees, required to satisfy the delay constraint of a real-time multicast session. We ran a simulation experiment of a videoconferencing multicast session in which all participants are both sources and receivers, i.e., $S = G$. Full duplex, simple, connected, random networks similar to the random networks described in section 3.1 were used in the experiment. We simulated 20-node and 50-node networks with an average node degree of 4. The node positions were fixed for each network size and the links interconnecting these nodes were generated using our random link generator described in section 3.1.1. The networks spanned an area of

$4000 * 2400$ Km^2 ($0.02*0.012$ seconds^2 in terms of propagation delay), and 155 Mbps links were used. Under these assumptions, the propagation components dominated the link delays, and hence the link delays were symmetric, i.e., $D(u, v) = D(v, u)$, and the resulting network was undirected.

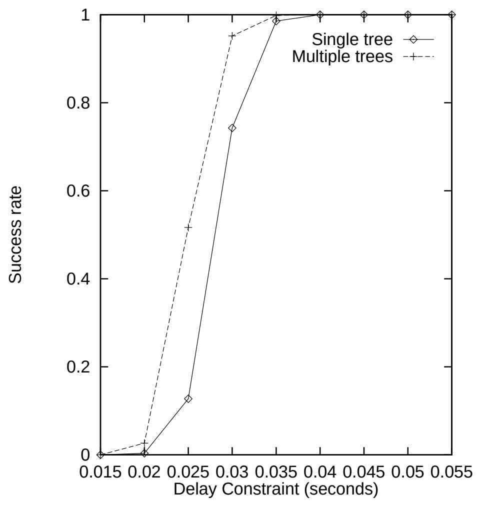
For each run of the experiment, we generated a random set of links to interconnect the fixed nodes and we selected a random multicast group with each group member being both a source and a destination. The experiment was repeated with different multicast group sizes and different delay constraint values. For each algorithm, we measured its success rate in finding a solution and the number of multicast centers, same as the number of shared trees, in the proposed solution. The experiment was run repeatedly until confidence intervals of less than 5%, using 95% confidence level, were achieved for all measured quantities. At least 500 networks were simulated for each measurement.

Before presenting the results of our experiment for the algorithms studied in this chapter, we present results which justify our work on the multiple shared multicast trees problem. Our primary motivation for investigating this problem was that multiple shared trees may be able to satisfy the delay constraint of a multicast application when a single shared tree fails. We ran our experiment using the optimal algorithm for constructing a single delay-constrained shared tree algorithm (a simple version of DiamOPT of chapter 7 which attempts find a delay-constrained solution without trying to minimize the tree cost) and the optimal algorithm for constructing a minimum number of delay-constrained shared trees, OPT. Figure 8.2 shows the success rates of both algorithms in finding a solution that satisfies the imposed delay constraint for 20-node networks. The need for multiple shared trees arises when the delay constraint is strict. For a fixed strict delay constraint value, e.g., 0.03 seconds, the difference between the success rates of the two algorithms increases as the group size increases. For a group size of 20 nodes and a delay constraint of 0.03 seconds, the difference in success rates exceeds 60%. In general, the improvement which multiple shared trees achieve over single shared trees, in case of stringent delay constraint values, is significant enough to justify our work on the multiple delay-constrained shared multicast trees problem.

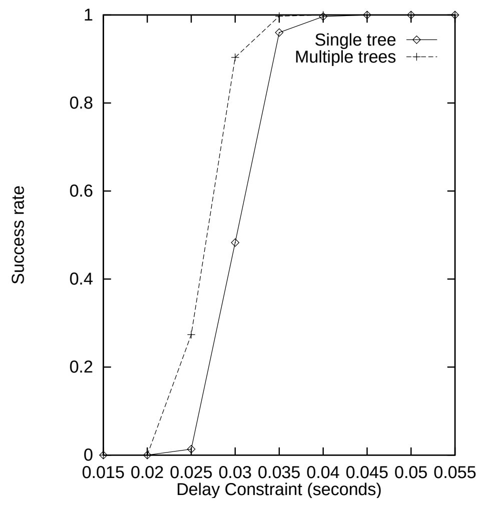
All multiple center selection algorithms studied in this chapter are capable of



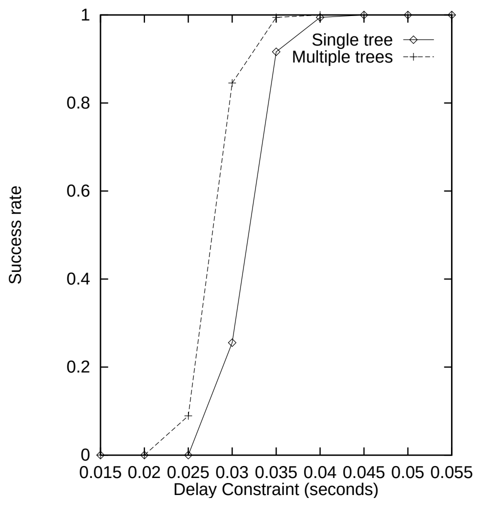
(a) 5 group members



(b) 10 group members

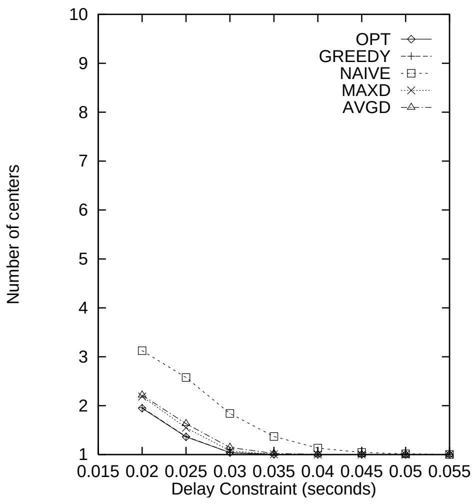


(c) 15 group members

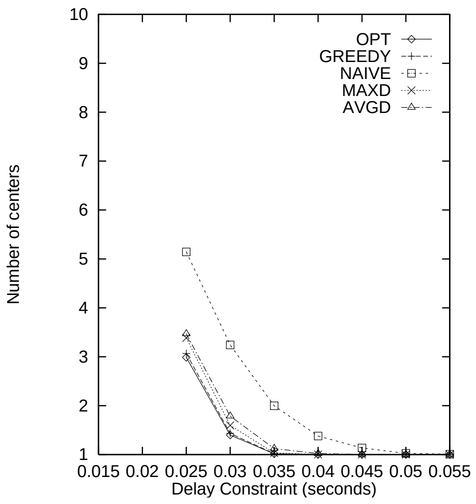


(d) 20 group members

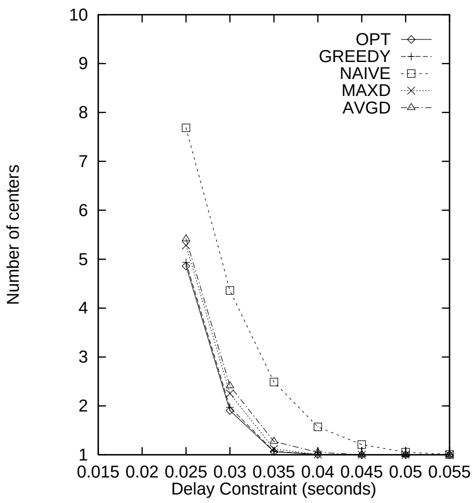
Figure 8.2: Success rates of the optimal single delay-constrained shared multicast tree algorithm and the optimal multiple delay-constrained multicast trees algorithm, 20-node networks, average degree 4, variable delay constraint and different multicast group sizes.



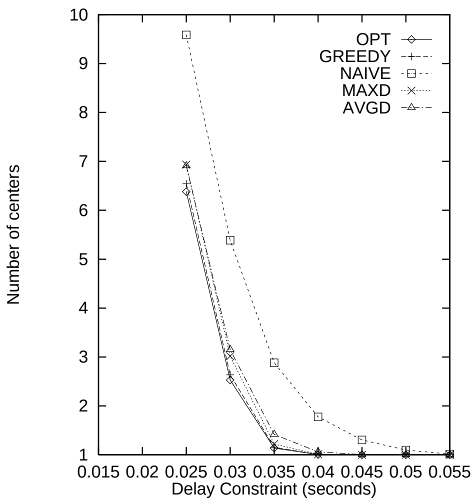
(a) 5 group members



(b) 10 group members



(c) 15 group members



(d) 20 group members

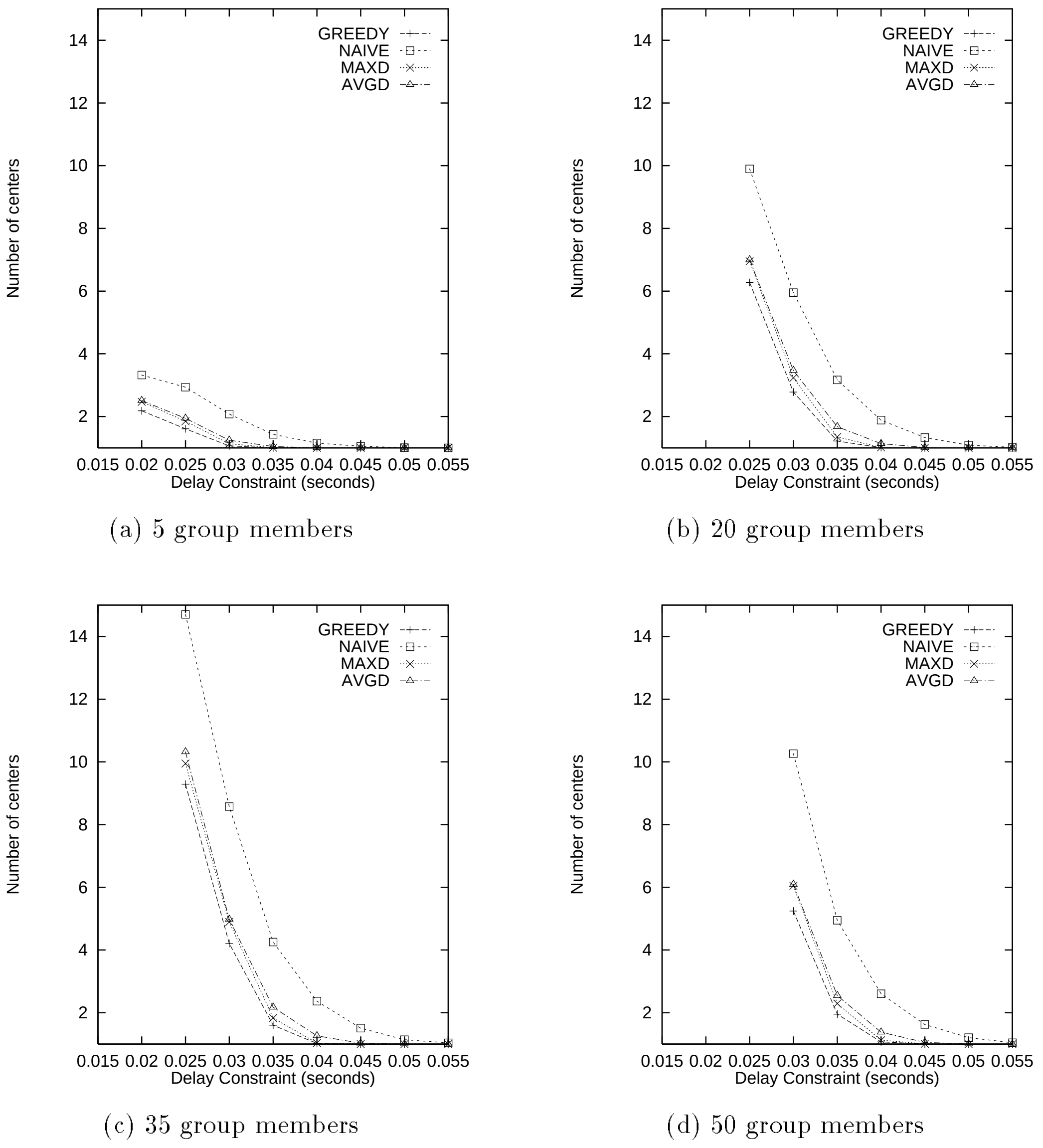
Figure 8.3: Average number of centers for various multiple delay-constrained shared multicast trees construction algorithms, 20-node networks, average degree 4, variable delay constraint and different multicast group sizes.

achieving the same success rate as the optimal algorithm, OPT. The difference between the algorithms lies in the number of centers each algorithm uses to satisfy the delay constraint. Figure 8.3 shows the average number of center each algorithm uses versus the delay constraint for different multicast group sizes for 20-node networks. We do not show the number of centers for very strict delay constraint values, because the algorithms had very small success rates, and hence we could not simulate enough successful runs to achieve 5% confidence intervals. Of course, as the delay constraint is relaxed, the number of centers required to satisfy the delay constraint decreases. For large enough delay constraint values, the number of centers approaches one. For a fixed delay constraint value, the number of centers increases as the size of the multicast group increases. From figure 8.3 it is obvious that GREEDY is a very good approximation algorithm. It is on the average as good as OPT in all cases shown. MAXD and AVGD yield very similar performance. They are also close to optimal, though not as close as GREEDY. NAIVE is the only heuristic that yields a poor performance. In NAIVE, whenever a destination fails to join any of the already existing shared trees, this destination is selected as the center of a new shared tree. In many cases this destination is remotely located, and its shared tree can not serve any other destinations besides itself. That is the cause of NAIVE's poor performance.

We repeated the experiment using 50-node networks. The results are shown in figure 8.4. However, we could not use OPT in that case due to its excessive running times. The performance of the heuristic relative to each other in case of 50-node networks is identical to the case of 20-node networks. We therefore conclude that NAIVE's center selection mechanism, which is almost arbitrary, results in poor performance with respect to minimizing the number of multicast centers. The other three heuristics are capable of selecting to find a number of multicast centers that is very close to optimal. GREEDY is slightly closer to OPT than MAXD and AVGD.

8.4 Conclusions

In this chapter, we continued our investigation of problems related to shared multicast trees. Our focus was on finding the minimum number of shared trees necessary to satisfy the delay constraint imposed by a multicast application. By minimizing the



(a) 5 group members

(b) 20 group members

(c) 35 group members

(d) 50 group members

Figure 8.4: Average number of centers for various multiple delay-constrained shared multicast trees construction algorithms, 50-node networks, average degree 4, variable delay constraint and different multicast group sizes.

number of multicast trees, we reduce the overhead required to construct and maintain these trees. Simulation results indicate that multiple shared trees can achieve a much higher success rate in satisfying the applications' delay requirements than a single shared tree.

We proved that the problem of finding the minimum number of shared trees necessary to satisfy the delay constraint of a multicast session is *NP*-complete, and we proposed an optimal algorithm, OPT, for solving this problem. OPT consists of two phases. In the first phase, it transforms the original problem into a set cover problem. Then, in the second phase, it solves the set cover problem using a branch and bound technique. To avoid the excessive complexity of the optimal algorithm, we proposed four heuristics: GREEDY, NAIVE, MAXD, and AVGD. All four heuristics have similar worst case time complexities. We outlined a possible distributed implementation for NAIVE, MAXD, and AVGD. These three heuristics may also allow sources and destinations to join and leave the shared multicast trees dynamically. If distributed implementation of GREEDY is possible, it will certainly be more complex than the proposed distributed implementation of the three other heuristics. Simulation results indicate that GREEDY, MAXD, and AVGD yield close to optimal performance with respect to minimizing the number of multicast centers with GREEDY being the best. NAIVE's performance is poor as compared to the performance of the other heuristics.

Chapter 9

Conclusions and Future Work

In this dissertation, we studied a number of routing problems for real-time communication on high-speed, connection-oriented, wide-area networks. Real-time applications, e.g., multimedia and real-time control, have QoS service requirements which must be guaranteed by the underlying network. One QoS requirement, the end-to-end delay constraint, can be guaranteed in wide-area networks by using the appropriate routing algorithms. In addition to the QoS requirements, many real-time applications have high bandwidth requirements, and hence it is important to use routing algorithms which manage the network bandwidth efficiently. In this dissertation, we defined two functions for each link in a network: a link cost which is a function of the utilized fraction of the link's capacity, and a link delay which is a function of the delay a packet experiences when it traverses that link. The objective of all routing problems we studied (except the problem of chapter 8) was to minimize the tree cost, or path cost, without violating the real-time application's delay constraint. Many real-time applications, e.g., videoconferencing, involve multiple sources and multiple destinations. Therefore, our focus was primarily on multicast routing problems.

In chapter 2, we surveyed previous work on multicast routing. Two types of multicast trees can be constructed to support a multicast session: source-specific trees or shared trees. In chapter 3, we evaluated the performance of different multicast routing algorithms when applied to construct source-specific multicast trees. In chapters 4 and 5, we studied two special cases of the source-specific multicast routing problem, the broadcast routing problem and the unicast routing problem, respectively. Starting from chapter 6, we turned our attention to shared multicast trees and the related center selection problems. We discussed the advantages and disadvantages of shared multicast trees as compared to source-specific trees. Then we presented a survey of previous work on center selection. In chapter 7, we studied the problem of constructing a single shared multicast tree to carry all traffic streams belonging to the same multicast session. Finally, in chapter 8, we studied the problem of constructing multiple shared multicast trees for the same multicast session.

All problems studied in this dissertation are *NP*-complete. The optimal solutions for these problems are therefore too complex, and can only be used to benchmark heuristic solutions. For each problem we studied in our work, we evaluated existing heuristic solutions and/or proposed new heuristics to solve that problem. We evaluated the performance of these heuristics using simulation of flat randomly generated wide-area networks. Due to the rapid growth of wide-area networks, researchers are currently developing mechanisms to organize these networks into hierarchies of subnetworks. We therefore recommend that future work should use hierarchical network models.

9.1 The Multicast Routing Problem

In chapter 2, we classified multicast routing algorithms into different categories, and presented important criteria to be considered when summarizing the features of a multicast routing algorithm. Then we presented an extensive survey of previous work on multicast routing algorithms. Our survey revealed that over the years, a lot of algorithms have been proposed for various multicast routing problems. During the past few years, motivated by the rapid evolution of real-time multicast applications, researchers proposed a number of delay-constrained multicast routing algorithms. Unfortunately, however, different researchers have made different assumptions when evaluating the performance of the algorithms they proposed. In addition, no evaluation studies, dedicated to comparing the performance of the different delay-constrained algorithms, have been reported in the literature.

The emphasis of our survey was on multicast routing algorithms, but we reviewed previous work on multicast routing protocols as well. Developing a multicast routing protocol is a very complex procedure. Many aspects must be taken into account, when designing a multicast routing protocol which are of no significance to researchers working on multicast routing algorithms. Recovery from link and node failures and robustness to erroneous state information are just two examples of such aspects. We therefore urge researchers to consider simplicity and ease of implementation during future work on multicast routing algorithms. These aspects may be even more important than the efficiency of the algorithm. Efficient, close to optimal, but complex multicast routing algorithms will not be adopted by protocol developers, because such algorithms would add to the difficulty of the already very difficult task of designing multicast routing protocols.

9.2 Evaluation of Source-Specific Multicast Routing Algorithms

In chapter 3, we evaluated most of the new delay-constrained multicast routing algorithms and a few selected unconstrained algorithms and presented a quantitative comparison of all these algorithms when applied in realistic high-speed networking environments. Our objective was to evaluate each algorithm's capability of providing guaranteed delay-constrained service for real-time applications and also to evaluate each algorithm's ability to manage the network bandwidth efficiently. In this evaluation study, we considered only source-specific multicast trees.

For our evaluation, we used simulation of multicast sessions on randomly generated, asymmetrically loaded networks of different sizes. We implemented a random network generator that is a modified version of Waxman's random network generator [42]. Our generator is fast and constructs two-connected networks biased towards shorter links, thus resembling real networks.

Our simulation results showed clearly that most unconstrained cost-oriented multicast routing algorithms are capable of constructing low-cost multicast trees and managing the network bandwidth efficiently. Unfortunately, these unconstrained algorithms can not satisfy the delay requirements of real-time applications in wide-area networks.

Several multicast routing protocols use reverse shortest path multicast trees. We studied one algorithm for constructing reverse shortest path trees, namely reverse path multicasting (RPM). We showed that RPM creates high-cost multicast trees when the link loads are asymmetric and that it is very inefficient in utilizing the network bandwidth, because it results in very unbalanced network loads. Current reverse shortest path-based multicast routing protocols do not perform admission control or resource reservation functions during route selection. There are separate protocols for resource reservation, and these protocols have minimal interaction with routing. We showed, using simulation, that incorporating routing, admission control, and resource reservation together in a single module can dramatically improve the efficiency of reverse shortest path-based multicast routing protocols in managing the available network bandwidth.

We concluded that the unconstrained multicast routing algorithms can not be applied to real-time applications on wide-area networks. Then we evaluated the performance of the delay-constrained multicast routing algorithms. The delay-constrained algorithms can be divided into two groups. Algorithms belonging to the first group are capable of constructing low-cost, close to optimal, delay-constrained multicast trees, and they can manage the network bandwidth efficiently. Unfortunately, however, these algorithms have large execution times that grow at fast rates with the size of the network. Our overall conclusion for this group of algorithms was that, in spite of their efficiency, they are too complex to be applied to any real wide-area networks.

Algorithms belonging to the second group of delay-constrained multicast routing algorithms do not perform as good as the ones belonging to the first group. However, they have much faster average execution times and lower worst case time complexities. Therefore, they scale better to large networks.

Neither the algorithms of the first group nor those of the second group can be adopted by a real multicast routing protocol, because several implementation issues, such as distributed implementation and dynamic join/leave of multicast group members, have not be addressed yet for any of them. We suggest that any future work on delay-constrained multicast routing should focus on simple algorithms and consider efficient, distributed, scalable implementations of such algorithms.

The experience we gained from our survey of previous work on multicast routing (chapter 2) and our evaluation of existing source-specific multicast routing algorithms (chapter 3), enabled us to identify some problems which are significant to real-time applications and which have not been addressed by researchers in the past. In addition, we found out that several research groups are currently working on delay-constrained multicast routing algorithms for constructing source-specific trees. So we decided to focus in our work on some of the problems which have not been addressed before. We studied two special cases of the delay-constrained source-specific multicast routing problem: the delay-constrained broadcast routing problem and the delay-constrained unicast routing problem. Finally, we worked on routing problems for constructing delay-constrained shared multicast trees.

9.3 Delay-Constrained Broadcast Routing Problem

In chapter 4, we studied the problem of constructing broadcast trees for real-time traffic with delay constraints in asymmetric networks. We formulated the problem as a delay-constrained minimum spanning tree problem, and then we proved that this problem is *NP*-complete by reducing a known *NP*-complete problem, the exact cover with three-sets problem, to it. To avoid the excessive complexity of the optimal solution, we proposed the bounded delay broadcast heuristic (BDB) to solve the delay-constrained minimum spanning tree problem. The heuristic consists of two phases. The first phase is based on Prim's algorithm and constructs a moderate-cost delay-constrained spanning tree. The second phase reduces the cost of that tree by replacing tree links with lower cost links not in the tree, without violating the imposed delay constraint. BDB is the first heuristic designed specifically for the delay-constrained minimum spanning tree problem. Since there are no other heuristics that address the same problem as BDB, we had to compare BDB to heuristics for solving the more general problem of delay-constrained multicast routing. Simulation results show that BDB's performance is close to optimal with respect to tree cost, both in case of asymmetric networks as well as in case of symmetric networks. Its cost performance is as good as that of the best delay-constrained multicast routing heuristic. In addition, its average execution times grow at the same rate as the execution times of the fastest delay-constrained multicast routing heuristic.

Unfortunately, however, we could not perform a conclusive analysis of the worst case time complexity of BDB, nor could we manage to prove the correctness of the heuristic. However, our extensive simulations indicate that it is always capable of constructing a loop-free delay-constrained broadcast tree within a finite time, if one exists. Deriving BDB's complexity, proving its correctness, and implementing it in a distributed fashion are open issues left for future research.

9.4 Delay-Constrained Unicast Routing Problem

We studied the delay-constrained unicast routing problem in chapter 5. We formulated the problem as a delay-constrained least-cost path problem, which is known to be *NP*-complete. Therefore, we proposed a distributed, source-initiated heuristic solution, the delay-constrained unicast routing algorithm (DCUR), to avoid the excessive complexity of the optimal solutions. DCUR requires only a limited amount of information at each node. The information at each node is stored in a cost vector and a delay vector. These vectors are constructed and maintained in exactly the same manner as the distance vectors which are widely deployed over current datagram networks. However, DCUR constructs paths in connection-oriented networks. DCUR is capable of detecting and eliminating any loops that may occur while it constructs a delay-constrained path. We proved the correctness of DCUR by showing that it is always capable of constructing a loop-free delay-constrained path within finite time, if such a path exists. The number of computations at each node participating in the path construction process is fixed, irrespective of the network size. The worst case message complexity of DCUR is dominated by the occurrence and removal of loops. It requires $O(|V|^3)$ messages to construct a single path in the worst case, where $|V|$ is the number of nodes. Fortunately, however, our simulation results show that DCUR requires much fewer messages on the average, because loop occurrence is rare. We compared the performance of DCUR to that of the optimal delay-constrained least-cost algorithm and the least-delay algorithms. These are the only three algorithms capable of satisfying the delay constraints imposed by real-time applications in wide-area networks. Our simulation results indicated that DCUR yields satisfactory performance with respect to both path cost and path delay. The average costs of DCUR-constructed paths are always within 10% from optimal.

The efficient performance and distributed nature of DCUR encourage us to use it as a starting point for implementing an integrated protocol that is capable of providing QoS guarantees for real-time applications. In the following, we present some of the issues that should be addressed by future work on DCUR.

We assumed that the state of the network does not change during the computation of DCUR, and that the contents of all vectors and routing table entries are correct and up-to-date. This is a limiting, unrealistic assumption, because networks frequently enter transient states in response to topology changes, e.g., link failures or link cost or link delay changes. Information about such failures and changes can not be propagated to all network nodes instantaneously. Mechanisms must be specified to enable DCUR to cope with situations when the contents of the cost vectors and the delay vectors at different nodes are not consistent.

Distance-vector-based protocols discourage the use of dynamic link cost metrics, e.g., utilization-based metrics, because such metrics change frequently, and, as a result, the network may remain in transient state for long periods of time. However, utilization-based cost metrics are essential for efficient management of the network resources. Therefore, mechanisms are needed that react immediately to a link cost change and propagate the effects of that change throughout the network fast enough, in order to limit the periods during which the network is in a transient stage.

Admission control checks and resource reservation functions should be incorporated into DCUR, such that resources are immediately reserved along the links of the constructed path. Separating routing from admission control and resource reservation may affect DCUR's effectiveness in managing the network resources, e.g., the network bandwidth.

In chapter 7, we used variations of DCUR to construct shared multicast trees. However, work remains to be done to extend DCUR to address routing of source-specific multicast trees. Work remains to be done also to specify the operation of DCUR in local-area multi-access networks in addition to the current specification that covers wide-area point-to-point networks only.

9.5 Shared Multicast Trees

Starting from chapter 6, we switched our attention to the construction of shared multicast trees. Shared multicast trees are an alternative for source-specific multicast trees in case of multicast sessions involving multiple sources and multiple destinations. Each approach has it advantages and disadvantages. Less overhead, both time and memory space, is required to construct and maintain a single, or only a few, shared tree per multicast session than the overhead required to construct and maintain a separate tree for each source. On the other hand end-to-end delays along shared trees are larger than those along source-specific trees, and the traffic concentration on shared tree links is higher.

One problem associated with the construction of shared multicast trees is the center selection problem. Most previous research proposes to start by selecting a node to be the center of a the multicast session, and to construct a shared tree around that node after that. Route selection algorithms used to construct a shared tree around the selected center are similar to those which we have already studied for source-specific routing. We therefore focused mainly on the center selection part of the shared multicast tree construction problems.

We reviewed previous work on shared multicast trees and the center selection problem. Our review covered the fields of graph theory and operations research in addition to communication networks. None of the previous work addressed the problem of constructing shared multicast trees for real-time applications with delay constraints. In addition, inadequate work has been reported to date on the multiple centers selection problem. Some of the advantages of using multiple centers with each center managing a separate shared tree are: more fault tolerance, smaller end-to-end delays than those along a single shared tree, and multiple shared trees may be used to provide multiple QoS levels.

After discussing the advantages and disadvantages of shared multicast trees, and surveying previous work, we went on to study the problems of constructing a single shared multicast tree and multiple shared multicast trees in the presence of an application-imposed delay-constraint.

9.6 Single Shared Multicast Tree Problem with Delay Constraint

In chapter 7, we studied the problem of constructing a single shared multicast tree subject to a delay constraint. We formulated the problem as a diameter-constrained minimum Steiner tree problem and proved that this problem is *NP*-complete. Previous work on shared multicast trees considered only unconstrained cases and proposed heuristics that consisted of a center selection phase and a route selection phase. We followed the same steps in our investigation of the delay-constrained shared multicast tree problem. Our work focused on the center selection phase. Therefore, we used a fixed delay-constrained routing algorithm. The routing algorithm we used is a variation of the DCUR heuristic of chapter 5 that allows nodes to join and leave a delay-constrained shared tree dynamically. We presented three distributed heuristics for the center selection phase and derived the worst case message complexity of each of them.

We ran simulation experiments to evaluate the performance of the different shared tree construction heuristics and to compare them to the optimal solution. We evaluated the algorithms based on their success rate in constructing a delay-constrained shared tree, the cost of the constructed tree, and their ability to balance the network load. Each of the heuristics we evaluated had its strengths and weaknesses. There was no clear winner among them.

The center selection heuristics we proposed are simple and distributed and can be easily implemented in real networks. The same directions for future research presented in section 9.4 for DCUR apply for the routing algorithm we used to construct shared multicast trees.

9.7 Multiple Shared Multicast Trees Problem with Delay Constraint

Simulation results indicated that multiple shared multicast trees can achieve a much higher success rate in satisfying the applications' delay constraint than a single shared multicast tree. In chapter 8, our focus was on finding the minimum number of shared trees necessary to satisfy the delay constraint imposed by a multicast application. By minimizing the number of multicast trees, we reduce the overhead required to construct and maintain these trees.

We proved that the problem of finding the minimum number of shared trees necessary to satisfy the delay constraint of a multicast session is *NP*-complete, and proposed an optimal algorithm for solving this problem. We also proposed four heuristic solutions to avoid the excessive complexity of the optimal algorithm. We proposed a distributed implementation scheme for three of the proposed heuristics. Simulation results showed that the only centralized heuristic yields close to optimal performance with respect to minimizing the number of multicast centers. Two of the distributed heuristics also yield close to optimal performance, though not as close as the centralized heuristic. The performance of the third distributed heuristic is very poor as compared to the other algorithms.

We suggest that future work on multiple shared multicast trees should take more aspects into account than just minimizing the number of multicast centers. For example, future research may consider how to distribute the multicast centers in the network in order to maximize the fault tolerance.

Having multiple shared trees for the same multicast sessions results in trees that may overlap and cross each other. Complicated mechanisms are needed to forward packets correctly over these trees. This was an important reason for the protocol independent multicasting protocol (PIM) [74] developers to abandon the use of multiple shared trees. Therefore, some work is needed on mechanisms for managing the routing table entries corresponding to multiple shared trees and for forwarding the packets correctly over those trees.

9.8 Summary of Main Contributions

Our contributions in this dissertation are the following.

- We formulated several delay-constrained routing problems, and we proved the *NP*-completeness of these problems. The proof of *NP*-completeness, we presented for the delay-constrained minimum spanning tree problem, is of particular value and gives guidelines to be followed when analyzing the complexities of other related problems.

- We proposed several heuristic solutions for the delay-constrained routing problems we studied. If we had to single out one algorithm to be the best, we would choose the delay-constrained unicast routing heuristic. It is a simple, distributed, scalable algorithm based on a limited amount of network state information. Some work remains to be done to improve the algorithm's robustness, but, overall, it makes a good candidate for a delay-constrained unicast routing protocol in connection-oriented networks.
- We used simulation to evaluate the algorithms we proposed as well as the algorithms proposed by many other researchers. In our simulation experiments, we imitated real high-speed networking environments. In the case of delay-constrained source-specific multicast routing algorithms, our evaluation was the first quantitative comparison of previously proposed algorithms under unified realistic assumptions.

List of References

[1] M. de Prycker, *Asynchronous Tranfer Mode, Solution for Broadband ISDN.* Prentice Hall, 3rd ed., 1995.

[2] M. Macedonia and D. Brutzman, "MBone Provides Audio and Video Across the Internet," *IEEE Computer*, vol. 27, no. 4, pp. 30–36, April 1994.

[3] D. Cheriton and S. Deering, "Host Group: a Multicast Extension for Datagram Internetworks," in *Proceedings of the Ninth ACM/IEEE Data Communications Symposium*, pp. 172–179, 1985.

[4] S. Deering, *Multicast Routing in a Datagram Internetwork.* PhD thesis, Stanford University, December 1991.

[5] D. Bertsekas and R. Gallager, *Data Networks.* Prentice-Hall, 2nd ed., 1992.

[6] S. Casner and S. Deering, "First IETF Internet Audiocast," *in Proccedings of ACM SIGCOMM, Computer Communications Review*, vol. 22, no. 3, July 1992.

[7] R. Karp, "Reducibility among Combinatorial Problems," in *Complexity of Computer Computations* (R. Miller and J. Thatcher, eds.), pp. 85–103, Plenum Press, 1972.

[8] R. Prim, "Shortest Connection Networks and Some Generalizations," *The Bell Systems Technical Journal*, vol. 36, no. 6, pp. 1389–1401, November 1957.

[9] K. Claffy, G. Polyzos, and H. Braun, "Traffic Characteristics of the T1 NSFNET Backbone," in *Proceedings of IEEE INFOCOM '93*, pp. 885–893, 1993.

[10] V. Paxson, "Growth Trends in Wide-Area TCP Connections," *IEEE Network*, August 1994.

[11] R. Bellman, *Dynamic Programming.* Princeton University Press, 1957.

[12] E. Dijkstra, "Two Problems in Connection with Graphs," *Numerische Mathematik*, vol. 1, no. 5, pp. 269–271, October 1959.

[13] B. Awerbuch, A. Bar-Noy, and M. Gopal, "Approximate Distributed Bellman-Ford Algorithms," in *Proceedings of IEEE INFOCOM '91*, pp. 1206–1213, 1991.

[14] D. Bertsekas, *Linear Network Optimization: Algorithms and Codes.* MIT Press, 1991.

[15] Y. Dalal and R. Metcalfe, "Reverse Path Forwarding of Broadcast Packets," *Communications of the ACM*, vol. 21, no. 12, pp. 1040–1048, December 1978.

[16] S. Deering and D. Cheriton, "Multicast Routing in Datagram Internetworks and Extended LANs," *ACM Transactions on Computer Systems*, vol. 8, no. 2, pp. 85–110, May 1990.

[17] F. Hwang and D. Richards, "Steiner Tree Problems," *Networks*, vol. 22, no. 1, pp. 55–89, January 1992.

[18] P. Winter, "Steiner Problem in Networks: A Survey," *Networks*, vol. 17, no. 2, pp. 129–167, Summer 1987.

[19] L. Kou, G. Markowsky, and L. Berman, "A Fast Algorithm for Steiner Trees," *Acta Informatica*, vol. 15, no. 2, pp. 141–145, 1981.

[20] H. Takahashi and A. Matsuyama, "An Approximate Solution for the Steiner Problem in Graphs," *Mathematica Japonica*, vol. 24, no. 6, pp. 573–577, February 1980.

[21] V. Rayward-Smith, "The Computation of Nearly Minimal Steiner Trees in Graphs," *International Journal of Mathematical Education in Science and Technology*, vol. 14, no. 1, pp. 15–23, January/February 1983.

[22] D. Wall, "Selective Broadcast in Packet-Switched Networks," in *Proceedings of the Sixth Berkeley Workshop on Distributed Data Management and Computer Networks*, pp. 239–258, February 1982.

[23] D. Wall, *Mechanisms for Broadcast and Selective Broadcast.* PhD thesis, Stanford University, June 1980.

[24] M. Doar and I. Leslie, "How Bad is Naive Multicast Routing," in *Proceedings of IEEE INFOCOM '93*, pp. 82–89, 1993.

[25] M. Doar, *Multicast in the Asynchronous Transfer Mode Environment.* PhD thesis, University of Cambridge, January 1993.

[26] V. Rayward-Smith and A. Clare, "On Finding Steiner Vertices," *Networks*, vol. 16, pp. 283–294, 1986.

[27] X. Jiang, "Routing Broadband Multicast Streams," *Computer Communications*, vol. 15, no. 1, pp. 45–51, January,February 1992.

[28] X. Jiang, "Distributed Path Finding Algorithm for Stream Multicast," *Computer Communications*, vol. 16, no. 12, pp. 767–775, December 1993.

[29] S. Ramanathan, "An Algorithm for Multicast Tree Generation in Networks with Asymmetric Links," in *Proceedings of IEEE INFOCOM '96*, pp. 337–344, 1996.

[30] C.-H. Chow, "On Multicast Path Finding Algorithms," in *Proceedings of IEEE INFOCOM '91*, pp. 1274–1283, 1991.

[31] Y.-W. Leung and T.-S. Yum, "Efficient Algorithms for Multiple Destinations Routing," in *Proceedings of the IEEE International Conference on Communications (ICC '91)*, pp. 1311–1317, 1991.

[32] F. Bauer and A. Varma, "Distributed Algorithms for Multicast Path Setup in Data Networks," *IEEE/ACM Transactions on Networking*, vol. 4, no. 2, pp. 181–191, April 1996.

[33] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* New York: W.H. Freeman and Co., 1979.

[34] R. Widyono, "The Design and Evaluation of Routing Algorithms for Real-Time Channels," Tech. Rep. ICSI TR-94-024, University of California at Berkeley, International Computer Science Institute, June 1994.

[35] Q. Sun and H. Langendoerfer, "Efficient Multicast Routing for Delay-Sensitive Applications," in *Proceedings of the Second Workshop on Protocols for Multimedia Systems (PROMS '95)*, pp. 452–458, October 1995.

[36] S. Wi and Y. Choi, "A Delay-Constrained Distributed Multicast Routing Algorithms," in *Proceedings of the Twelveth International Conference on Computer Communication (ICCC '95)*, pp. 833–838, 1995.

[37] V. Kompella, J. Pasquale, and G. Polyzos, "Multicast Routing for Multimedia Communication," *IEEE/ACM Transactions on Networking*, vol. 1, no. 3, pp. 286–292, June 1993.

[38] V. Kompella, J. Pasquale, , and G. Polyzos, "Multicasting for Multimedia Applications," in *Proceedings of IEEE INFOCOM '92*, pp. 2078–2085, 1992.

[39] C. Noronha and F. Tobagi, "Optimum Routing of Multicast Streams," in *Proceedings of IEEE INFOCOM '94*, pp. 865–873, 1994.

[40] V. Kompella, J. Pasquale, and G. Polyzos, "Two Distributed Algorithms for the Constrained Steiner Tree Problem," in *Proceedings of the Second International Conference on Computer Communications and Networking (IC^3N '93)*, pp. 343–349, 1993.

[41] Q. Zhu, M. Parsa, and J. Garcia-Luna-Aceves, "A Source-Based Algorithm for Delay-Constrained Minimum-Cost Multicasting," in *Proceedings of IEEE INFOCOM '95*, pp. 377–385, 1995.

[42] B. Waxman, "Routing of Multipoint Connections," *IEEE Journal on Selected Areas in Communications*, vol. 6, no. 9, pp. 1617–1622, December 1988.

[43] A. Waters, "A New Heuristic for ATM Multicast Routing," in *Proceedings of the Second IFIP Workshop on Performance Modeling and Evaluation of ATM Networks*, pp. 8.1–8.9, July 1994.

[44] H. Salama, D. Reeves, Y. Viniotis, and T.-L. Sheu, "Comparison of Multicast Routing Algorithms for High-Speed Networks," Tech. Rep. TR 29.1930, IBM, September 1994.

[45] B. Waxman, "Performance Evaluation of Multipoint Routing Algorithms," in *Proceedings of IEEE INFOCOM '93*, pp. 980–986, 1993.

[46] J. Kadirire, "Minimising Packet Copies in Multicast Routing by Exploiting Geographic Spread," *Computer Communication Review*, vol. 24, no. 3, pp. 47–62, July 1994.

[47] J. Kadirire and G. Knight, "Comparison of Dynamic Multicast Routing Algorithms for Wide-Area Packet Switched (Asynchronous Transfer Mode) Networks," in *Proceedings of IEEE INFOCOM '95*, pp. 212–219, 1995.

[48] E. Biersack and J. Nonnenmacher, "WAVE: A New Multicast Routing Algorithm for Static and Dynamic Multicast Groups," in *Proceedings of the Fifth International Workshop on Network and Operating System Support for Digital Audio and Video*, pp. 228–239, 1995.

[49] F. Bauer and A. Varma, "ARIES: A Rearrangeable Inexpensive Edge-Based On-Line Steiner Algorithm," in *Proceedings of IEEE INFOCOM '96*, pp. 361–368, 1996.

[50] K. Barath-Kumar and J. Jaffe, "Routing to Multiple Destinations in Computer Networks," *IEEE Transactions on Communications*, vol. COM-31, no. 3, pp. 343–351, March 1983.

[51] G. Rouskas and I. Baldine, "Multicast Routing with End-to-End Delay and Delay Variation Constraints," in *Proceedings of IEEE INFOCOM '96*, pp. 353–360, 1996.

[52] H. Tode, Y. Sakai, H. Okada, and Y. Tezuka, "Multicast Routing Algorithm for Nodal Load Balancing," in *Proceedings of IEEE INFOCOM '92*, pp. 2086–2095, 1992.

[53] F. Bauer and A. Varma, "Degree-Constrained Multicasting in Point-to-Point Networks," in *Proceedings of IEEE INFOCOM '95*, pp. 369–376, 1995.

[54] M. Ammar, S. Cheung, and C. Scoglio, "Routing Multipoint Connections Using Virtual Paths in an ATM Networks," in *Proceedings of IEEE INFOCOM '93*, pp. 98–105, 1993.

[55] S.-B. Kim, "An Optimal VP-Based Multicast routing in ATM Networks," in *Proceedings of IEEE INFOCOM '96*, pp. 1302–1309, 1996.

[56] Y. Tanaka and P. Huang, "Multiple Destination Routing Algorithms," *IEICE Transactions on Communications*, vol. E76-B, no. 5, pp. 544–552, May 1993.

[57] L. Wei and D. Estrin, "The Trade-offs of Multicast Trees and Algorithms," in *Proceedings of the Third International Conference on Computer Communications and Networking (IC^3N '94)*, pp. 17–24, 1994.

[58] C. Noronha and F. Tobagi, "Evaluation of Multicast Routing Algorithms for Multimedia Streams," in *Proceedings of the IEEE International Telecommunications Symposium*, August 1994.

[59] C. Semeria and T. Maufer, "Introduction to IP Multicast Routing." Internet Draft, May 1996.

[60] S. Deering, "Host Extensions for IP Multicasting." Internet RFC 1112, `http://ds.internic.net/rfc/rfc1112.txt`, August 1989.

[61] W. Fenner, "Internet Group Management Protocol, Version 2." Internet Draft, May 1996.

[62] O. Hermanns and M. Schuba, "Performance Investigations of the IP Multicast Arcitecture," *Computer Networks and ISDN Systems*, vol. 28, no. 4, pp. 429–439, February 1996.

[63] D. Waitzman, C. Partridge, and S. Deering, "Distance Vector Multicast Routing Protocol." Internet RFC 1075, `http://ds.internic.net/rfc/rfc1075.txt`, November 1988.

[64] V. Kumar, *MBone: Interactive Multimedia On The Internet.* Macmillan Publishing, Simon & Schuster, 1995.

[65] C. Hedrick, "Routing Information Protocol." Internet RFC 1058, `http://ds.internic.net/ rfc/rfc1058.txt`, June 1988.

[66] G. Malkin, "RIP Version 2, Carrying Additional Information." Internet RFC 1723, `http://ds.internic.net/rfc/rfc1723.txt`, November 1994.

[67] T. Pusateri, "Distance Vector Multicast Routing Protocol." Internet Draft, July 1996.

[68] J. Moy, "Mutlicast Extension to OSPF." Internet RFC 1584, `http://ds.internic.net/ rfc/rfc1584.txt`, May 1994.

[69] J. Moy, "MOSPF, Analysis and Experience." Internet RFC 1585, `http://ds.internic.net/ rfc/rfc1585.txt`, May 1994.

[70] J. Moy, "Multicast Routing Extensions for OSPF," *Communications of the ACM*, vol. 37, no. 8, pp. 61–66, August 1994.

[71] J. Moy, "OSPF Version 2." Internet RFC 1583, `http://ds.internic.net/ rfc/rfc1583.txt`, March 1994.

[72] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C.-G. Liu, and L. Wei, "The PIM Architecture for Wide-Area Multicast Routing," *IEEE/ACM Transactions on Networking*, vol. 4, no. 2, pp. 153–162, April 1996.

[73] D. Estrin *et al.*, "Protocol Independent Multicast-Dense Mode (PIM-DM): Protocol Specification." Internet Draft, January 1996.

[74] S. Deering *et al.*, "Protocol Independent Multicast-Sparse Mode (PIM-SM): Protocol Specification." Internet Draft, June 1996.

[75] A. Ballardie, P. Francis, and J. Crowcroft, "Core Based Trees (CBT): An Architecture for Scalable Inter-Domain Multicast Routing," in *Proceedings of ACM SIGCOMM '93*, pp. 85–95, September 1993.

[76] A. Ballardie, *A New Approach to Multicast Communication in a Datagram Internetwork.* PhD thesis, University of London, May 1995.

[77] A. Ballardie, "Core Based Trees (CBT) Multicast Architecture." Internet Draft, February 1996.

[78] A. Ballardie, S. Reeve, and N. Jain, "Core Based Trees (CBT) Multicast: Protocol Specification." Internet Draft, April 1996.

[79] T. Billhartz, J. Cain, E. Farrey-Goudreau, D. Fieg, and S. Batsell, "Performance and Resource Cost Comparisons for CBT and PIM Multicast Routing Protocols in DIS Environments," in *Proceedings of IEEE INFOCOM '96*, pp. 85–93, 1996.

[80] Z. Zhang, C. Sanchez, B. Salkewicz, and E. Crawly, "Quality of Service Extensions to OSPF (QOSPF)." Internet Draft, June 1996.

[81] E. Crawley, "Multicast Routing over ATM." Internet Draft, February 1996.

[82] P. Manyem, *Routing Problems in Multicast Networks.* PhD thesis, North Carolina State University, August 1996.

[83] S. Rampal and D. Reeves, "An Evaluation of Routing and Admission Control Algorithms for Multimedia Traffic," *Computer Communications*, vol. 18, no. 10, pp. 755–768, October 1995.

[84] D. Mitzel, D. Estrin, S. Shenker, and L. Zhang, "An Architectural Comparison of ST-II and RSVP," in *Proceedings of IEEE INFOCOM '94*, 1994.

[85] S. Damaskos and H. Salama, "Reservation Mechanisms for Efficient Resource Management in Internetworks," in *Proceedings of the Third IEEE International Conference on Multimedia Computing and Systems (ICMCS '96)*, (Hiroshima, Japan), pp. 557–561, June 1996.

[86] H. Gabow, Z. Galil, T. Spencer, and R. Tarjan, "Efficient Algorithms for Finding Minimum Spanning Trees in Undirected and Directed Graphs," *Combinatorica*, vol. 6, no. 2, pp. 109–122, 1986.

[87] H. Abdel-Wahab, I. Stoica, and F. Sultan, "A Distributed Algorithm to Compute the Minimum Spanning Tree in a Communication Network," in *Proceedings of the Second Annual Joint Conference on Information Sciences*, pp. 429–432, September 1995.

[88] R. Gallager, P. Humblet, and P. Spira, "A Distributed Algorithm for Minimum-Weight Spanning Trees," *ACM Transactions on Programming Languages and Systems*, vol. 5, no. 1, pp. 66–77, January 1983.

[89] J. Garcia-Luna-Aceves and J. Behrens, "Distributed, Scalable Routing Based on Vectors of Link States," *IEEE Journal on Selected Areas in Communications*, vol. 13, no. 8, pp. 1383–1395, October 1995.

[90] R. Simha and B. Narahari, "Single Path Routing with Delay Considerations," *Computer Networks and ISDN Systems*, vol. 24, no. 5, pp. 405–419, June 1992.

[91] M. Aida, I. Nakamura, and T. Kubo, "Optimal Routing in Communication Networks with Delay Variations," in *Proceedings of IEEE INFOCOM '92*, pp. 153–159, 1992.

[92] S. Plotkin, "Competitive Routing of Virtual Circuits in ATM Networks," *IEEE Journal on Selected Areas in Communications*, vol. 13, no. 6, pp. 1128–1136, August 1995.

[93] W.-T. Chen and U.-J. Liu, "Routing Problem with Performance Requirement Translation for Multimedia Communications in an ATM Wide-Area Network," in *Proceedings of IEEE International Conference on Communications (ICC'94)*, pp. 1490–1494, 1994.

[94] L. Fratta, M. Gerla, and L. Kleinrock, "The Flow Deviation Method - An Approach to Store-and-Forward Communication Network Design," *Networks*, vol. 3, no. 2, pp. 97–133, 1973.

[95] J. Jaffe, "Algorithms for Finding Paths with Multiple Constraints," *Networks*, vol. 14, no. 1, pp. 95–116, Spring 1984.

[96] Z. Wang and J. Crowcroft, "Quality-of-Service Routing for Supporting Multimedia Applications," *IEEE Journal on Selected Areas in Communications*, vol. 14, no. 7, pp. 1228–1234, September 1996.

[97] S. Baase, *Computer Algorithms, Introduction to Design and Analysis.* Addison-Wesley Publishing Company, 2nd ed., 1988.

[98] Y.-W. Leung and T.-S. Yum, "Optimum Connection Paths for a Class of Video-Conferences," in *Proceedings of the IEEE International Conference on Communications (ICC '91)*, pp. 859–865, 1991.

[99] F. Buckley and F. Harary, *Distance in Graphs.* Addison-Wesley Publishing Company, 1990.

[100] G. Handler and P. Mirchnadani, *Location on Networks: Theory and Algorithms.* The MIT Press, 1979.

[101] P. Kolesar and W. Walker, "An Algorithm for the Dynamic Relocation of Fire Companies," *Operations Research*, vol. 22, no. 2, pp. 249–274, March–April 1974.

[102] G. List, P. Mirchandani, M. Turnquist, and K. Zografos, "Modeling and Analysis for Hazardous Materials Transportation: Risk Analysis, Routing Scheduling and Facility Location," *Transportation Science*, vol. 25, no. 2, pp. 100–114, May 1991.

[103] D. Trietsch, "Optimal Multifacility defensive Location on Planes with Rectilinear Distances," *Networks*, vol. 23, no. 6, pp. 517–523, September 1993.

[104] C. Toregas, R. Swain, C. ReVelle, and L. Bergman, "The Location of Emergency Service Facilities," *Operations Research*, vol. 19, no. 6, pp. 1363–1373, October 1971.

[105] S. Deering *et al.*, "Protocol Independent Multicast-Sparse Mode (PIM-SM): Protocol Specification." Internet Draft, September 1995.

[106] S. Shukla, E. Boyer, and J. Klinker, "Multicast Tree Construction in Network Topologies with Asymmetric Link Loads," Tech. Rep. NPS-EC-94-012, Naval Postgraduate School, Department of Electrical and Computer Engineering, September 1994.

[107] K. Calvert, E. Zegura, and M. Donahoo, "Core Selection Methods for Multicast Routing," in *Proceedings of the Fourth International Conference on Computer Communications and Networking (IC^3N '95)*, pp. 638–642, 1995.

[108] M. Donahoo and E. Zegura, "Core Migration for Dynamic Multicast Routing," in *Proceedings of the Fifth International Conference on Computer Communications and Networking (IC^3N '96)*, pp. 92–98, 1996.

[109] D. Thaler and C. Ravishankar, "Distributed Center-Location Algorithms: Proposals and Comparisons," in *Proceedings of IEEE INFOCOM '96*, pp. 75–84, 1996.

[110] R. Braden *et al.*, "Resource Reservation Protocol (RSVP) - Version 1 Functional Specification." Internet Draft, August 1996.

[111] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms.* MIT Press; McGraw-Hill, 1990.

Appendix

Appendix A

The Random Link Generator

The random link generation algorithm creates a connected network when given a set of nodes. The algorithm consists of two phases. The first phase is outlined below.

```
1.  given a set of nodes $V$ and their positions in a Cartesian plane, a desired average
    node degree $\overline{e}$, and the parameters $\alpha$ and $\beta$;
2.  $L :=$ the maximum distance between any two nodes in $V$;
3.  $E := \emptyset$;
4.  select a random node $u$;
5.  repeat {
6.      select a random node $v \neq u$ such that $(u,v) \notin E$ and $(v,u) \notin E$;
7.      $P_e(u,v) := \beta \exp \frac{-l(u,v)}{L\alpha}$; ($l(u,v)$ is the distance between node $u$ and $v$)
8.      generate a random real value $rnd$ such that $0 \leq rnd \leq 1$;
9.      if $rnd < P_e(u,v)$ then $E := E \cup (u,v) \cup (v,u)$[1];
10. } until the degree of $u$ is equal to 2;
11. for each node $u \in V$ {
12.     if the degree of $u$ is equal to 0 {
13.         select a random node $v \neq u$ such that $(u,v) \notin E$ and $(v,u) \notin E$
            and such that the node degree of $v$ is $\geq 1$;
14.         $P_e(u,v) := \beta \exp \frac{-l(u,v)}{L\alpha}$;
15.         generate a random real value $rnd$ such that $0 \leq rnd \leq 1$;
16.         if $rnd < P_e(u,v)$ then $E := E \cup (u,v) \cup (v,u)$;
```

[1]This means creating a directed link from u to v and another directed link from v to u.

```
17.     } until the degree of $u$ is equal to 1;
18.     if the degree of $u$ is equal to 1 {
19.         select a random node $v \neq u$ such that $(u, v) \notin E$ and $(v, u) \notin E$;
20.         $P_e(u, v) := \beta \exp \frac{-l(u,v)}{L\alpha}$;
21.         generate a random real value $rnd$ such that $0 \leq rnd \leq 1$;
22.         if $rnd < P_e(u, v)$ then $E := E \cup (u, v) \cup (v, u)$;
23.     } until the degree of $u$ is equal to 2;
24. };
```

Step 13 forces the network resulting at the end of the first phase to be connected. Any connected network must have an average node degree ≥ 2. It can be proven using induction that the average node degree at the end of phase 1 is $\leq (4 - (6/|V|))$. Therefore as $v \rightarrow \infty$ the average node degree at the end of phase 1 can not exceed 4. This means that our algorithm is not suitable for generating random networks with an average node degree of less than 4. In the second phase, the algorithm applies P_e to create links interconnecting randomly chosen nodes until the average node degree of the random network is equal to the desired average degree, $\overline{e}$. The average node degree of a network is equal to $(|E|/|V|)$. The second phase is outlined below.

```
25. while $(|E|/|V|) < \overline{e}$ {
26.     select two random nodes $u$ and $v$ such that $u \neq v$ and $(u, v) \notin E$
        and $(v, u) \notin E$;
27.     $P_e(u, v) := \beta \exp \frac{-l(u,v)}{L\alpha}$;
28.     generate a random real value $rnd$ such that $0 \leq rnd \leq 1$;
29.     if $rnd < P_e(u, v)$ then $E := E \cup (u, v) \cup (v, u)$;
30. };
```

Appendix B

Pseudo Code of DCUR

First, here is a list of all control messages exchanged between nodes executing DCUR:

- *Construct_Path*(source node, destination node, delay constraint value,
  value of the delay from the source to the downstream
  node receiving the message)
- *Query*(destination node)
- *Response*(destination node,
  least-delay value from the responding node to the destination)
- *Remove_Loop*(source node, destination node, delay constraint)
- *Acknowledge*(source node, destination node)

The following function is executed by the source node s when it receives a request from an application to construct a delay-constrained path to a destination node d. The elements of a routing table entry *rout*, presented in section 5.3, are accessed as $rout.source, rout.destination, \ldots$, etc.

```
1.  Initiate_Path_Construction(source node $s$, destination node $d$,
                                    delay constraint $\Delta$) {
2.      if $least\_delay\_value(s, d) > \Delta$ send a failure indication to the application;
3.      else {
4.          $active\_node := s$;
5.          $previous\_active\_node := null$;
6.          $delay\_so\_far := 0$;
7.          call Path_Construction($s$, $d$, $\Delta$, $delay\_so\_far$, $active\_node$,
                                            $previous\_active\_node$);
8.      };
9.  };
```

The following function is executed by *active_node* when it receives a *Construct_Path* message from *previous_active_node*. Also called at the source s to initiate the path construction.

```
1.  Path_Construction(source node $s$, destination node $d$, delay constraint $\Delta$,
                      current delay $delay\_so\_far$, current node $active\_node$,
                      previous node $previous\_active\_node$) {
2.      if $active\_node = d$ {
3.          create a new routing table entry, $rout$;
4.          $rout.source := s$;
5.          $rout.destination := d$;
6.          $rout.previous\_node := previous\_active\_node$;
7.          $rout.next\_node := null$;
8.          $rout.previous\_delay := delay\_so\_far$;
9.          $rout.flag := null$;
10.         send an $Acknowldge(s, d)$ message to $previous\_active\_node$;
11.     }
12.     else {
13.         if a routing table entry, $rout$ with $rout.source = s$ and
            $rout.destination = d$ already exists
14.             send a $Remove\_Loop(s, d, \Delta)$ message to $previous\_active\_node$;
15.         else {
16.             $use\_LDPATH := False$;
17.             if $least\_cost\_nhop(active\_node, d) = least\_delay\_nhop(active\_node, d)$
18.                 $use\_LDPATH := True$;
19.             if $use\_LDPATH = False$ {
20.                 $lc\_nhop := least\_cost\_nhop(active\_node, d)$;
21.                 send $Query(d)$ message to $lc\_nhop$;
22.                 wait to receive a $Response(d, delay)$ message from $lc\_nhop$;
23.                 if $(delay\_so\_far + D(active\_node, lc\_nhop) + delay) \leq \Delta$ {
24.                     create a new routing table entry, $rout$;
25.                     $rout.source := s$;
26.                     $rout.destination := d$;
27.                     $rout.previous\_node := previous\_active\_node$;
28.                     $rout.next\_node := lc\_nhop$;
29.                     $rout.previous\_delay := delay\_so\_far$;
30.                     $rout.flag := LCPATH$;
31.                     $delay\_so\_far := delay\_so\_far + D(active\_node, lc\_nhop)$;
32.                     send a $Construct\_Path(s, d, \Delta, delay\_so\_far)$ message to $lc\_nhop$;
33.                 }
34.                 else $use\_LDPATH := True$;
35.             };
36.             if $use\_LDPATH = True$ {
37.                 $ld\_nhop := least\_delay\_nhop(active\_node, d)$;
38.                 create a new routing table entry with, $rout$;
39.                 $rout.source := s$;
40.                 $rout.destination := d$;
41.                 $rout.previous\_node := previous\_active\_node$;
42.                 $rout.next\_node := ld\_nhop$;
43.                 $rout.previous\_delay := delay\_so\_far$;
44.                 $rout.flag := LDPATH$;
45.                 $delay\_so\_far := delay\_so\_far + D(active\_node, ld\_nhop)$;
46.                 send a $Construct\_Path(s, d, \Delta, delay\_so\_far)$ message to $ld\_nhop$;
47.             };
48.         };
49.     };
50. };
```

The following function is executed by node *current* when it receives a *Query* message from *active_node*.

```
1.  Process_Query(current node $current$, querying node $active\_node$, destination
    node $d$) {
2.      send a $Response(d, least\_delay\_value(current, d))$ message to $active\_node$;
3.  };
```

The following function is executed by *active_node* when it receives a *Remove_Loop* message.

```
1.  Loop_Removal(source $s$, destination $d$, current node $active\_node$, delay con-
    straint $\Delta$) {
2.      find the routing table entry, $rout$, with $rout.source = s$ and
        $rout.destination = d$;
3.      if $rout.flag = LCPATH$ {
4.          $nhop := least\_delay\_nhop(active\_node, d)$;
5.          $rout.flag := LDPATH$;
6.          $rout.next\_node := nhop$;
7.          $delay\_so\_far := rout.previous\_delay + D(active\_node, nhop)$;
8.          send a $Construct\_Path(s, d, \Delta, delay\_so\_far)$ message to $nhop$;
9.      }
10.     else {
11.         send a $Remove\_Loop(s, d, \Delta)$ message to the $rout.previous\_node$;
12.         delete $rout$;
13.     };
14. };
```

The following function is executed by a node *current* when it receives an *Acknowledge* message.

```
1.  Process_Acknowledge(source $s$, destination $d$, current node $current$) {
2.      if $current \neq s$ {
3.          find the routing table entry, $rout$, with $rout.source = s$ and
            $rout.destination = d$;
4.          send an $Acknowledge(s, d)$ message to $rout.previous\_node$;
5.      }
6.     else send an indication to the application that path construction is complete;
7. };
```

Appendix C

Pseudo Code of DCSHARED

First, here is a list of all control messages exchanged between nodes executing DC-SHARED:

- $Construct\_Path$(new member node, center node, group address, DCJOIN delay constraint value Δ', value of the delay from the joining node to the node receiving the message)
- $Query$(center node)
- $Response$(center node, least-delay value from the responding node to the center)
- $Remove\_Loop$(new member node, center node, group address, DCJOIN delay constraint value Δ')
- $Acknowledge$(new member node, center node, group address, value of the delay from the center to the node receiving the message)
- $Prune\_Path$(center node, group address)
- $Update\_Central\_Delay$(center node, group address, value of the delay from the center to the node receiving the message)

Each node n in the shared multicast tree has a permanent routing table entry which consists of the following fields:

- $center$ = the address of the center c,
- $address$ = the multicast group address i,

- *next_nodes* = list of addresses of all adjacent nodes on the shared tree,
- *nhop_to_center* = address of the next node on the shared tree in the direction of the center, and
- *central_delay* = delay from the center to node n along the shared tree.

When a node is the *active_node* during the path construction phase of DCJOIN it constructs a temporary routing table entry which consists of the following fields:

- *joining* = the address of the new member node m,
- *center* = the address of the center c,
- *address* = the multicast group address i,
- *previous_node* = address of the *previous_active_node*,
- $next\_node = \begin{cases} least\_cost\_nhop(active\_node, c) & \\ \qquad\qquad \text{if the LC path direction is chosen,} & \\ least\_delay\_nhop(active\_node, c) & \\ \qquad\qquad \text{if the LD path direction is chosen,} & \end{cases}$
- *previous_delay* = delay from m to the *active_node*, and
- $flag = \begin{cases} LCPATH & \text{if the LC path direction is chosen,} \\ LDPATH & \text{if the LD path direction is chosen.} \end{cases}$

C.1 Path Construction Phase of DCJOIN

It is assumed that a permanent routing table entry is established a the center of a multicast session immediately at session initiation time. Initially, the *next_nodes* list of that routing table entry is empty.

The functions listed in this section are executed during the path construction phase of DCJOIN. The following function is executed by the node m when it receives a request from an application to join a multicast group with address i and construct a delay-constrained path to the center c. The application's delay constraint is Δ. The DCJOIN delay constraint is $\Delta' \leq \Delta/2$.

```
1. Initiate_Path_Construction(new member node $m$, center node $c$,
                              group address $i$, delay constraint $\Delta'$) {
```

```
2.    if $least\_delay\_value(m, c) > \Delta'$ send a failure indication to the application;
3.    else {
4.        $active\_node := m$;
5.        $previous\_active\_node := null$;
6.        $delay\_so\_far := 0$;
7.        call Path_Construction($m$, $c$, $i$, $\Delta'$, $delay\_so\_far$, $active\_node$,
                                  $previous\_active\_node$);
8.    };
9. };
```

The following function is executed by *active_node* when it receives a *Construct_Path* message from *previous_active_node*. Also called at the new member node m to initiate the path construction.

```
1. Path_Construction(new member node $m$, center node $c$, group address $i$, delay
                     constraint $\Delta'$, current delay $delay\_so\_far$, current
                     node $active\_node$, previous node $previous\_active\_node$) {
2.    if $active\_node$ has a permanent routing table entry, $perm$, with
      $perm.center = c$ and $perm.address = i$ and
      $(delay\_so\_far + perm.central\_delay) \leq \Delta'$ {
3.        $perm.next\_nodes := previous\_active\_node \cup perm.next\_nodes$;
4.        $delay\_from\_center := perm.central\_delay +$
                                 $D(active\_node, previous\_active\_node)$;
5.        send an $Acknowledge(m, c, i, delay\_from\_center)$ message to
          $previous\_active\_node$;
6.    }
7.    else {
8.        if $active\_node$ has a temporary routing table entry, $temp$, with
          $temp.joining = m$ and $temp.center = c$ and $temp.address = i$
9.            send a $Remove\_Loop(m, c, i, \Delta')$ message to $previous\_active\_node$;
10.       else {
11.           $use\_LDPATH := False$;
12.           if $least\_cost\_nhop(active\_node, c) = least\_delay\_nhop(active\_node, c)$
```

```
13.             $use\_LDPATH := True$;
14.         if $use\_LDPATH = False$ {
15.             $lc\_nhop := least\_cost\_nhop(active\_node, c)$;
16.             send $Query(c)$ message to $lc\_nhop$;
17.             wait to receive a $Response(c, delay)$ message from $lc\_nhop$;
18.             if $(delay\_so\_far + D(active\_node, lc\_nhop) + delay) \leq \Delta'$ {
19.                 create a new temporary routing table entry, $temp$;
20.                 $temp.joining := m$;
21.                 $temp.center := c$;
22.                 $temp.address := i$;
23.                 $temp.previous\_node := previous\_active\_node$;
24.                 $temp.next\_node := lc\_nhop$;
25.                 $temp.previous\_delay := delay\_so\_far$;
26.                 $temp.flag := LCPATH$;
27.                 $delay\_so\_far := delay\_so\_far + D(active\_node, lc\_nhop)$;
28.                 send a $Construct\_Path(m, c, i, \Delta', delay\_so\_far)$ message to
                    $lc\_nhop$;
29.             }
30.             else $use\_LDPATH := True$;
31.         };
32.         if $use\_LDPATH = True$ {
33.             $ld\_nhop := least\_delay\_nhop(active\_node, c)$;
34.             create a new temporary routing table entry, $temp$;
35.             $temp.joining := m$;
36.             $temp.center := c$;
37.             $temp.address := i$;
38.             $temp.previous\_node := previous\_active\_node$;
39.             $temp.next\_node := ld\_nhop$;
40.             $temp.previous\_delay := delay\_so\_far$;
41.             $temp.flag := LDPATH$;
42.             $delay\_so\_far := delay\_so\_far + D(active\_node, ld\_nhop)$;
43.             send a $Construct\_Path(m, c, i, \Delta', delay\_so\_far)$ message to
```

```
                    $ld\_nhop$;
44.             };
45.         };
46.     };
47. };
```

The following function is executed by node n when it receives a *Query* message from *active_node*.

```
1. Process_Query(current node $n$, querying node $active\_node$, center node $c$) {
2.     send a $Response(c, least\_delay\_value(n, c))$ message back to $active\_node$;
3. };
```

The following function is executed by *active_node* when it receives a *Remove_Loop* message.

```
1. Loop_Removal(new member node $m$, center node $c$, group address $i$,
                     current node $active\_node$, DCJOIN delay constraint $\Delta'$) {
2.     find the temporary routing table entry, $temp$, with $temp.joining = m$,
       $temp.center = c$, and $temp.address = i$;
3.     if $temp.flag = LCPATH$ {
4.         $nhop := least\_delay\_nhop(active\_node, c)$;
5.         $temp.flag := LDPATH$;
6.         $temp.next\_node := nhop$;
7.         $delay\_so\_far := temp.previous\_delay + D(active\_node, nhop)$;
8.         send a $Construct\_Path(m, c, i, \Delta', delay\_so\_far)$ message to $nhop$;
9.     }
10.    else {
11.        send a $Remove\_Loop(m, c, i, \Delta')$ message to the $temp.previous\_node$;
12.        delete $temp$;
13.    };
14. };
```

C.2 Acknowledgment Phase of DCJOIN

The acknowledgment phase serves the following purposes: creating permanent routing table entries and removing the temporary entries along the constructed path, removing loops existing at the end of the path construction phase, updating the central delays for all affected nodes in the shared tree, and informing the new member node that a connection to the shared tree has been established.

The following function is executed when a node, *current*, receives an *Acknowledge* message.

```
 1. Process_Acknowledge(new member node $m$, center node $c$, group address $i$,
                        current node $current$,
                        delay from the center $delay\_from\_center$) {
 2.    find the temporary routing table entry, $temp$, with $temp.joining = m$,
       $temp.center = c$, and $temp.address = i$;
 3.    if a permanent routing table entry, $perm$, with $perm.center = c$ and
       $perm.address = i$ exists {
 4.       send a $Prune\_Path(c, i)$ message to the $perm.nhop\_to\_center$;
 5.       $perm.next\_nodes := perm.next\_nodes - perm.nhop\_to\_center$;
 6.       $perm.nhop\_to\_center := temp.next\_node$;
 7.       $perm.central\_delay := delay\_from\_center$;
 8.       for each node $v \in perm.next\_nodes$ {
 9.          if $v \neq temp.next\_node$ and $v \neq temp.previous\_node$ {
10.             $delay := delay\_from\_center + D(current, v)$;
11.             send an $Update\_Central\_Delay(c, i, delay)$ message to $v$;
12.          };
13.       };
14.       $perm.next\_nodes := perm.next\_nodes \cup temp.next\_node \cup$
                 $temp.previous\_node$;
15.    }
16.    else {
17.       create a new permanent routing table entry, $perm$;
18.       $perm.center := c$;
19.        $perm.address := i$;
20.        $perm.next\_nodes := \{temp.next\_node, temp.previous\_node\}$;
21.        $prem.nhop\_to\_center := temp.next\_node$;
22.        $perm.central\_delay := delay\_from\_center$;
23.    };
24.    if $current \neq m$ {
25.        $delay\_from\_center := delay\_from\_center +$
                                  $D(current, temp.previous\_node)$;
26.        send an $Acknowledge(m, c, i, delay\_from\_center)$ message to
           $temp.previous\_node$;
27.    }
28.    else $m$ sends a success indication to the application layer;
29.    delete $temp$;
30. };
```

The following function is executed when a node, *current*, receives a *Prune_Path* message from a node *previous*.

```
1. Process_Prune(center node $c$, current node $current$, group address $i$,
                 previous node $previous$) {
2.     find the permanent routing table entry, $perm$, with $perm.center = c$
       and $perm.address = i$;
3.     $perm.next\_nodes := perm.next\_nodes - previous$;
4.     if $current$ is not a member in the multicast group and $current \neq c$ and
       $perm.nhop\_to\_center$ is the only member in $perm.next\_nodes$ {
5.         send a $Prune\_Path(c, i)$ message to $perm.nhop\_to\_center$;
6.         delete perm;
7.     };
8. };
```

The following function is executed when a node, *current*, receives an *Update_Central_Delay* message.

```
1. Process_Update_Delay(center node $c$, group address $i$, current node $current$,
                        delay value $delay\_from\_center$) {
```

```
2.      find the permanent routing table entry, $perm$, with $perm.center = c$
        and $perm.address = i$;
3.      $perm.central\_delay := delay\_from\_center$;
4.      for each node $v \in perm.next\_nodes$ {
5.          if $v \neq perm.nhop\_to\_center$ {
6.              $delay := delay\_from\_center + D(current, v)$;
7.              send an $Update\_Central\_Delay(c, i, delay)$ message to $v$;
8.          };
9.      };
10. };
```

C.3 Pseudo Code of DCLEAVE

The following function is executed by the multicast group member node node m when it receives a request from the application to leave the multicast group. Its objective is to prune the shared multicast tree after node m leaves.

```
1.  Process_Leave(leaving node $m$, center node $c$, group address $i$) {
2.      find the permanent routing table entry, $perm$, with $perm.center = c$
        and $perm.address = i$;
3.      if $m \neq c$ and $perm.nhop\_to\_center$ is the only member in $perm.next\_nodes$ {
4.          send a $Prune\_Path(c, i)$ message to $perm.nhop\_to\_center$;
5.          delete perm;
6.      };
7.  };
```

Appendix D

Pseudo Code of DCINITIAL

We denote the multicast group member that starts the operation of DCINITIAL as the source node and the multicast group member farthest away from the source node as the destination node. Here is a list of all control messages exchanged between nodes executing DCINITIAl.

- *Construct_Path*(source node, destination node, group address, delay constraint value, value of the delay from the source node to the node receiving the message)
- *Query*(destination node)
- *Response*(destination node, least-delay value from the responding node to the destination)
- *Remove_Loop*(source node, destination node, group address, delay constraint value)
- *Find_Center*(source node, destination node, group address, delay constraint value, value of the delay from the destination to the node receiving the message, value of the total delay from source to destination)
- *Set_Routing_Entry*(source node, destination node, center node, group address, value of the delay from the center to the node receiving the message)

After the execution of DCINITIAL is completed, each node in the initial shared tree will have a permanent routing table entry which is identical to the one used by DCSHARED and presented in appendix C. This compatibility between DCINITIAL and DCSHARED is necessary so that multicast group members can join or leave the the initial shared multicast tree. DCINITIAL uses temporary routing table entries that identical to those of DCSHARED's temporary routing table entries. However, with slightly different terminology to simplify the presentation of the pseudo code. A temporary routing table entry consists of the following fields:

- $source$ = the address of the source node s,
- $destination$ = the address of the destination node d,
- $address$ = the multicast group address i,
- $previous\_node$ = address of the $previous\_active\_node$,
- $next\_node = \begin{cases} least\_cost\_nhop(active\_node, c) & \\ \qquad \text{if the LC path direction is chosen,} \\ least\_delay\_nhop(active\_node, c) & \\ \qquad \text{if the LD path direction is chosen,} \end{cases}$
- $previous\_delay$ = delay from s to the $active\_node$, and
- $flag = \begin{cases} LCPATH & \text{if the LC path direction is chosen,} \\ LDPATH & \text{if the LD path direction is chosen.} \end{cases}$

D.1 Path Construction Phase of DCINITIAL

The functions listed in this section are executed during the path construction phase of DCINITIAL. The following function is executed by the node s to start construction of the initial shared tree (a delay-constrained path).

```
1. Initiate_Path_Construction(source node $s$, group address $i$, delay constraint $\Delta$)
   {
2.     search the delay vector for the multicast group member $d$ farthest away from
       $s$ such that the delay between $s$ and $d$ does not violate $\Delta$;
3.     if $d = null$ send a failure indication to application;
4.     else {
5.         $active\_node := s$;
6.         $previous\_active\_node := null$;
7.         $delay\_so\_far := 0$;
8.         call Path_Construction($s$, $d$, $i$, $\Delta$, $delay\_so\_far$, $active\_node$,
                                  $previous\_active\_node$);
9.     };
10. };
```

The following function is executed by *active_node* when it receives a *Construct_Path* message from *previous_active_node*. Also called at the source node s to initiate the path construction.

```
1.  Path_Construction(source node $s$, destination node $d$, group address $i$, delay
                      constraint $\Delta$, current delay $delay\_so\_far$, current
                      node $active\_node$, previous node $previous\_active\_node$) {
2.     if $active\_node = d$ {
3.         create a new temporary routing table entry, $temp$;
4.         $temp.source := s$;
5.         $temp.destination := d$;
6.         $temp.address := i$;
7.         $temp.previous\_node := previous\_active\_node$;
8.         $temp.next\_node := null$;
9.         $temp.previous\_delay := delay\_so\_far$;
10.        $temp.flag := null$;
11.        $delay\_from\_destination := D(active\_node, previous\_active\_node)$;
12.        $total\_delay := delay\_so\_far$;
13.        send a $Find\_Center(s, d, i, \Delta, delay\_from\_destination, total\_delay)$
           message to $previous\_active\_node$.
14.    }
15.    else {
16.        if a temporary routing table entry, $temp$ with $temp.source = s$ and
           $temp.destination = d$ and group address $i$ already exists
```

```
17.             send a $Remove\_Loop(s, d, i, \Delta)$ message to $previous\_active\_node$;
18.         else {
19.             $use\_LDPATH := False$;
20.             if $least\_cost\_nhop(active\_node, d) = least\_delay\_nhop(active\_node, d)$
21.                 $use\_LDPATH := True$;
22.             if $use\_LDPATH = False$ {
23.                 $lc\_nhop := least\_cost\_nhop(active\_node, d)$;
24.                 send $Query(d)$ message to $lc\_nhop$;
25.                 wait to receive a $Response(d, delay)$ message from $lc\_nhop$;
26.                 if $(delay\_so\_far + D(active\_node, lc\_nhop) + delay) \leq \Delta$ {
27.                     create a new temporary routing table entry, $temp$;
28.                     $temp.source := s$;
29.                     $temp.destination := d$;
30.                     $temp.address := i$;
31.                     $temp.previous\_node := previous\_active\_node$;
32.                     $temp.next\_node := lc\_nhop$;
33.                     $temp.previous\_delay := delay\_so\_far$;
34.                     $temp.flag := LCPATH$;
35.                     $delay\_so\_far := delay\_so\_far + D(active\_node, lc\_nhop)$;
36.                     send a $Construct\_Path(s, d, i, \Delta, delay\_so\_far)$ message
                        to $lc\_nhop$;
37.                 }
38.                 else $use\_LDPATH := True$;
39.             };
40.             if $use\_LDPATH = True$ {
41.                 $ld\_nhop := least\_delay\_nhop(active\_node, d)$;
42.                 create a new temporary routing table entry with, $temp$;
43.                 $temp.source := s$;
44.                 $temp.destination := d$;
45.                 $temp.address := i$;
46.                 $temp.previous\_node := previous\_active\_node$;
47.                 $temp.next\_node := ld\_nhop$;
48.                 $temp.previous\_delay := delay\_so\_far$;
49.                 $temp.flag := LDPATH$;
50.                 $delay\_so\_far := delay\_so\_far + D(active\_node, ld\_nhop)$;
51.                 send a $Construct\_Path(s, d, i, \Delta, delay\_so\_far)$ message to $ld\_nhop$;
52.             };
53.         };
54.     };
55. };
```

The following function is executed by node n when it receives a *Query* message from *active_node*.

```
1. Process_Query(current node $n$, querying node $active\_node$, destination node $d$)
   {
2.     send a $Response(d, least\_delay\_value(n, d))$ message back to $active\_node$;
3. };
```

The following function is executed by *active_node* when it receives a *Remove_Loop* message.

```
1. Loop_Removal(source node $s$, destination node $d$, group address $i$,
                current node $active\_node$, delay constraint $\Delta$) {
2.     find the temporary routing table entry, $temp$, with $temp.source = s$,
       $temp.destination = d$, and $temp.address = i$;
3.     if $temp.flag = LCPATH$ {
4.         $nhop := least\_delay\_nhop(active\_node, d)$;
5.         $temp.flag := LDPATH$;
6.         $temp.next\_node := nhop$;
7.         $delay\_so\_far := temp.previous\_delay + D(active\_node, nhop)$;
8.         send a $Construct\_Path(s, d, i, \Delta, delay\_so\_far)$ message to $nhop$;
9.     }
10.    else {
11.        send a $Remove\_Loop(s, d, i, \Delta)$ message to the $temp.previous\_node$;
12.        delete $temp$;
13.     };
14. };
```

D.2 Acknowledgment Phase of DCINITIAL

The acknowledgment phase serves the following purposes: finding the center of the multicast session and computing the constraint Δ' and creating permanent routing table entries and removing the temporary entries along the constructed path.

The following function is executed when a node, *current*, receives a *Find_Center* message.

```
1. Process_Find_Center(source node $s$, destination node $d$, group address $i$,
                       delay constraint value $\Delta$, value of the delay from the
                       destination to the node receiving the message
                       $delay\_from\_destination$, value of the total delay from
                       source to destination $total\_delay$, current node
                       $current$) {
2.    find the temporary routing table entry, $temp$, with $temp.source = s$,
      $temp.destination = d$, and $temp.address = i$;
3.    if max($delay\_from\_destination, total\_delay - delay\_from\_destination$) <
        max($delay\_from\_destination + D(current, temp.previous\_node)$,
            $total\_delay - delay\_from\_destination -$
                        $D(current, temp.previous\_node)$)) {
            (comment: this if statement ensures that $current$ is closer to the
             center of the path than any other node)
4.       select this node to be the center $c := current$;
5.       $\Delta' := \min(\frac{\Delta}{2}, \Delta - \max(delay\_from\_destination,$
                          $total\_delay - delay\_from\_destination))$;
6.       create a new permanent routing table entry, $perm$;
7.       $perm.center := c$;
8.       $perm.address := i$;
9.       $perm.next\_nodes := \{temp.next\_node, temp.previous\_node\}$;
10.      $prem.nhop\_to\_center := null$;
11.         $perm.central\_delay := 0$;
12.         send a $Set\_Routing\_Entry(s, d, c, i, D(current, temp.next\_node))$
            to $temp.next\_node$.
13.         send a $Set\_Routing\_Entry(s, d, c, i, D(current, temp.previous\_node))$
            to $temp.previous\_node$.
14.         delete $temp$;
15.         advertize the address of $c$ and the value of $\Delta'$ to all nodes in the network;
16.     }
17.     else {
18.         $delay\_from\_destination :=$
                $delay\_from\_destination + D(current, temp.previous\_node)$;
19.         send a $Find\_Center(s, d, i, \Delta, delay\_from\_destination, total\_delay)$
            message to $temp.previous\_node$.
20.     };
21. };
```

The following function is executed when a node, *current*, receives a *Set_Routing_Entry* message from *previous_node*.

```
1. Process_Set_Routing_Entry(source node $s$, destination node $d$, center node $c$,
                             group address $i$, value of the delay from the
                             center to the node receiving the message
                             $delay\_from\_center$, current node $current$,
                             previous node $previous$) {
2.     find the temporary routing table entry, $temp$, with $temp.source = s$,
       $temp.destination = d$, and $temp.address = i$;
3.     create a new permanent routing table entry, $perm$;
4.     $perm.center := c$;
5.     $perm.address := i$;
6.     $perm.next\_nodes := \{temp.next\_node, temp.previous\_node\}$;
7.     $prem.nhop\_to\_center := previous$;
8.     $perm.central\_delay := delay\_from\_center$;
9.     if $previous = temp.previous\_node$ {
```

```
10.         if $temp.next\_node \neq null$ {
11.             $delay\_from\_center := delay\_from\_center +$
                                                $D(current, temp.next\_node)$;
12.             send a $Set\_Routing\_Entry(s, d, c, i, delay\_from\_center)$ to
                $temp.next\_node$.
13.         };
14.     }
15.     else {
16.         if $temp.previous\_node \neq null$ {
17.             $delay\_from\_center := delay\_from\_center +$
                                                $D(current, temp.previous\_node)$;
18.             send a $Set\_Routing\_Entry(s, d, c, i, delay\_from\_center)$ to
                $temp.previous\_node$.
19.         };
20.     };
21.     delete $temp$;
22. };
```