UNIVERSITY OF CINCINNATI

Date: May 8th, 2007

I, Michael T. Helmick,

hereby submit this work as part of the requirements for the degree of:

Doctor of Philosophy (Ph.D.)

in:

Computer Science and Engineering

It is entitled:

Efficient Group Communication and the

Degree-bounded Shortest Path Problem

This work and its defense approved by:

Chair: Dr. Fred Annexstein

Dr. Kenneth Berman

Dr. Jerome Paul

Dr. Karen Tomko

Dr. Gary Lewandowski

Efficient Group Communication and the Degree-bounded Shortest Path Problem

A dissertation submitted to the

Division of Research and Advanced Studies
of the University of Cincinnati

in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in the Department of
Computer Science
of the College of Engineering

May 2007

by

Michael T. Helmick
M.S., Northern Kentucky University, 2004, Highland Heights, Kentucky
B.S., Xavier University, 2000, Cincinnati, Ohio

Thesis Advisor and Committee Chair: Fred S. Annexstein, Ph.D.

Abstract

In this thesis we develop a framework for studying and understanding the tradeoffs involved in efficient multicast route determination. Using this framework, we developed an algorithm (**Myriad**) that exhibits superior theoretical and empirical results over previously published work. We have validated this work through extensive simulation using a variety of metric spaces, and by proving a series of mathematical theorems concerning degree-bounded spanning trees over metric spaces.

This study is done through examination of the *degree-constrained minimum average-latency spanning tree problem* (or DC-MAL). Creating a solution which places each participant at their optimal locality along a path from the root is an NP-hard problem, thus motivating a search for approximate solutions. This type of organization for information relaying is a natural model of the multicast problem and lends itself to theoretical analysis that can be applied in application level peer-to-peer overlay networks. Out-degree constraints follow directly from the need for high bandwidth multimedia streams and the ability to relay the information without any loss. Study of this problem is important since it is expected that the demand for multicast applications will continue to grow, and such applications require scalability to millions of simultaneous subscribers.

Acknowledgments

Academic endeavors such at this are large enough in scope that they span many years, experiences, and even life stages. The incredible commitment required for these pursuits is held in place by an extensive support network of family and friends. To all my friends and family members, thank you for being supportive of my research over the past five years. The most important person in my support network continues to be my best friend and wife, Cortny Helmick. She has been extraordinarily supportive and is my biggest fan, as I am hers.

No graduate student can succeed without academic mentors. I have been influenced in a positive way by many of my professors over the years and would like to extend my gratitude to those who have provided instruction, guidance, and mentoring during my time at Xavier University, Northern Kentucky University, and the University of Cincinnati. In particular, success would not be possible without the support and guidance of my dissertation committee, Ken Berman, Jerry Paul, Karen Tomko, and Gary Lewandowski.

I would like to extend two special notes of thanks. First, to Dr. Gary Lewandowski who provided inspiration and guidance while I was student at Xavier University, and who has served on both of my graduate committees, guiding my academic career for 10 years. And to my committee chair and advisor, Fred Annexstein, for always providing just the right amount of guidance as I progressed in my research over the past two and a half years.

Mike Helmick

Cincinnati, Ohio

May 2007

Table of Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

The problem of multicasting on the Internet is that of finding communication paths from one source to multiple recipients [7, 9]. Internet Protocol (IP) multicasting, as it is known today, is based on the work of Stephen Deering [30]. Prior to the proposal for bridging isolated datagram networks [28, 29, 30], multicast was restricted to local area networks. Multicasting on the Internet is difficult due to the number of potential participants (up to thousands or even millions), managing subscription groups (introducing state into a stateless system [24]), and bandwidth requirements for multimedia content being sent from a single source to a large, distributed subscriber base.

Study of this problem is important since it is expected that the demand for multicast applications will continue to grow, and such applications require scalability to millions of simultaneous subscribers. Use of unicast transmission to service large subscriber groups only wastes bandwidth and becomes impractical as the amount bandwidth required for individual streams grows [69].

In this thesis, we attack the multicast problem by providing a framework for experimentation, evaluation, and algorithm development. We bring forward the primary factors for multicast tree evaluation (through examining existing metrics and introducing new metrics,

see Section 1.2) and concentrate on minimizing the average-latency from the source of a communication to all recipients, where latency represents the minimum communication traversal time along the path from the source to a destination. The following is our thesis statement.

> *In this thesis we develop a framework for studying and understanding the tradeoffs involved in efficient multicast route determination. Using this framework, we developed an algorithm (***Myriad***) that exhibits superior theoretical and empirical results over previously published work. We have validated this work through extensive simulation using a variety of metric spaces, and by proving a series of mathematical theorems concerning spanning trees over metric spaces.*

Our process of experimentation and algorithm development has uncovered tradeoffs between tree weight and latency and maximum depth and latency [41], with the depth-latency tradeoff being the primary one addressed here. All of this is done within the context of models of the the Internet with the goals of satisfying bandwidth requirements and reducing redundant transmissions using volunteer computing via end-systems on the fringes of the Internet [24].

Due to lack of wide spread adoption of IP multicasting, technical limitations, and connectivity restrictions, application level control of multicasting functionality is a focus of current research [23, 25, 32, 34, 45, 69]. In this thesis, we examine the problem of application level multicasting as an alternative to network level multicasting. Among Internet standards, the Distance Vector Multicast Routing Protocol (DVMRP) [6] continues to be the primary method of determining multicast routes and is based on *reverse path forwarding* [43]. We propose alternative solutions that take into consideration more information for informed calculation as an alternative to both network level and application level reverse path forwarding solutions (such as [20]).

The ability to calculate efficient multicast routes is important because of the practical ap-

plications that can be built with trees that exhibit low average-latency. Continued research on this problem as an enabling technology for application level (or end system) multicasting on the Internet is important because of the current and potential uses for multicast distribution and peer-to-peer technology to power applications and deliver information. Multicast has the ability to reduce server load and delivery time [57, 60]. Several applications have been built that have been enabled by multicasting, including a stock trading system [1], online collaboration [49], general content distribution [40], and, perhaps most often cited, delivery of streaming multimedia content [24, 26, 58, 61, 64]. We believe that the delivery of rich multimedia content will continue to gain importance in the future as computing becomes ubiquitous [48].

Examination of this problem is warranted because of the potential impact of efficient route construction for information dissemination on the Internet. As bandwidth intensive content (live and pre-recorded broadcast video content) and frequently updated items (discrete news content) become more widely utilized on the Internet, it becomes desirable for server providers to reduce their processing and bandwidth requirements through multicast. In order to enable near real-time data delivery by the creation of multicast routes that minimize the average-latency and reduce network stress through enforcements of degree constraints, algorithms are needed to produce the required multicast trees.

In this thesis, we present a framework for studying, understanding, and designing algorithmic solutions for the multicasting problem. Our framework focuses on minimizing the average-latency of end systems under a model of network bandwidth constraints realized by constraining the out-degree of multicast tree solutions. In this framework, we consider the distortion of degree-bounded solution trees measured over an underlying metric space. In Chapter 4 we prove some lower bounds (Theorem 2.2) on the distortion over metric spaces associated with hypercube topologies. In Chapter 2 we prove upper bounds on the distortion of trees constructed by our heuristic algorithm *Myriad* on general metric spaces

(Theorem 2.1). In Chapter 4 we present an extensive experimental study which compares the (distortion) performance of Myriad to the best known solutions over a variety of metric spaces.

In this thesis we present a new algorithm, named *Myriad*, which calculates efficient multicast distribution paths in a graph. The goal of any such algorithm is to provide paths that deliver the desired information with minimum average-latency for all participants while remaining within the physical constraints of the system. For computer networks, degree constraints are the logical representation of bandwidth limitations at end systems on a network.

Many fundamental network optimization problems, such as minimum cost shortest path, become NP-hard with degree constraints [35, 53]. Solutions to these types of problems are well studied and continue to be studied and combined as variations of bicriteria problems related to minimum cost degree-bounded network design [16, 39, 41, 42, 47, 52, 53].

In this thesis, we examine the area of network design problems in the context of minimizing the average delivery time for all participants. Specifically, we examine the *degree-constrained minimum average-latency spanning tree problem* (hereafter referred to as *DC-MAL*) as it relates to a modeling of application level multicasting in peer-to-peer systems on the Internet. *DC-MAL* is fully described in Section 1.1.

During the process of this investigation we have created a new heuristic algorithm, *Myriad*, to create trees that provide approximate solutions to *DC-MAL*. Using high performance computing and custom simulation software, we have compared *Myriad* to some of the best performing algorithms in this area. We have studied comparisons over several classes of underlying metrics. Our focus is on the average-latency and resultant tree height for calculated multicast distribution trees. Our empirical study focused on comparisons with three previous works, the KLS [42] algorithm for minimum diameter degree-bounded spanning trees, the greedy MDDBST [60] algorithm designed for multicasting, and the *Overlay Multicast*

Network Infrastructure (OMNI) [16]. A full description of the experimental methodology employed can be found in Chapter 4. The results of our experiments indicated that there is a tradeoff between achieving a lower average-latency value and producing a tree of minimum height. We refer to a tree of minimum possible height, considering an out-degree constraint, to be a *compact tree* and demonstrate the effect on average-latency when forcing a tree to be compact. *Myriad* is shown to outperform existing work both through the examination of theoretical bounds and extensive experimentation.

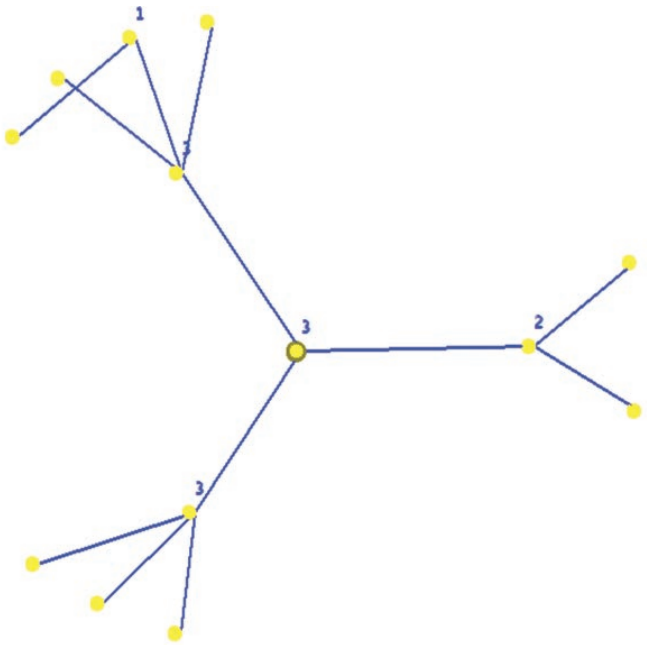


Figure 1.1: Sample multicast tree calculated by *Myriad*, non compact.

To highlight the effects on average-latency by forcing a compact tree, we refer to Figure 1.1 and Figure 1.2. Both of these trees have been calculated over the same input metric (Euclidean 2-d space) using the *Myriad* algorithm, with Figure 1.2 showing a forcibly compact tree. In this example the ratio of average-latency in the compact tree to the non-compact tree is $\frac{272.3846}{254.3846} = 1.071$. The difference in depth is 1, with the compact tree using the minimum required 3 levels, and the non-compact version using 4 levels.

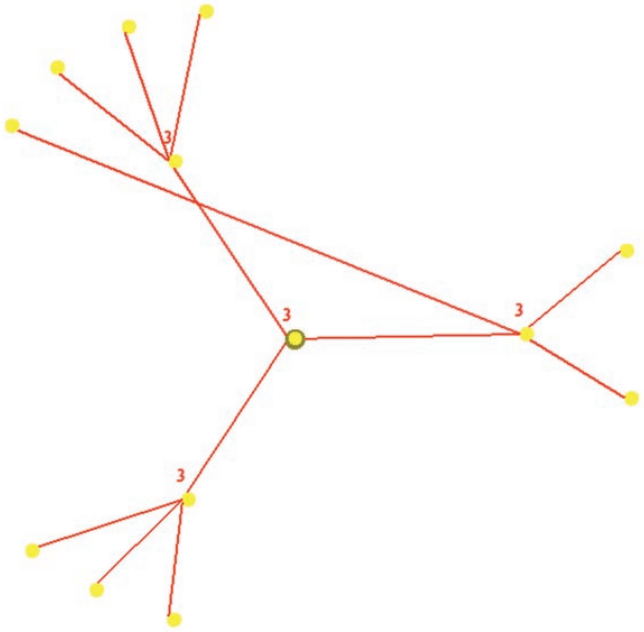


Figure 1.2: Sample multicast tree calculated by *Myriad*, compact.

This work also contributes a set of performance metrics for describing and comparing multicast trees. These metrics measure things such as average depth (hops), maximum depth, average load (out degree for non-leaf nodes), total delivery time, average delivery time for all nodes, and longest path delivery time. This advances a methodology for examining and categorizing depth-latency tradeoffs in multicast systems and provides a framework for study and experimentation on this and related issues. The framework we have developed consists of a software implementation of an extensible graph structure and simulator for examining network design problems (Chapter 4). Our definitions of important multicast tree metrics (Section 1.1) and discussion of experimental results (Section 4.5) provide a conceptual framework for discussion of the multicast problem and comparison of solutions to this problem.

1.1 Multicast Problem Definition

On the Internet, two end systems are connected by multiple physical links spanned by middleboxes such as routers, switches, firewalls, and network address translators [13]. For purposes of computation, the path between two end systems is abstracted to a single virtual link and assigned a representative weight based on the latency (minimum one way trip time) between the two hosts. This abstraction follows from IP network routing functions.

In order to calculate an effective multicast tree over a particular set of nodes, we need to be able to measure or derive the latency between all pairs of nodes, allowing a complete graph to be constructed. We study the problem in a graph theoretic model, where the input is a weighted complete graph, $G = (V, E)$. V is the set of all vertices (end systems in the network) and E is the set of weighted, undirected edges between all nodes. We focus on networks in which all nodes are subscribers of the multicast group or one in which non-subscribers can be ignored. The edges in the graph are undirected, indicating that there is the potential for information to flow in either direction. Edges are assigned weights corresponding to the latency between the nodes they connect.

In this thesis and in [41], we define a bicriteria problem with these constraints; (1) *minimize the maximum depth in the tree* and (2) *minimize the average-latency* in the class of out-degree B spanning trees. As a performance metric we will compute the product of *average-latency* and *maximum depth* and show results for constraining the maximum depth to the minimum possible depth, and allowing the height of the tree to change based on the graph. By examining the normalized product of these two metrics we are able to compare the blended result for all of the tested algorithms. Bounding the maximum depth of the tree, and therefore bounding the maximum hops, is a meaningful metric for networks in which time to live (TTL) is a parameter on messages, i.e. Gnutella [2]. Reducing the number of hops between the root and leaves also lowers the number of potential failure points along any

given root to leaf path. We will refer to this problem as the *degree-constrained, minimum average-latency spanning tree problem* (or **DC-MAL** for short).

The definition of the DC-MAL problem follows from two similar problems.

1. **DC-MD**: The degree-constrained minimum diameter problem is defined by *minimizing the diameter of the tree* in the class of out-degree B spanning trees.

2. **DC-MML**: The degree-constrained minimum maximum-latency problem is defined by *minimizing the maximum-latency* in the class of out-degree B spanning trees.

These three problems are all related and can be combined to create more restrictive problem specifications.

While minimizing diameter is a good general goal for group communication, there are additional criteria that are important for enabling (near) real-time delivery for internet multicast applications. In [16], this problem is defined as a bicriteria problem, which can be stated as a minimization of two criteria: (1) *the average-latency* and (2) *the maximum-latency*, under degree constraints. This is a combination of DC-MAL and DC-MML.

Within a given tree T, the length $d_T(v)$ of the path from the root to any node v in the tree yields the latency required for a message to reach v. This tree distance $d_T(v)$ is simply the sum of the metric distances on the unique path to v. Hence, we refer to the average-latency based on these tree distances, and this average-latency is calculated as $\frac{1}{n}\sum_{v \in T} d_T(v)$. For the hop measurement of $depth_T(v)$, we use each node's level in the tree, with the root having a $depth = 1$.

We use these measures in our algorithms for degree-bounded spanning tree construction, in particular approximating the minimum average-latency in a multicast distribution tree while forming a compact tree (in terms of depth).

Calculation of a multicast tree in the context of the DC-MAL is NP-hard following from a reduction from the traveling repairman problem [12], which we show below. DC-MD is

shown to be NP-complete in the decision form by Shi et al. [60] via a reduction from the traveling salesman problem. According to published literature, the *minimum latency problem* is NP-Complete [12, 36, 56]. By Theorem 1.1, DC-MAL is NP-Complete.

Calculation of a tree using DC-MAL over the $\mathbb{R}^1$ can be done in polynomial time, thus the problem is not NP-Complete for this metric space. For out degree one, the $O(n^2)$ algorithm for solving the LINE-TRP problem can be used [12]. If the out-degree is larger, $B \geq 2$, then calculation of an optimal distribution tree can still be done in polynomial time by only making one out connection from each vertex, with two out connections from the root (one in each direction). Figure 1.3 shows an example tree in $\mathbb{R}^1$ where $B = 3$, edges are drawn off the metric line for illustration purposes only. It is an open problem whether or not an optimal solution to DC-MAL can be found in polynomial time in $\mathbb{R}^1$ where $B \geq 2$ and the tree is compact.

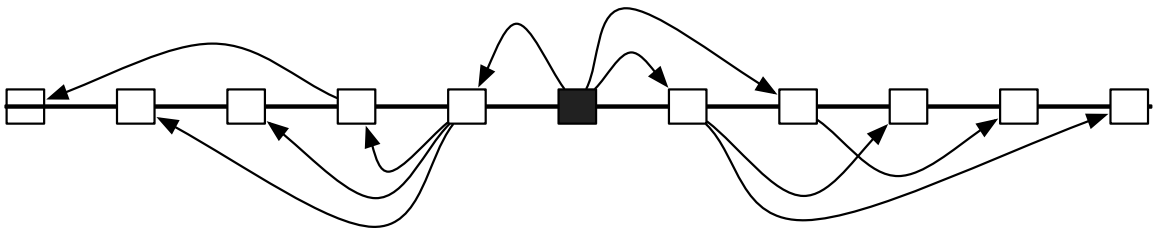


Figure 1.3: DC-MAL solution in $\mathbb{R}^1$, with out-degree $B = 3$.

Let the formal definition of the decision version of DC-MAL be: given a graph, $G = (V, E)$ where $n = |V|$, an out-degree bound $B > 0$, an average-latency measurement $= l$, and a root r: can a spanning tree rooted at r be built that obeys the out-degree bound and has an average-latency of l? We also use the definition of the traveling repairman problem (TRP) as defined in [12]. Let the definition of the decision version of TRP be: given a graph, $G = (V, E)$ where $n = |V|$, a starting location s: does a tour exist such that the total cost of the tour is t where $t = \sum_{v \in V} d_T(r, v)$? The general version of TRP is NP-Complete, as is the Euclidean version of TRP [12].

Theorem 1.1. *The decision version of the degree-constrained minimum average latency problem (DC-MAL) is NP-Complete. The problem remains NP-Complete over metric spaces, in particular over* $\mathbb{R}^2$.

Proof. DC-MAL is NP-Complete via reduction from the traveling repairman problem (TRP), TRP $\propto$ DC-MAL.

An instance of TRP can be directly transformed to an instance of DC-MAL, by using the same graph and specified root. The out-degree for DC-MAL is set to 1. The waiting time metric for TRP, t, is transformed to an average latency requirement such that $l = t/n$. This transformation is done in polynomial time.

It is immediate that there is a yes answer for DC-MAL *iff* the answer is yes for the instance of TRP.

□

1.2 Performance Metrics for Multicast Trees

In order to assist in computation and analytic comparison of multicast trees, we analyze algorithms using several classes of underlying metrics. There are also several standard notations that we use throughout this work in relation to multicast trees, they are presented here:

1. **input graph**: The completed weighted input graph G is defined as $G = (V, E)$, where V is the set of all vertices in the graph and E is the set of all edges in the graph.

2. **number of vertices**: The number of vertices in the graph n is simply the size of the vertex set $|V| = n$.

3. **out-degree bound**: We define the out-degree bound parameter as B where $B \geq 2$.

4. **multicast tree**: The multicast tree calculated from an input graph G is defined as T. It is the case that T is a subgraph of G, $T \subset E(G)$.

5. **root**: The root of the multicast tree is a specific vertex r where $r \in V(G)$

6. **metric distance**: The distance between any two vertices $u, v \in V$ is defined as $d_m(u, v)$. The actual metric function used is dependent on the underlying metric space.

7. **tree distance**: The distance between any two vertices $u, v \in V$ in the tree is defined as $d_T(u, v)$. This distance calculation is the sum of the edge weights in the tree T from an ancestor v to its descendent u. For shorthand, we represent the distance from the root in the tree as $d_T(v)$, an equivalent to $d_T(v, r)$.

8. **height**: The height of a node in the tree $h(v)$ is defined to be the depth of the node in the tree +1.

The performance metrics used are defined as:

1. **height/maximum depth**: The height of a multicast tree is simply a count of the number of levels in the tree. We begin counting with the root being of height 1. Given the constraints of a degree bound B, the *minimum* height of any such degree bound spanning tree is $\lceil log_B n \rceil$. The level, or depth, of an individual vertex $v \in V$ in the final tree T is denoted as $depth_T(v)$.

2. **average-depth**: The average depth of all vertices in the tree is a useful measure as to the perceived compactness of the tree and is calculated as $\frac{1}{n} \sum_{v \in T} depth_T(v)$.

3. **minimum possible depth**: The minimum possible maximum depth for any tree formed over the input graph based on $|V| = n$ and B is defined to be $\lceil \log_B n \rceil$ for all graphs.

4. **compactness**: The measure of the deviation from the minimum possible depth given the degree constraints on the input graph. A tree is *perfectly compact* if the *maximum depth* in the tree is the *minimum possible depth*. In literature, a perfectly compact tree is sometimes referred to as complete or full.

5. **latency**: The delivery latency for any individual vertex is the sum of the path weights from the source node to that node and is denoted by $d_T(r, v) \, \forall \, v \in V$.

6. **average-latency**: The average latency for all vertices in the tree provides a measurement of the mean time required for delivery and is highly sensitive to differences among trees. The average-latency of a tree T is calculated as $\frac{1}{n} \sum_{v \in T} d_T(v)$

7. **maximum latency:** The maximum time to deliver, $\forall v \in V \mid max(d_T(r, v))$

8. **average non-leaf load**: The average out-degree of non-leaf vertices in the tree.

9. **diameter:** The diameter of a tree is defined as the maximum point to point communication cost for any two vertices in the tree along the tree paths. The diameter may or may not pass through the root vertex. This measure is useful for designing group communication systems that are not based on single source transmission.

10. **diameter hops:** The number of edges along the path that determines the diameter of the tree.

Furthermore we consider the connection between these performance metrics and observe that there are tradeoffs in degree-bounded spanning tree formation, especially considering maximum depth and average-latency [41]. There is a meaningful relationship between these two metrics for multicast trees. Empirically, we have demonstrated that as the tree is reduced in depth, the average-latency measurement increases.

1.3 Network Modeling - Formal Framework

Our goal in this thesis is to study algorithmic solutions to the multicast problem in the context of realistic assumptions about underlying network models. There are two prevailing difficulties with computing solutions to this problem in any distributed system:

1. The defining and modeling of networks measurements

2. Computation and distribution of the multicast tree routes

Since the required input is a complete graph, network latencies must be measured, recorded and collected in order to perform the tree computation. If the calculation is done centralized, then there is a single point of failure and a potential bottleneck on the multicast route determination node. Once the tree is calculated, at a minimum, subtrees must be distributed so that message delivery can commence. These are, of course, technical challenges to implementing multicast solutions based on solutions to the minimum average-latency degree-bounded directed spanning tree problem.

A metric space representation is utilized as it closely models the underlying network structure and reduces the complexity required in developing theoretical results. Modeling network as a metric space and/or a graph model is a common practice in multicast tree research [16, 42, 60, 61]. In order to gather the metric space information we may rely on network segmentation such as found in [11, 16] in order to reduce the number of latency measurement pairs that need to be sampled. This would allow for tree calculations to be done on a network segment basis, reducing the amount of data needed and the time needed to produce a solution. More details are given in Chapter 5.

In this thesis we consider only metric spaces which obey the triangle inequality, defined as $d_m(a,b) \leq d_m(a,c)+d_m(c,b) \forall a,b,c \in V$ where $a \neq b \neq c$ and $d_m(a,b)$ is the distance between two vertices in the metric space. Such an assumption may not be completely accurate in

networks. However, the metric serves as a reasonable approximation to reality [16, 42, 60]. In addition, we consider a variety of classes of metric spaces to ensure integrity of the results.

The exact graph topology models used in experimental study are; (1) 2-dimensional random geometric, (2) transit-stub using the Georgia Tech Internetwork Topology Models (GT-ITM) generator [67], and (3) high dimensional Euclidean hypercube topologies using ℓ_1 and ℓ_2 as metric functions. Additional details of the graph topologies studied and used for experimentation can be found in Chapter 4.

In a practical, distributed network (such as the Internet), some of these assumptions give way to real world concerns. With increasing size, it becomes less possible to measure the metric space, and assign global coordinates to hosts on the Internet. Hosts on the Internet in general to not obey the triangle inequality. However, systems such as Vivaldi [27] have been used to assign coordinates based on round trip time (RTT) measurements with reasonable estimates.

The challenge then becomes not gathering the data or assigning coordinates, but rather we must focus on proper network segmentation to reduce the computational space for tree calculation. There are many approaches to hierarchical peer-to-peer, hierarchical multicast, and hybrid application level multicast, such as: XScribe [50], OMNI [16], SplitStream [21], Tulip [11], the work of Tran et al. [63], as well as other platforms not mentioned here.

For real world deployment we aim to blend existing approaches and leverage hybrid models. Since the possibility for group membership in general purpose information discrimination to become quite large, clusters may form naturally. For example play-by-play for the Olympic hockey finals may cluster naturally with concentrations of subscribers in the home countries of the teams. We believe that logical network segmentation and introduction of a hierarchical classification is a natural approach. A case study of the effects of multicasting and the degree-bounded spanning tree problem on real world systems is presented in Chapter 5.

1.4 Document Map

The remainder of this body of work details the *DC-MAL* problem, how this problem relates to multicasting, a new approximation algorithm as a solution, practical considerations for deployment, related work, and a discussion of experimental results.

A new solution to DC-MAL, the *Myriad* algorithm, is presented in Chapter 2, including pseudocode and analysis. In order to give context to this research, Chapter 3 details the related work used as both a basis and a benchmark for our theoretical and experimental advancements.

Details of the experimental methodology employed are given in Chapter 4, giving a full description of the test software developed and how experiments were conducted using high performance computing (HPC) clusters. Experimental results are presented and analyzed in the same Chapter, Section 4.5. In Chapter 5 we present a systems architecture view of how inclusion of *Myriad* could proceed for some existing P2P systems and discuss possible performance gains by doing so.

Final thoughts and forward looking statements are given in Chapter 6.

Chapter 2

Algorithmic Design of *Myriad*

Our algorithm for solving the multicast tree construction problem is the result of informed experimentation and analysis. The techniques used to construct trees and the arrived upon solution, the *Myriad* heuristic algorithm, are the result of research that exploits all available latency information in order to construct a multicast tree (a degree-bounded, rooted, directed spanning tree) of minimum average-latency.

The obvious solution to providing the minimum average-latency is to actually provide the minimum latency for each node requesting the multicast delivery. This is of course the well known shortest path calculation, resulting in a tree if the shortest path is taken from a specified root. Assuming that the triangle inequality holds for the metric space, we are likely to end up with a star topology (Figure 2.1), or we can coerce the network to a star topology by minor adjustments in the coordinates of vertices that are aligned on the same axis from the root. While at first glance this delivery tree might look acceptable, it is not. There are several issues at work here. First off, we have overloaded the source node as it must send the same message to each individual mode, this violates the principles of the degree-bounded spanning tree and is the exact problem that multicast transmission is designed to solve. All that has been done in this case is to minimize the path between the source and

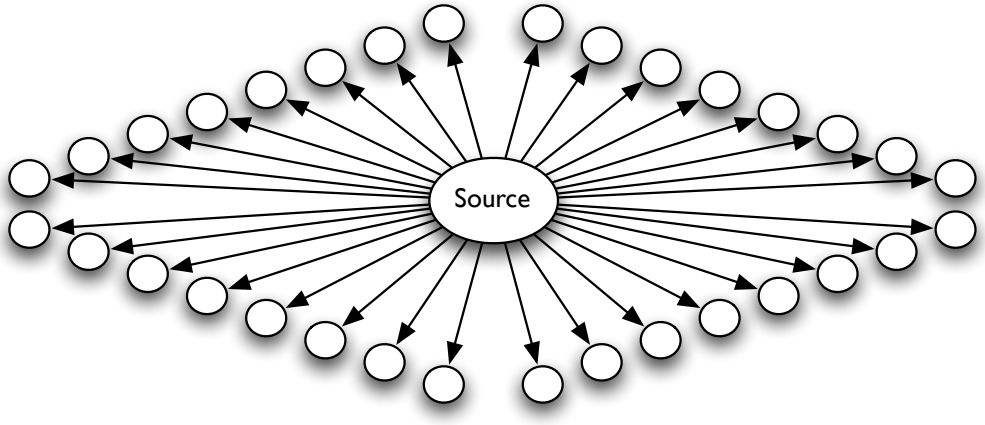


Figure 2.1: Example routing using shortest path calculation, and the resultant star topology.

all destinations, but other factors have not been considered. While the traversal time of the individual links is optimal, the overall delivery time and load will not be. The source node is likely to either not be able to provide information in a timely manner or stop functioning altogether under excessive load.

Our first iteration over a solution started instead with the minimum cost spanning tree using the latency as cost, and minimizing the sum of all traversals. In experimentation, the minimum spanning tree based algorithm does not hold up against other existing solutions due to longer paths being formed, and was later revised. It is possible that a minimum spanning tree based solution could be used to minimize overall cost where that is a primary metric.

In order to form a good solution to DC-MAL, we return to the shortest path tree as a starting point. This allows us to operate within an unconstrained framework where every vertex starts at an individual optimum in regards to minimum latency, and consequently minimum average-latency is also at an optimal value. Of course, the degree-bound constraints are violated (as shown in Figure 2.1) except where the out-degree constraint is greater than

or equal to the number of vertices in the graph minus 1, $B \geq |V| - 1$. Section 2.2 fully describes the *Myriad* algorithm and how it provides an effective solution to this problem.

Any algorithm that uses the shortest path tree as a starting point will be forced to move vertices from their individual optimums in the starting tree. By definition a vertex that is not moved from the starting shortest path tree placement is at its optimal delivery point for the multicast tree. The algorithm that produces the least amount of stretch when adjusting for degree bound is the algorithm that yields the minimum average-latency.

In order to assist in the discussion of algorithm design, we present some *design ingredients* that can be selected and applied in the construction of algorithms. These ingredients became apparent through the study of *DC-MAL* and the development of *Myriad*.

1. Shortest path tree as a starting point
2. Minimum spanning tree as a starting point
3. Clustering, using Euclidean distance
4. Restricting connections to parents nearer to the root
5. Dynamic optimization, by repeatedly modifying the tree, a hill climbing strategy
6. Compactness assurance

Each of these ingredients has a place in the discussion on these algorithms, and we will use this list to annotate which ingredients an algorithm utilizes. In order to justify the inclusion of each ingredient we present the motivation of each ingredient.

1. *Shortest path tree as a starting point*: The shortest path tree is used as a starting point when latency is a primary optimization factor in the network design. It is possible that for some networks the shortest path tree does not violate out-degree constraint and is then, by definition, the optimal solution.

2. *Minimum spanning tree as a starting point*: The minimum spanning tree is a useful starting point if the primary goal is total cost minimization. From this starting point adjustments can be made to improve things like latency and out-degree bound while seeking to remain within a budget for total edge cost. None of the algorithms presented in this work use this ingredient, although the precursors to *Myriad* operated with the minimum spanning tree as a starting point.

3. *Clustering via Euclidian distance*: Some algorithms make use of clustering of vertices. This technique is used when there is good reason to group a subsection of the vertices in the graph.

4. *Restricting connections to parents nearer to the root*: An ingredient introduced by *Myriad* under the observation that connecting through vertices not nearer to the root does not yield better results in the general case. This property is used in the proof of Theorem 2.1, and gives theoretical bounds and performance.

5. *Dynamic optimization*: Dynamic optimization is a technique whereby an initial tree is calculated and later optimized as the graph dynamically changes, and thus what constitutes an efficient tree will change as well.

6. *Compactness*: Does the algorithm make a goal of producing a tree of minimum required height, a compact tree?

Of these design ingredients, *Myriad* uses three ingredients: the shortest path tree as a starting point, the restricting of connections to parents nearer to the root, and compactness. Our experiments also make use of clustering by way of incorporating *Myriad* into the KLS algorithm [42], but clustering is not essential to the design of *Myriad*. During the process of algorithm development, experiments have been conducted using the minimum spanning tree as starting point. This design ingredient has been found to be most useful in producing

minimum total cost degree-bounded spanning trees where cost is a metric function tied to the sum of edge weights. Of the algorithms studied, only OMNI [16] makes use of dynamic optimization for multicasting.

2.1 Problem Constraints

The principal constraint placed on this problem is that of the degree bound. For simplicity we refer only to the out-degree bound B since all hosts (besides the root) have an in-degree of exactly 1, allowing us to ignore this constant. There are several motivations for enforcing an out-degree constraint in the multicast application space derived from bandwidth considerations.

A second constraint placed on solutions to this problem is that of tree height, which manifests itself in several ways. We can consider either the observed measures of actual layers in the tree, or the measurement of root to node path lengths (latency). It seems that each of these measurements plays an important role and we must be mindful of both. In many cases, there appear to be benefits to producing a tree of suboptimal height [41], but we do not want to take this to an extreme (force out-degree one, a linear distribution network) as it has an undesirable impact on average-latency.

Tran et al. [63] provide arguments for both of these constraints. In order to minimize end-to-end delay, a tree should be kept short, minimizing height. This reduces the number of paths between the source and all destinations, thus reducing the possible points of failure, and potentially reducing the sum of the path lengths (although this is largely depending on the length of the individual paths used). Along the same lines, imposing a degree-bound limits the bottleneck effects at any given host and also reduces potential effects of a node failure. We believe that degree-bound is also a necessary constraint for bandwidth, as supported by the maximum degree of any end host [44].

Under these constraints we seek to build a multicast tree that does not go over the degree-bound constraints, and also provides minimum stretch (distortion) in relation to average-latency for connecting end systems.

2.2 Foundations for *Myriad*

Here we present a new solution to the degree-bounded minimum average-latency spanning tree problem. Our new algorithm, *Myriad* uses a heuristic to form degree-bounded spanning trees by ensuring that all hosts connect to the source by connecting through a host that is closer to the root. This helps to reduce latency introduced by deviations from the optimal path.

We have discovered that forcing the upper levels of the tree to be full, but not forcing a tree of overall minimum height, we are able to achieve better experimental results in terms of average-latency measurements. *Myriad* concentrates on producing a minimum average-latency without forcing a tree of minimum height, and has an optimization step that optionally derives a tree of low average-latency while guaranteeing a tree of minimum height ($\lceil \log_B n \rceil$ levels). We examined the effects of producing a compact tree over clusters, producing degree-bounded spanning trees without regard to hop count, and enforcing a strict bound on the maximum hop count.

2.3 *Myriad*

The *Myriad* algorithm for approximating degree-bounded spanning trees begins by calculating the shortest path tree from the root. The shortest path tree is then modified using the heuristic whereby we state that each node must be connected to a parent that is closer to the root than itself, that is, if a's parent is b, then $d_M(b, root) \leq d_M(a, root)$. Visually, this

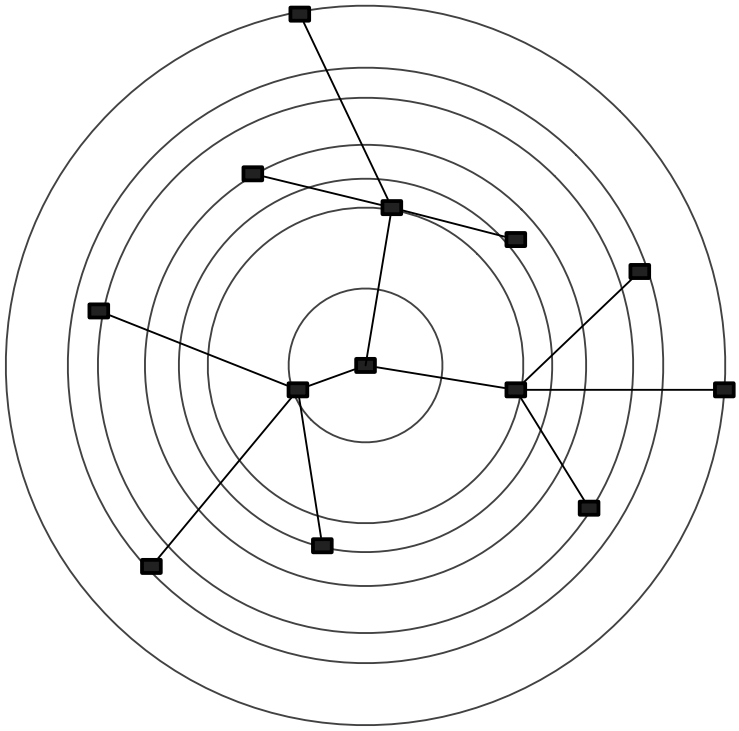


Figure 2.2: Example circular layout, with each node connected to the root through a node closer to the root than itself.

is represented by concentric circles which are drawn for each radius of a node from the root, and all nodes are connected to a parent in a circle of equal to, or small radius than the ring they are a member of (see Figure 2.2). Pseudocode for *Myriad* is shown as Algorithm 1 on Page 39, we guide the reader through this pseudocode later. *Myriad*, as described here, is a centralized algorithm. We provide some thoughts on how this might be implemented as a distributed algorithm in Chapter 6.

Throughout the process, nodes that are over their degree bound are relieved of this load by clearing the current search space (current level) and forcing the load one level lower. While this guarantees that the current level contains no nodes above the maximum out-degree, nodes in the next level may now have this characterization and need to be cleared of extra load.

Initially, all nodes in the shortest path tree will either be connected directly to the root, or have a path to the root that connects only through nodes closer to the root (if they are in the same axis from the root in the ℓ_1 metric space). For both the random distribution and hypercube models studied, the shortest path tree is a simple 2 level star topology instead of the calculated shortest path tree. Forcing the calculation of the star topology, rather than computing the shortest path tree provides a reduction in computational complexity.

The relationship of each node always connecting to the root through nodes closer to the root is kept in effect by correcting the tree one level at a time. For example, starting at the root (level 1), the closest B children at each vertex are kept as children, while the remaining children (up to $n - B - 1$) are made children of the closest B children. The process is then repeated until all nodes in the tree have $\leq B$ children at all levels.

There are scenarios that arise where the B closest children to a vertex are not the B children that are also closest to the root as demonstrated in Figure 2.3. This scenario does arise in some graphs due to the relaxed rules of the heuristic. In this example, node Y would normally be the third child of node X, and node Z would be made a grandchild of X. However, this would violate the organizational rule stating that each vertex be connected to a vertex that is closer to the root than itself. This is solved by swapping the tree position of Y and Z. Node Z is made a child of X, and Y is made a grandchild X and a child of Z.

2.3.1 PostOptimization

The *Myriad* algorithm has an extension called the *PostOptimization* step that, when run, can either compact the tree only when improvement occurs, or force the tree to be of minimum possible height as determined by the out-degree bound and number of vertices in the input graph ($\lceil \log_B n \rceil$). After the base *Myriad* algorithm has completed, the tree is compacted by examining the tree in breadth-first search order, starting with the root. Any vertex encountered that has an out-degree $< B$ is brought up to an out-degree of B by promoting

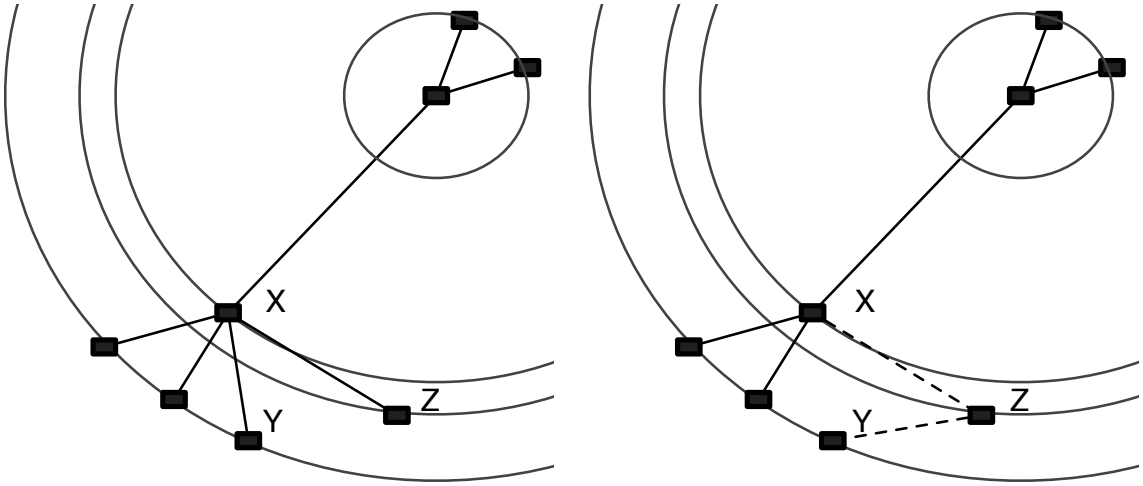


Figure 2.3: Example correction, when a node's B closest children are not the 3 closest nodes to the root (with a maximum out-degree of 3).

a sub-tree. If the node being examined is at level i, sub-trees rooted in levels $\geq i + 2$ are considered potential children. The promotion which causes the smallest increase to average-latency is the transfer enacted in the current tree. This procedure guarantees a final tree of minimum height, at the expense of increased average-latency. The procedure for for the post optimization step is presented as Algorithm 2 on Page 40.

2.3.2 Pseudocode Guide

Here, we provide a guide to the reader to assist with the understanding of the details of how *Myriad* operates and provide sufficient information for new implementations of the algorithm. These guides refer to the pseudocode as listed in Algorithms 1, 2, and 3.

Guide for `Myriad`

Since the implementation of *Myriad* is largely based on the manipulation of sets, we give a quick reference list of what each set means as used in Algorithm 1, Page 39).

- `overloadedSet` - The current set of all vertices $v \in V$ whose out-degree in the overlay tree is larger than B. This set is first introduced on line 2 and is updated on each pass through the loop.

- `curLevelSet` - A set of all vertices that are in the current level of the tree being processed. These vertices will be inspected for being over the out degree bound, but will not be moved from the current level. This set is introduced on line 5 and is reinitialized for each level as processing progresses.

- `fixed` - The `fixed` set is the set of vertices that are children of the nodes in the `curLevelSet`. A node is added to the `fixed` set if it is one of the closest B children of their parent.

- `toMove` - If a child of a node in the `curLevelSet` can't be put in the `fixed` set, then it is placed in the `toMove` set.

The input to *Myriad* is a complete graph or subsection of a graph, annotated as the set V. In our implementation each vertex is aware of its out edges, their weight, and has the ability to get a reference to the connecting vertex. Our graph implementation allows for easy calculation of an individual vertex's parent and separation of overlay versus non-overlay edges. The r parameter to the algorithm is the source of the communication and the root of the result tree. B defines a uniform out-degree constraint for the tree, however, in our implementation the out-degree is vertex specific and is part of the vertex data structure.

Myriad (Algorithm 1, Page 39) begins by calculating the shortest path tree, rooted at r, over the vertex set V. Dijkstra's algorithm for shortest path calculation is used [31]. Because of how our graph abstract data type is implemented, this information is easily stored at a vertex level and retained for further calculation. Our implementation also includes an option for forcing a 2-level start topology and forgoing the more complex shortest path calculation.

On line 2 (and later 24) a call is made to the `findOverloadedVertices` method. This method performs a breadth-first search on the tree and returns a set of vertices that currently have an out-degree $> b$. Execution of this algorithm progresses until the `overloadedSet` is empty. Since organization of the tree proceeds form the top level down, the `level` variable is initialized to 1, since this is a 1 based numbering of tree levels.

In order to assist in implementation speed, each vertex is also assigned an integer that represents its current depth in the tree. The `nodesInLevel` method on line 5 makes use of this fact, although this could just as easily be accomplished with a depth-first tree traversal. By isolating nodes in the current level to the `curLevelSet`, we are isolating the particular vertices that may need to be relieved of excessive out-degree. If there is no intersection between the `curLevelSet` and the `overloadedSet` (line 6), we simply move down the tree to the next level and recalculate the `overloadedSet` (line 24).

Assuming there is in intersection of the `curLevelSet` and the `overloadedSet` on line 6, we begin processing the current `level`. Two sets, `fixed` and `toMove`, are initialized and kept empty. These sets will be used to hold vertices which should not be moved from their current position unless absolutely necessary as part of a swap (line 17) and vertices that must be moved to correct out-degree constraints respectively.

On line 8, the main processing loop for the current level begins. The goal here is not to immediately put all vertices in their best location, but rather to select the best possible vertices to be in the next level (`level + 1`). Each vertex in the `curLevelSet` is processed, and order is unimportant. If a vertex currently has a degree bound higher than B (line 9), the closest B children of that vertex are added to the `fixed` set. This makes them candidates for accepting overloaded vertices from the `toMove` set. All other children of the current vertex being examined are placed in the `toMove` set which will force them to (1) stay in the same level and move to a different parent or (2) move 1 level lower in the tree. If a

vertex in the current level is below the out-degree constraint of B, that vertex is also added to the `fixed` set, allowing it to gain children during the next phase.

After the current level has been processed, we move on to find a new location in the tree for vertices that have been placed in the `toMove` set (loop starting at line 15). For each vertex in the `toMove` set, we assign it to the closest vertex in the `fixed` set. If a situation occurs that the vertex can not be assigned (line 17), as shown in Figure 2.3, a swap occurs in order to keep the rules of *Myriad* in place.

Myriad terminates when no overloaded vertices remain in the tree. At this point, there is no guarantee that the tree will be of minimum height, for this the `PostOptimize` step is needed.

Guide for `PostOptimize`

The `PostOptimize` (Algorithm 2, Page 40) step is always run after *Myriad* has constructed a tree. This method takes in a multicast tree T, the root of that tree r, and the out-degree B. The fourth parameter, `forceCompact`, is a boolean value that indicates if the tree should be reduced in height only when improvements in average-latency are found (when false) or if the tree should be reduced in height until the tree is at minimum height. Even when a tree of minimum height is produced, there is no guarantee that leaves will occur only in the last two levels of the tree.

The outer loop of the algorithm executes once for each level in the tree, starting with level 1, the root. Since the tree is compacted as the loop executes, we can be guaranteed that the loop will only execute at most $\lceil log_B b \rceil$ times, the minimum required height, as noted in the invariant on line 2. Based on the current level numbering processed we identify a set of potential vertices to consider for relocation. The inner loop then repeats for each vertex in the current level, in breadth-first search order where edges are searched in ascending weight order.

For each vertex, a candidate move set is initialized and inspected to find the best move. The procedure for determining the candidate move set is Algorithm 3 and is detailed in the next section. We start by assuming that there is not a better move that can be made (line 6) and that any average-latency improvement must be positive in magnitude (line 7). If a compact tree is desired, we simply adjust the minimum improvement threshold to be $-\inf$, allowing for a negative "improvment" to average-latency.

We now examine each vertex in the `movers` set (line 9) to determine the effects of relocating each vertex to be a child of the current parent being inspected by the loop on line 3. The calculations from line 12 through 17 calculate the current average-latency in the subtree rooted at m (current `movers` vertex) in its current position and what the average-latency in the subtree would be if m was pulled up. If this improvement is the best improvement known on this pass, it is saved. After the loop, the best improvement to average-latency, or lowest increase to average-latency for a compact tree, the vertex change is made (lines 27-28).

Guide for `getCandidateMoveSet`

In order to identify which vertices might be used to compact the tree are identified by the `getCandidateMoveSet` procedure (Algorithm 3, Page 41). This process starts by identifying vertices that are currently two levels below the current level being processed, with the tree T, root T, and current *level* being inputs to the method. If a compact tree is desired, the search space is extended to include vertices that are three levels below the current level (lines 2 and 3), otherwise the search space is expanded to include all vertices. This is done since a non-compact tree can focus solely on looking for the best improvement for average-latency, while the compact version need to focus on pulling up a larger subtree.

2.3.3 Sample Run of Myriad

Here, we provide a step by step example of how *Myriad* organizes vertices in a graph. The first step is to calculate the shortest path tree, which is the starting point for *Myriad*. Figure 2.4 shows a sample graph after the shortest path tree has been calculated.

Progressively each level is examined for vertices that are overloaded, and those vertices are moved to be children of vertices in the `fixed` set. These vertices might be other members of the same level or moved to a lower level in the tree. On the next three pages, a graphical representation of the execution of `Myriad is shown`.

- Figure 2.4 - Starting tree, the shortest path tree calculation
- Figure 2.5 - Level 1 (the root) is cleared of excessive load
- Figure 2.6 - Level 2 is cleared of excessive load
- Figure 2.7 - Level 3 is cleared of excessive load

After level 3 is cleared of excessive load, there are no more overloaded vertices in the graph, so the algorithm terminates.

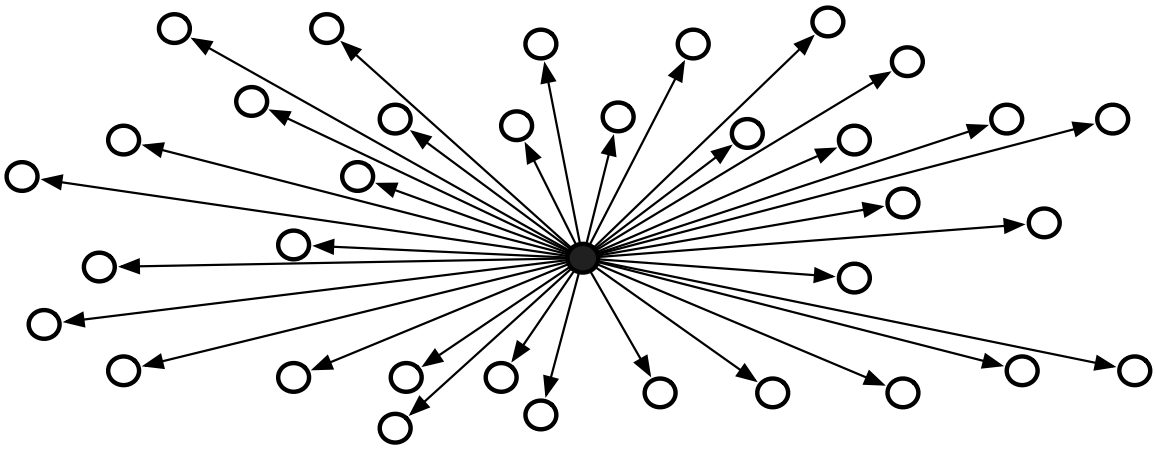


Figure 2.4: Myriad step 1 - the shortest path tree from the root is calculated.

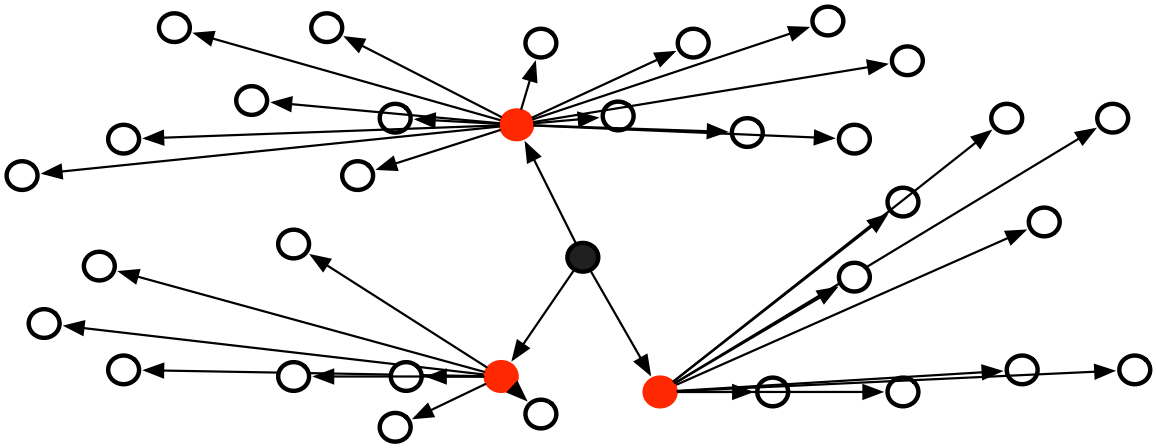


Figure 2.5: Myriad step 2 - Level 1 (the root) is adjusted to obey the out-degree bound.

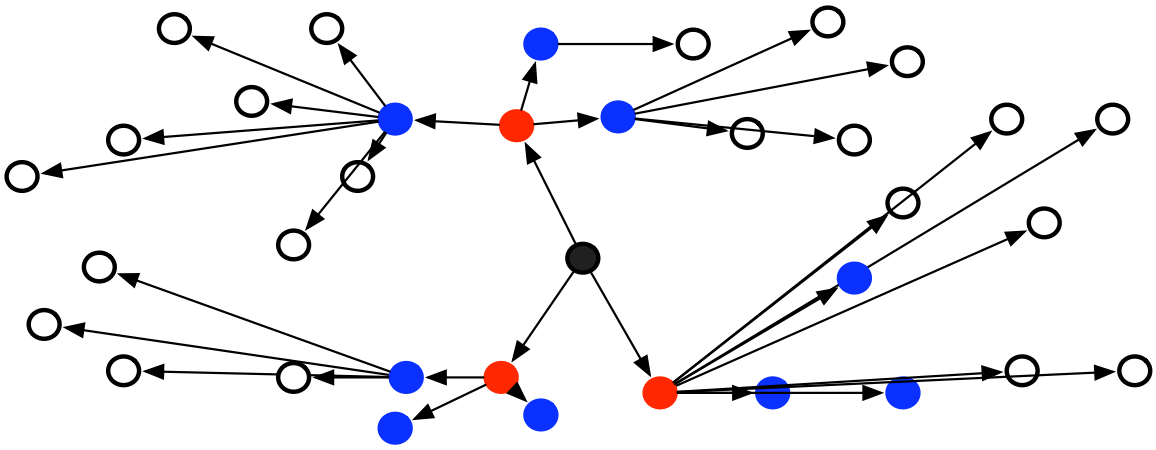


Figure 2.6: Myriad step 3 - Nodes in level 2 (children of the root) are cleared of excessive load.

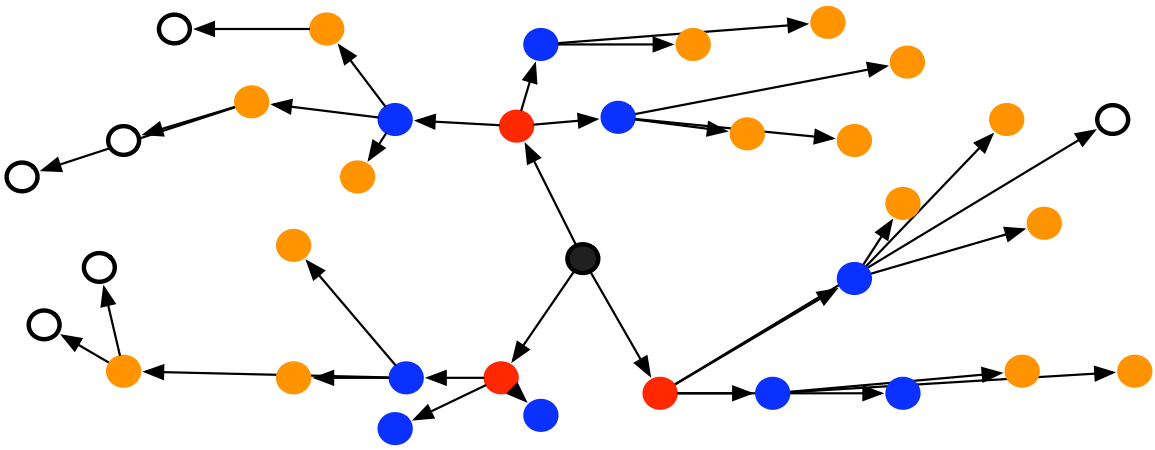


Figure 2.7: Myriad step 4 - Nodes in level 3 are cleared of excessive load.

2.3.4 Stretch

Stretch is defined as the ratio of distance to the root in the constructed tree over the shortest path distances, or, the optimal placements in the tree. For graphs embedded in a metric space obeying the triangle inequality, the optimal placement in any tree that does not obey out-degree constraints is the same as the vertex placements along the shortest paths. In this case, maximum stretch bounds the distortion incurred by running *Myriad*. We have also shown a new upper bound for distortion in hypercube networks (below).

Stretch in *Myriad*

By Theorem 2.1, it follows that the maximum stretch for *any vertex* in a tree produced by *Myriad* is bounded by the depth of that vertex in the tree. Let r represent the root of T, $d_m(v, r)$ represents the metric distance from v to r, $d_T(v, p)$ represent distance in the tree to v from an ancestor p, $h(v, p)$ represent the depth of v in the tree from p, and let k substitute for $h(v, p)$

Theorem 2.1. *Myriad produces a tree $T = (V, E)$, with maximum out-degree B, such that the maximum stretch for any vertex $v \in V$ is at most $\frac{d_T(v,r)}{d_m(v,r)} \leq 2(k+1) = 2 * maxdepth$ where k is the level in the tree, or $O(h(v, r))$. This theorem holds for all metric spaces that obey the triangle inequality.*

Proof. By definition, the root r is under no stretch.

By definition, the direct children of the root r are under no stretch, and connect with their minimum distance, $d_m(v, r)$, via direct connection.

For all other vertices, $v \in V$, their connection (by definition of the algorithm) must connect to the root through a parent, p, that is closer to the root in the metric space (i.e., $d_m(v, r) \geq d_m(p, r)$). The tree distance $d_T(v, r) = d_T(v, p) + d_T(p, r)$. Let v be at level $K > 1$ of the tree. We have the following;

$$
\begin{aligned}
d_T(v,r) &= d_T(v,p) + d_T(p,r) && (2.1)\\
&\leq d_T(v,p) + 2 * k * d_m(p,r) && (2.2)\\
&\leq 2 * d_m(v,r) + 2 * k * d_m(p,r) && (2.3)\\
&\leq 2 * d_m(v,r) + 2 * k * d_m(v,r) && (2.4)\\
&\leq (2k+2) * d_m(v,r) && (2.5)\\
&\leq 2(k+1) * d_m(v,r) && (2.6)
\end{aligned}
$$

Equation 2.2 follows from 2.1 since any vertex in the tree must connect to the root through a parent p (and thus all ancestors) closer to the root. The maximum stretch for vertex p is 2 times the depth in the tree and the distance of that vertex in the metric space by inductive hypothesis.

Equation 2.3 follows by the heuristic and the triangle inequality.

Equation 2.4 follows from the inequality $d_m(v,r) \leq d_m(p,r)$. This must be true for v to be a child of p, otherwise the two vertices would see a role reversal by the rules of *Myriad*.

Equations 2.5 and 2.6 follow by the factoring of terms.

Thus the maximum stretch for a vertex is bound by $\leq 2(k+1) = 2 * h(v,r)$. □

Conjecture 2.1. Following from Theorem 2.1, if a compact tree can be formed using the *Myriad* heuristic and *Myriad* produces a perfectly compact tree, the maximum depth of a vertex v is bound by $\lceil \log_B(|V|) \rceil$. It follows that the maximum stretch in a tree produced by *Myriad* for any vertex $v \in V$ is $2 * \lceil \log_B(|V|) \rceil \Rightarrow O(\log_B(|V|))$.

We label Conjecture 2.1 as such because in order to facilitate better compact trees we allow the compaction step, when a minimum height tree is desired, to deviate from the heuristic of always connecting to a parent closer to the root. These deviations are only taken

in the event that average-latency for a subtree is improved by breaking the rule of always connecting closer to the root. We have experimented with a variant of *Myriad* that strictly enforces the connection heuristic, and by those experiments believe the above conjecture to be true.

Distortion in Hypercubes

Given a discrete metric space M, the *T-distortion* induced by a rooted spanning tree T is simply the ratio of the average-latency (equivalently, total distance) to the root in the tree T divided by the average shortest distance to the root given by the metric M. Define the T_B-distortion of a metric M as the minimum value of the T-distortion over all B-ary spanning trees (and thus satisfying a degree bound of $B+1$). For the hypercube metrics we have the following lower bound results on the T_B-distortion of $H^p_{2^n}$. The $H^p_{2^n}$ hypercube is defined such that the ℓ_p metric function is used and there are 2^n nodes (n bits of representation).

Theorem 2.2. *The T_B-distortion of the metric $H^p_{2^n}$ is $\Omega(n^{1-\frac{1}{p}}/lnB)$.*

Proof. A lower bound for the T_B-distortion is obtained by calculating the ratio of the sum of minimum possible path lengths in any B-ary spanning tree divided by the sum of the shortest distances to the root node in the metric. In any B-ary spanning tree of the metric H^p_n, the minimum possible average hop-distance to the root is at least $log_B 2^n - 2$. Also, the average distance in the metric is seen to be at least $\frac{1}{2}(n/2)^{\frac{1}{p}}$. This follows since at least half the nodes of the hypercube are (metric) distance $(n/2)^{\frac{1}{p}}$ from the root node. Thus the T_B-distortion is at least $cn^{1-\frac{1}{p}}/lnB$ for some constant $c > 0$; and the theorem follows. □

The result above does not provide a meaningful bound for the T_B-distortion for the usual hamming metric, $p = 1$. In fact, for $B = 2$ the well known doubly-rooted binary tree embedding in $H^1_{2^n}$ (see [46]) shows that the T_2-distortion of $H^1_{2^n}$ is $\Theta(1)$. It is not difficult

to show that this $\Theta(1)$ result extends for all B that are powers of 2. We leave as an open problem whether or not this result extends for other values of B.

2.4 Analysis of *Myriad*

Myriad (Algorithm 1) as presented, begins with a straight forward implementation of Dijkstra's algorithm for shortest path calculation, with a running time of $O(|V|^2)$. If we modify the shortest path calculation to simply return a strict star topology (2 level), we can reduce this step to an $O(n)$ operation. In practice, this simplification provides the same final result for graphs in $\mathbb{R}^2_n$. For experimentation, we use advance knowledge of the topology to determine if Dijkstra's algorithm or a simple star topology algorithm should be run.

We make the assumption that the points in the cluster are randomly distributed and are not along the same axis from the root. We can solve the problem of two nodes on the same axis by nudging one, forcing a 2 level star topology. This allows us to focus on non-degenerative cases for analysis.

The outer loop of the algorithm repeats until the overloaded set is emptied. Each pass through the loop, one level is cleared of overloaded nodes, by overloading nodes in the next level, requiring $\Omega(\log_B n)$ iterations if we force the tree to be balanced or compact in this step. Since we do not force the tree to be balanced or compact, the worst case for this loop is n. We can be guaranteed that there will not be n iterations, otherwise there would be no vertices over the out-degree constraint. On average, for randomly distributed points in $\mathbb{R}^2_n$, this loop will evaluate on the order of $O(n)$, while in practice executing fewer times..

The first inner loop (line 8) executes once for each vertex in the current level, this is a fixed quantity that grows within known limits for each level, and is represented by the summation $\sum_{l=1}^{log_B n} b^l = \frac{b(n-1)}{b-1}$. While this quantity decreases, it is still on the order of $O(n)$. Inside of this loop, a heap is built for the ordering and selection of vertices with this operation

having a runtime of $O(ln(n))$. The second inner loop (line 15) executes once for each possible vertex in the move set. This is bound by the number of vertices below the current level, $\sum_{l=1}^{log_B n} n - b^l = n * log_B n - \frac{n-1}{B-1} - n + 1$. The dominating factor of the inner loops is the first with the internal heap structure, giving a total running time of $O(n * ln(n))$

In the worst case, considering the entire *Myriad* algorithm, the average running time of the algorithm is in the class $O(n^2 * ln(n))$, assuming that the 2 level star topology is used as a starting calculation.

2.5 Clustering And Filtering

We begin by examining the KLS algorithm [42], which uses node clustering to segment the network, internal cluster organization, and then meta cluster organization for a final result. We examine the effects of using the *Myriad* as the internal cluster organization method for KLS, as well as comparing the algorithms individually, and against other algorithms in the area (experimental methodology and results are presented in Chapter 4).

The KLS algorithm is an approximation algorithm for solving the *degree-bounded minimum diameter spanning tree problem*, and begins by forming clusters of nodes through filtering. The algorithm proceeds by performing a binary search to determine the best value for Δ, the estimated diameter of the degree-bounded spanning tree. Cluster representatives are then chosen iteratively by selecting the vertex that covers the most uncovered vertices within a radius of 3α, where $\alpha = \frac{\Delta}{\sqrt{\log_B n}}$. The α threshold represents the difference between long and short edges in the graph, this allows a limit to be imposed on the number of short edges and long edges in the final product, giving a bound of $O(\sqrt{\log_B n}\,\Delta)$ on the diameter of the produced degree-bounded spanning tree [42]. The final approximation tree is composed of clusters containing only short edges, joined together by long edges.

We examine the effects of changing how the clusters of short edges are organized, provid-

ing a significant improvement in runtime result according to the criteria set forth in Section 1.1. In their proof, Kanemann et al. [42] state that the cluster local structure must be a spanning tree of the nodes belonging to the cluster of minimum height. This relies on the fact that any edge used to join nodes within a local cluster will not be longer then $2r$ (where r is the radius of the cluster, 3α) as long as the triangle inequality is in effect. Theorem 2.3 states the upper bound on distance from the root, for any node in a local cluster.

Theorem 2.3. *Given any complete metric M and a root node r, let $d(v) = d_M(v, r)$ be the distance from v to r in M. There is a compact B-ary spanning tree such that every node v has a tree distance $d_T(v)$ to the root that is bounded by $2 * depth_T(v) * d(v) = O(\log_B n) * d(v)$.*

Proof. It is always possible to construct a B-ary spanning tree with the property that each node's parent is no further from the root than the node itself. This can be done by simply sorting the nodes by their distance to the root and creating a compact B-ary tree T whose breadth first traversal respects this ordering. The proof will follow with compaction of the depth of nodes. The root r clearly satisfies the statement of the Theorem. Now let us consider $v \neq r$. Let us consider the distance $d_T(v)$. We have that $d_T(v) \leq d_M(v, u) + d_T(u) \leq d_M(v, u) + 2\, depth(u) * d(u)$ using the inductive hypothesis, where u is the parent of v. By the triangle inequality we have that $d_M(v, u) \leq d_M(v, r) + d_M(r, u)$. A direct calculation shows that $d_T(v) \leq 2 * depth_T(v) * d(v)$ and the result follows by induction. Again, note that compact B-ary trees formed on n nodes will have depth at most $\lceil \log_B n \rceil$. □

This allows us to state that the total number of edges that contribute to the diameter of a cluster is bounded by $O(\log_B n)$. We have studied various methods for organizing the clusters in the KLS algorithm and examine the cost of building a tree that minimizes average-latency versus building a tree that minimizes the maximum depth for leaves in the tree. We have measured the performance of our *Myriad* algorithm (with and without forcing a compact tree) in the context of KLS clusters and on the entire network, without clustering. While

clustering and filtering provides for lower computational complexity and aids in the ability to produce a distributed algorithm, the resultant trees do not exhibit lower average-latency when compared to the non-clustering version of *Myriad* in both its compact and non-compact forms.

Let us first consider a system for naive construction of trees obeying Theorem 2.3. An algorithm can be constructed that uses the fact that no matter which edges are used in the construction of the local tree, the maximum distance from the root will still be $O(r\lceil \log_B n \rceil)$ since the tree is compact. In this case, the tree is built by adding nodes to the tree in the order of their distance from the cluster representative (breadth-first search order). Consequently, this organization algorithm is runtime efficient, forming an out-degree B tree of minimum height with a runtime of $O(n)$. However, the average-latency of trees formed through this method are not as efficient as they can be. The resultant trees do fall within the mathematical bounds set forth above and in [42] for local clusters and feed into the larger problem when the clusters are combined to form a final tree.

Algorithm 1 `Myriad(` V`,` r`,` B `)`: Organize the vertices V into a tree with root r and maximum out-degree B by optimizing the shortest path tree and enforcing that each node has a parent closer to the root than itself.

```
1: shortestPathTree( V, r )
2: overloadedSet ← findOverloadedVertices( V, r )
3: level ← 1
4: while overloadedSet ≠ ∅ do
5:    curLevelSet ← nodesInLevel( V, r, level )
6:    if overloadedSet ∩ curLevelSet ≠ ∅ then
7:       Set fixed, toMove ← new Set
8:       for all currentVertex ∈ curLevelSet do
9:          if currentVertex ∈ overloadedSet then
10:            The closest B children of currentVertex are added to the fixed set, the rest are
               added to the toMove set.
11:         else if currentVertex.outDegree < B then
12:            fixed.add( currentVertex )
13:         end if
14:      end for
15:      for all moveVertex ∈ toMove do
16:         Connect moveVertex to the closest vertex in the fixed set that is also closer to the
            root.
17:         if moveVertex can not be moved then
18:            Swap moveVertex with a sibling that is farther from the root than moveVertex.
19:         end if
20:      end for
21:   else
22:      level ← level + 1
23:   end if
24:   overloadedSet ← findOverloadedVertices( V, r )
25: end while
```

Algorithm 2 `PostOptimize(` T, r, B, $forceCompact$ `)`: Take a previously calculated multicast tree T (output from *Myriad*, Algorithm 1) and force the tree to be of compact height, defined as $\lceil \log_B n \rceil$, where n is the number of vertices in the original graph, and B is the maximum out-degree of each vertex. $forceCompact$ is a boolean that indicates if the algorithm must run until the tree is compact, otherwise the post optimize step will only make changes that reduce the average-latency of T.

```
1: for all level ∈ T, starting with the root, level 1 do
2:   // INVARIANT: No matter how unbalanced the input tree is, this loop will only
     execute up to the maximum number of required levels in the tree.
3:   for all vertex V ∈ the current level of T in breadth-first search order do
4:     movers ← getCandidateMoveSet( T, r, level)
5:     while out degree of V is less than B do
6:       moveImprovedVertex ← nil
7:       improvement ← 0
8:       improvement ← −1000000 if forceCompact
9:       for all vertex m ∈ movers do
10:        if m.parent ≠ V then
11:          // Here we are seeing if we can improve the average subtree latency, or the
             minimum increase to average subree latency at m
12:          averageSTL ← averageSubtreLatencyFrom( m )
13:          toLRoot ← pathWeightToRoot( m )
14:          totalSTL ← averageSTL ∗ m.subtreeSize + toLRoot ∗ m.subtreeSize
15:          // what it could be
16:          potentialToRoot ← pathWeightToRoot( V ) + V.egeTo(m).weight
17:          newSTL ← averageSTL∗m.subtreeSize+potentialToRoot∗m.subtreeSize
18:          if potentialToRoot < totalSTL or forceCompact then
19:            diff ← totalSTL − newSTL ∗ m.subtreeSize
20:            if diff > improvement then
21:              improvement ← diff
22:              moveImproved ← mover
23:            end if
24:          end if
25:        end if
26:      end for
27:      if moveImproved ≠ nil then
28:        make mostImproved a child of V
29:      else
30:        break // can't improve anything else at V
31:      end if
32:    end while
33:  end for
34: end for
```

Algorithm 3 `getCandidateMoveSet(` T, r, $level$, $forceCompact$ `)` : A sub-procedure of the `PostOptimize` function. Based on the level being processed, returns a field of candidate vertices for pulling up, starting with nodes at least 2 levels below the one being processed.

```
1: movers ← getVerticesAtLevel( T, r, level + 2 )
2: if forceCompact then
3:    movers → movers∪ getVerticesAtLevel( T, r, level + 3 )
4: else
5:    inc ← 2
6:    field ← getVerticesAtLevel( T, r, level + inc )
7:    while field ≠ ∅ do
8:       movers ← movers ∪ field
9:       inc ← inc + 1
10:      field ← getVerticesAtLevel( T, r, level + inc )
11:   end while
12: end if
```

Chapter 3

Related Work

Here we give specific focus to related work pertaining to the algorithms that are studied experimentally when compared to *Myriad*. Additional related work pertaining to the systems level and applications of algorithms such as *Myriad* can be found in Chapter 5.

The three algorithms that we focus on are:

- KLS [42]
- MDDBST [60]
- Overlay Multicast Network Infrastructure (OMNI) [16].

Details of each of these algorithms are given in relation to their original description and how they relate compare *Myriad* and potential hybrid solutions.

Multicast distribution routes can be represented by a rooted, directed spanning tree. These trees can be imposed by the physical network structure as in IP multicasting, or derived by an algorithm in the context of a logical network structure. The problem of constructing such a rooted tree which covers all subscribers is complicated by the need to balance network resources while optimizing the servicing of the communication group. In order to enable real-time or near real-time delivery for bandwidth intensive multicast

communication streams (such as a live video broadcast), the multicast distribution tree must have a bounded out-degree in order to allow for smooth playback/presentation of the material for subscribers. For example, a client with a peak upload bandwidth of 128Kbps participating in an overlay multicast requiring 48Kbps can only relay the communication to, at most, two additional participants.

This leads us to study the minimum average-latency degree-bounded directed spanning tree problem, a well known NP-hard problem [19]. Recent works propose approximate solutions to this problem [16, 59, 60]. This work differs slightly in the study of this problem, in that we focus on the additional constraint of minimizing the depth or hop count from the source in the message distribution. Hop count and time-to-live (TTL) bounds are important in the design of communication protocols, particularly at the application-level, in order to minimize congestion in unstructured networks (see e.g., [15]). We consider each of the following algorithms for comparison to *Myriad* as described in the experimental methodology set forth in Chapter 4.

3.1 KLS

Kanemann et al. [42] present an algorithm (hereafter referred to as KLS) for approximating the degree-bounded minimum diameter spanning tree problem. The mechanism used in the KLS algorithm works by a process of clustering and filtering, and for complete metric networks produces a spanning tree with an approximation ratio of $O(\sqrt{log_B n})$ for diameter, where B is the degree bound and n is the number of nodes. Diameter is used as a primary optimization metric as it is assumed that any pair of nodes in the network would initiate communication with each other. Degree constraints are modified by "quality of service and technological constraints" [42], what we have been referring to as bandwidth limitations.

The *bounded degree minimum diameter spanning tree problem* (BDST) is defined such

that given a complete graph $G = (V, E)$ where edges obey a symmetric length function and an out degree $B \geq 2$, produce a tree T with maximum node-degree B and minimum diameter. The minimization metric is exactly defined as:

$$\Delta(T) := max_{u,v \in V} dist_l^T(u, v) \tag{3.1}$$

which is simply the maximum distance in the tree T between any two verticies $u, v \in V$ when using the metric function defined by l.

KLS functions by performing a binary search over a field of estimated diameter values, where the estimated diameter is denoted by Δ, while producing a tree of diameter no more than $O(\sqrt{\log_B n})\Delta$. In our implementation we start by setting delta to the length of the shortest edge between any two vertices in the metric space. This value is then multiplied by 2 in repetitive bases through the algorithm until Δ is greater than the the length of the shortest edge times the number of vertices in the graph, or until the algorithm only produces one cluster during execution.

The main body of the algorithm executes as such. A threshold value α is initialized to $\Delta/\sqrt{\log_B n}$. Clusters are then formed based on a coverage radius of 3α. Given the nodes not yet assigned to a cluster, a representative is chosen based on the maximum coverage of unassigned nodes. This process repeats until all nodes have been assigned to a cluster. Meta-organization is done on the clusters at this point, forming a hierarchy over the unorganized clusters. The largest cluster is the "root cluster," and all other clusters are assigned a parent in size order. In our experiments, this is often a 2 level tree over the clusters.

To round out the algorithm, the internal structure of the local clusters is set. This local structure is done such that the cluster itself is of minimum height. While Kanemann et al. [42] show that there is a maximum number of edges involved, we have shown a bound for

maximum distance from the root for any node in a local cluster (or compact tree in general) in Theorem 2.3.

This leaves us the ability to insert any compact tree algorithm for local cluster organization. Thus, in our experiments we examine a quick breadth-first search based construction and using *Myriad* for cluster organization as described in Chapter 4.

3.2 MDDBST

The MDDBST algorithm comes from work presented by Shi et al. [60] and is positioned as an application-level multicast organization algorithm in a similar fashion to *Myriad*. MDDBST is closest competitor to *Myriad* in terms of average-latency measurements, but produces trees of much greater depth (Chapter 4).

The design goal for MDDBST is to create a tree of minimum average-latency (categorized as delay optimization) with bandwidth limitations manifested as degree constraints. The delay optimization is categorized as a diameter minimization problem in a similar fashion to [42]. By the theorem stated in [60] the decision version of finding such a tree is NP-complete, and the optimization problem is an NP-hard problem.

The heuristic algorithm, MDDBST, is a greedy algorithm that functions in a similar fashion to Prim's algorithm for calculating minimum spanning trees. MDDBST requires an input graph $G = (V, E)$ that has a complete cost metric $c(u, v) \forall u, v \in V$ and a degree constraint $d_max(v) \forall V$. As with *Myriad*, MDDBST does allow for variable out-degree among the vertices, but our experiments use a consistent degree bound to garner the best comparisons.

MDDST proceeds by starting with a single node tree (the root) and looping once for each additional vertex $v \in V$ until all vertices are in the result tree T. On each pass, each vertex is updated to calculate the longest distance required to connect that vertex to a vertex already in T, considering the degree constraints of the vertices in T. After the maximum distance to

connect all remaining vertices is calculated, the minimum required edge distance is selected. The process then repeats.

3.3 OMNI

From an implementation standpoint, OMNI [16] is the most difficult to implement since it is described as a distributed algorithm. OMNI is a two-tier system requiring an infrastructure of dedicated Multicast Service Nodes (MSNs [59]), and clients connect directly to a multicast service node or organized using some other technique. In a similar fashion to the algorithms mentioned so far, degree constraints are observed for the MSN set, as motivated by bandwidth utilization. The ultimate goal of OMNI is to produce a degree-bounded spanning tree of minimum average-latency and minimum maximum-latency over the MSN set.

OMNI is initialized by creating a binary tree in which vertices are placed in the tree by their ascending distance from the root. After that, there are dynamic optimizations that occur in parallel at each vertex. For simulation, we have constructed a non-distributed version of OMNI that simulates the distributed version of the algorithm.

Dynamic transformations in OMNI are categorized by *local transformations* and *probabilistic transformations*. The local transformations are defined as follows:

- *Child-Promote:* If an MSN has available degree, one of its grandchildren is promoted to be a direct child, subject to the constraint that average-latency is improved for the subtree.

- *Parent-Child Swap:* A parent and child are swapped in the tree (the current subtree rooted at the child are not moved) if the operation will improve the average-latency.

- *Iso-level-2 Swap:* Two MSNs are the same level in the tree swap their positions, the

two nodes must have the same grandparent in the tree. The operation is performed if the average-latency is improved.

- *Iso-level-2 Transfer:* Moves an MSN to a different parent, keeping that node in the same level, with the same grandparent. Since this is not a swap, the operation must find a transfer spot that has available out-degree and improves average-latency.

- *Aniso-level-1-2 Swap:* A swap of two MSNs (i and j) such that the parent of i is the grandparent of j and average-latency is improved for the subtree.

The local transformations are designed to achieve local minima, but do not consider global concerns. Probabilistic transformations address global concerns by performing random swaps taking into consideration the client population at each MSN vertex.

3.4 Comparison with Known Bounds

Table 3.1 shows the currently known theoretical bounds for the algorithms studied. Bounds are not known in all categories for all algorithms. The *Myriad* algorithm has a latency bound of $O(\log_B n\, r)$ (for compact trees, by Corollary 2.1), where r is the radius of the graph. Let r represent the radius of the metric space such that $r = maxd_m(x, y) \forall x, y \in V$. Forcing a compact tree with *Myriad* also gives a bound on the maximum depth (hop count + 1). This bound is $O(\log_B n)$ and is equal to the minimum depth required to form a tree of out-degree B over n nodes, $\lceil \log_B n \rceil$.

Let Δ^* represent the minimum diameter of a spanning tree over G.

Table 3.1: Known theoretical bounds for the algorithms studied. For latency bounds, r is half the maximum distance between two vertices. For the KLS algorithm, Δ is the estimated diameter of the spanning tree. Results for OMNI are presented in [16], and results for KLS are presented in [42].

Algorithm	Max Hops	Maximum Latency	Average Latency	Stretch
Myriad (Compact)	$O(\log_B n)$	$O(\log_B n\, r)$	$O(\log_B n\, r)$	$O(\log_B n)$
KLS [42] w/ Myriad	$O(\log_B n)$	–	–	–
MDDBST [60]	–	–	–	–
OMNI [16]	$O(\log_2 n)$	$O(log_2 n\, r)$	–	–
KLS [42]	$O(\log_B n)$	$O(\Delta^* \sqrt{\log_B n})$	–	–

Chapter 4

Experimental Methodology and Experimental Results

In order to asses *Myriad*'s ability to construct multicast trees, we consider a strategy of repeated experiments over many network topologies and compare the performance in a study of 6 algorithms:

1. KLS [42]
2. Hybrid, KLS with *Myriad*
3. *Myriad*
4. *Myriad*, with compaction
5. MDDBST [60]
6. *OMNI* [16].

Each of these algorithms is tested using several types of metric spaces, including two dimensional random topologies in the Euclidean plane, internet-like topologies generated by GT-ITM [67], and high dimensional hypercube network topologies. Within each of the

topologies, the network size and out-degree bound are varied in order to capture a wide range of data. Details of the experimental setup follow.

The experimental methodology is imperative in the development of the *Myriad* algorithm. We used a process of informed analysis, algorithm design, and testing in a repetitive, iterative fashion. Constant observation of the running algorithm through interactive visualization and analysis of output allowed for continual refinement of the solution. Testing improved the *Myriad* because of the volume of test data that we were able to examine. By using different visualization techniques, we were able to see where *Myriad* was not making the best decisions and alter the algorithm.

Here, we describe the *GraphSim* software that we have built and its uses in experimentation (Section 4.1). Then, the different graph and network topology models are shown (Section 4.2). Lastly, we detail the approach taken in the experimental study, including the input parameter field and conditions (Section 4.4).

4.1 *GraphSim* as an Experimental Testbed

To aid the investigation of *DC-MAL* and the development of the *Myriad* algorithm, we have developed a graph simulation framework called *GraphSim*. *GraphSim* is at the core an implementation of the graph abstract data type. There are two distinct implementations, an object-oriented graph model and an implementation that uses bit labels (described later). *GraphSim* is written in the Java programming language, and makes extensive use of the `interface` language construct, allowing algorithms to be written once and tested against different implementations. This is precisely how experiments are conducted on different topologies requiring different internal representations.

The basic architecture of *GraphSim* is to separate the graph interface, implementations,

algorithms, analysis utilities, and topology generators. This modular approach has allowed for agility within the framework and quick changes in the process of experimentation.

The first graph implementation developed is the object-oriented graph implementation. It supports named vertices as well as undirected and directed edges. A network with asymmetric latencies can be simulated by creating a graph with directed edges between two vertices and assigning different weights to the edges. This implementation also supports a concept called 'layers.' Layers allow the same graph to be quickly cloned so that multiple algorithms can be run either sequentially or in parallel over a graph.

A graph implementation based on bit labels (called *BitGraph*) has been added in order to reduce the memory footprint required to represent exceptionally large graphs. Vertices are represented by anywhere from 2 to 32 bits, with that bit string serving as both the label and the position in a high dimensional space. Two different metrics over the vector space n are considered, the ℓ_1 norm and the ℓ_2 norm (it is not difficult to extend to any ℓ_p). By definition, both of these norms obey the triangle inequality. Either of these metric functions allow us to represent a high dimensional hypercube topology with minimum storage.

The ℓ_1 norm is defined as the hamming distance, or the count of the number of positions by which two bit labels differ. From an implementation standpoint, this is the number of 1 bits in the xor of the two bit strings. The Euclidean ℓ_2 norm is simply the square root of ℓ_1. Since edge distances are calculated at runtime, the *BitGraph* implementation can accept custom objects implementing any given symmetric distance metric.

Both the object-oriented and bit based graph implementations allow for individual edges to be marked as *overlay edges*. This allows for an algorithm to leave the entire graph in place while marking which edges participate in a given solution (spanning tree, multicast tree, tour, etc.). These overlay edges can also be easily reset, allowing for execution of different algorithms on the same graph object without requiring the graph to be created multiple times.

GraphSim contains a collection of graph algorithms and is easily extendable using the provided abstract class, `GraphAlgorithm`. Each graph algorithm simply extends this abstract class, and it then becomes possible to execute algorithms and chain the executions of algorithms using the command pattern [33]. The suite of included algorithms in includes classes for:

1. Calculating average subtree latency in the overlay
2. Calculating the diameter of an overlay
3. Finding the longest edge in the graph or overlay
4. Finding the shortest edge in the graph or overlay
5. Calculating the minimum spanning tree
6. Calculating a rooted shortest path tree
7. *Myriad* [41] (with and without compactness)
8. MDDBST [60]
9. OMNI [16]
10. KLS [42]

as well as other algorithms used to compose those stated above and for measurement of graphs and overlays on those graphs.

In order to assist with algorithm development and understanding, *GraphSim* includes a set of visualization tools. Graphs can be viewed in a scaled 2 dimensional coordinate space, live, as algorithms run. This same visualization can be set to output image files displaying the results or progress of an algorithm. *GraphSim* is also capable of creating graphs in the DOT file format that can be rendered by GraphViz [4].

Experiments using the *GraphSim* framework can be executed in parallel on a cluster computer. A self contained client/server mechanism is available for coordination and scaling to any number of networked computers.

Random 2-dimensional graphs using euclidian distance as the edge metric can be generated using the *tunable topology generator* included with *GraphSim*. This topology generator is configurable on the parameters of:

1. number of hosts
2. number of domains using network address translation
3. percentage of hosts using network address translation
4. maximum latency between any two hosts
5. out degree for each host
6. if the out degree for a host should be random.

Network address translation (NAT) [8] is simulated through the removal of inbound edges for all hosts in a domain except for a designated router for that domain. This feature is not used in this study, but allows for easy extension and future experimentation for networks of this type.

Multicast Live is the most recent addition to the *GraphSim* package and allows for point and click interaction with a graph and shows live visualization as multicast trees are recalculated instantly. This software has been created to help others gain a better understanding of how multicast trees are formed over graphs when out-degree constraints are involved. The main window for *Multicast Live* is shown as Figure 4.1 and the application is available as a Java applet at `http://www.graphsim.org/`.

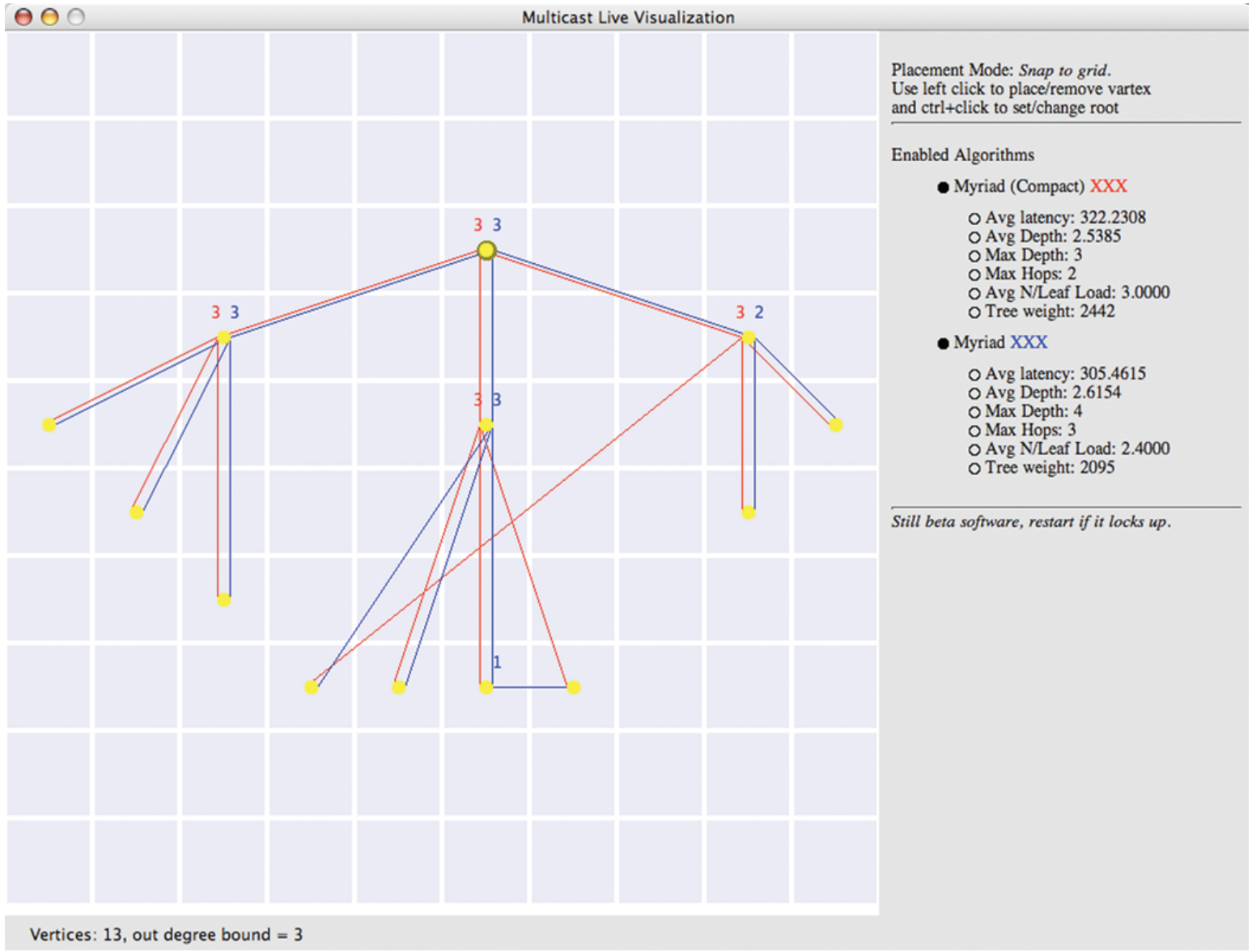


Figure 4.1: Multicast Live, interactive multicast visualization software.

4.2 Metric Space Models

Experiments were conducted over input sets from three different topology models, providing three different graph models for representing the network. All of the generated topologies share some common properties in that they are all weighted, undirected, complete graphs. The minimum one-way communication cost between any two end systems in the network is represented by the distance function $d(x, y)$. Each of the graph models provides a consistent metric function which obeys the triangle inequality, such that $d(a, b) \leq d(a, c) + d(c, b) \; \forall \; a, b, c \in V$ where $a \neq b \neq c$. The three topology models are; (1) 2-dimensional random geometric, (2) transit-stub using the Georgia Tech Internetwork Topology Models (GT-ITM) generator [67], and (3) n-dimensional Euclidean hypercube topologies.

4.2.1 $\mathbb{R}^2_n$

Two dimensional random geometric topologies are generated using the *tunable topology generator* that we have created as part of *GraphSim*. The topology generator is capable of generating graphs that are embedded in the plane at random locations, and where Euclidian distance is used as the metric function. Nodes are placed randomly in a circle, at a random distance and angle from the center of the circle.

4.2.2 $GT - ITM_n$

GT-ITM [67] is used to generate transit-stub networks which simulate sub-domains connected to each other through an internet backbone. The output graphs from GT-ITM are not complete graphs, so the missing edges are virtualized by computing the all-pairs shortest path. This models real Internet communication, as a message from a to b is abstracted to a single link while the actual message will traverse several links.

4.2.3 $H_{2^n}^p$

The third model, n-dimensional hypercube topologies, allows us to examine solutions to the degree-bounded spanning tree problem in high-dimensional metric spaces. We define the hypercube metric $H_{2^n}^p$ as follows, each point of the 2^n point metric is labeled with a unique n-bit binary label. The distance between two points of $H_{2^n}^p$ is computed using the ℓ_p norm $||x-y||_p = (\sum |x_i - y_i|^p)^{\frac{1}{p}}$. The ℓ_1 norm is the ordinary hamming distance which is calculated as the number of positions that the two point-labels differ in. The ℓ_2-norm can simply be calculated as the square root of the hamming distance. For purposes of notation we refer to the distance function for hypercube networks as $d_{\ell_p}(x, y)$.

4.3 Metric Spaces

The ability for these metric spaces to have distances that are quickly calculated aids in the ability to rapidly execute experiments over various random networks. A network where latencies are always calculated and never stored allows us to experiment on larger networks due to the smaller memory footprint within the simulator. This form of representation is used for hypercube topologies.

This allows for simulation to be performed over a complete graph without regard to the placement of that graph in a network. At the start of the simulation we are assumed to have all necessary information that would need to be gathered by the mechanism of the overlay network being utilized, allowing for simulation of entire or segmented networks. We further assume that this is how implementation would proceed in a deployable solutions, that multicast routing decisions made at any level in the hierarchy are based on complete information for that network segment or subsegment.

4.4 Empirical Study

We have conducted an empirical study aimed at measuring the performance of the *Myriad* solution, compared against existing algorithms for networks of different sizes, degree bounds, and topology models. We use *GraphSim*, integrated with the tunable topology generator, which allows for automated, repetitive execution and measurement of graph algorithms on many different random graphs in a short period of time. *GraphSim* also includes a module for loading graphs generated by GT-ITM [67] that have been converted to the included alternative ("alt") format. *GraphSim* is object-oriented and designed to make extensive use of interfaces, allowing for multiple implementations of the graph data structure. For experimentation, we use the version of *GraphSim* capable of running on a high performance computing (HPC) cluster. Experiments have been conducted using HPC clusters at both the University of Cincinnati and at Miami University.

Given the base set of algorithms and topologies described in this Chapter, we devised a suite of repeatable experiments that can run unattended on a cluster and scale to use the amount of resources available. One large experiment is run for each of the topology models and is set up as a parameter sweep over the appropriate parameters for that topology.

For random two dimensional topologies we vary the *number of nodes* and *the out-degree bound*. This setup also allows for an additional variable parameter of maximum latency between any two nodes. During early experimentation we found that varying this parameter only scaled the results proportionately, thus we have fixed the maximum latency to be 5000 units. Each parameter set is then tested over different random networks. Table 4.1 shows the actual parameter set used for experiments. In the table, an increment of 1 for the *maximum latency* field is required to cause the software to exceed the maximum and then move on to the next iteration.

Since OMNI does not organize the entire network, but rather a subset of the network

Parameter	Minimum	Maximum	Increment
Network Size	100	1600	150
Out-degree Bound	3	6	1
Maximum Latency	5000	5000	1
Repetitions	2 to 5		

Table 4.1: Parameter set for random two dimensional topology experiments.

refereed to as Multicast Service Nodes (MSNs), results for OMNI must be interpreted with the knowledge that while MSNs obey out-degree constraints when talking to each other, those constraints are broken when the MSN starts to serve clients. This is by design in the OMNI description, and we have accounted for this in the transit-stub topology experiments. *Further discussion on this point is given along with the experimental results, Section 4.5.*

Transit-stub networks generated by GT-ITM [67] must be in place before experiments can be run. A large group of transit-stub networks has been generated and converted to a format for easy loading into *GraphSim*. We focused on 600 and 1000 node networks (100 of each) and tested each network for an out-degree bounds of $3, 4, 5, 6$, and 7. The same algorithms are run on the transit-stub networks, except that OMNI is executed first and allowed to select the best 128 MSNs out of the starting set. This is done in order to simulate other algorithms in the real world scenario of deploying owned servers and allowing geographically dispersed clients to connect to the nearest server. The network is reduced to the MSN set selected by OMNI, and then the remaining algorithms are used to organize the MSNs. Results for this category are over the exact same set of nodes and do not violate out-degree constraints in any way.

Hypercube networks allow for the greatest parameter sweep due to their compact representation and relative ease of generation. For these experiments, there are two parameters to vary: *bits for node representation* and *out-degree bound*. The number of bits used to represent an individual node in the network is used to its fullest extent, and thus 8 bits

of representation allow for a network of 256 nodes. The OMNI algorithm is not run on hypercube topologies. Table 4.2 contains the parameter set used for hypercube topology experiments.

Parameter	**Minimum**	**Maximum**	**Increment**
Representation Bits	8	12	1
Out-degree Bound	3	7	1

Table 4.2: Parameter set for random high dimensional hypercube topology experiments.

The use of 13 bits to represent a node allows for networks of size $8,192$ to be examined. It is possible to scale to much large networks with additional time and system memory being the only requirements.

4.5 Experimental Results

Based on the experimental methodology presented in Chapter 4, extensive simulations have been conducted using *GraphSim*. The results of these experiments have been collected and analyzed in order to provide meaningful comparison between *Myriad* and other algorithms (KLS [42], MDDBST [60], and OMNI [16]) in the context of different network topologies. This is done in order to show the utility of *Myriad* under different initial conditions.

Our analysis and presentation of the experimental results is separated into three subsections grouped by the experimental topologies used. All experiments conducted indicate superior results for *Myriad* when compared to other published algorithms as described in Chapter 4.

4.5.1 Random Topologies

Two dimensional random topologies are generated that have a specified bound on distance, and that use the Euclidean distance ($\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$) as a metric function. This

defines a symmetric space where the metric function obeys the triangle inequality. The number of nodes is kept relatively low since we believe that a hierarchical solution to application level multicast will prevail, limiting the number of nodes involved in any single multicast tree calculation, but leaving several parallel tree commutations within any given subscription group and/or overlay routing network.

While many results have been captured, we examine some specific scenarios. It is also necessary to make a notation on OMNI. Our implementation of OMNI is a centralized simulation of the distributed algorithm described by Banerjee et al. [16]. All 5 types of local transformations and the probabilistic transformations are implemented. Our experiments utilize a P_{rand} value of 0.1 and a simulated annealing temperature parameter of 10, having chosen both of these values based on the results presented in [16]. OMNI is not directly comparable because, by definition, MSNs are assigned a client load that may be above the degree-bound. The authors of OMNI do not specify if or how client load is scheduled, so we assume that they are all serviced directly by their MSN. This, however, leaves MSNs above the degree-bound in some cases since degree-bound is enforced for MSN to MSN connections and does not seem to be enforced for MSN to client connections. This can have an effect on the results presented for OMNI as nodes not in the MSN group are simply attached to their closest MSN. In terms of peer-to-peer networks, OMNI would be considered a hybrid system rather than purely decentralized overlay network.

There are six different algorithm (or combinations of algorithms) to report on:

1. KLS [42] using breadth-first search based cluster organization
2. KLS with *Myriad* (compact) for cluster organization
3. *Myriad* without forcing a compact tree
4. *Myriad* with a compact tree

5. MDDBST [60]

6. *OMNI* [16].

These same six algorithm implementations apply to all experiments for the two dimensional random and GT-ITM [67] topologies.

550 nodes, Out-degree 3, Random 2D

The experiment for 550 nodes and out-degree 3 has been repeated 4 times, with different random distributions each time. As a starting point, Table 4.3 shows the average and standard deviation for the metrics of *depth* and *average-latency*. Since the compact version of *Myriad* guarantees a compact tree, all calculated trees are of the minimum number of hops, $\lceil log_3(500) \rceil = 6$, with depth being the number of hops plus one. The best overall tree created (in terms over average-latency) is the tree produced by the non-compact version of *Myriad* which exhibits not only the lowest average average-latency measurement, but also the narrowest deviation in the trees produced over 4 different random topologies. The differences between the average-latency result for the compact and non-compact version of *Myriad* illustrate empirically the tradeoff between depth and average-latency when a tree of minimum of height is desired. Figure 4.2 (Page 64) shows a plot of this same information, with depth (x-axis) plotted against average-latency (y-axis). The radius of the circles is relative and scaled to the normalized product of depth and average latency as shown in table 4.4.

In examining both of the measures of maximum depth and average-latency, experiments for this network size ($n = 550$) and out-degree bound ($B = 3$) demonstrate how both the compact and non-compact version of *Myriad* out perform the other algorithms in the experiment. In examining the average over all experimental runs, using *Myriad* but not forcing a compact tree produces the lowest average-latency measurement at 1333.53, with

Table 4.3: Results for the depth and average-latency measurement, for random 2D networks, with 550 nodes and out-degree 3. Results are averaged over 4 runs.

Algorithm	Max Depth	Std. Deviation	Average-Latency	Std. Deviation
Myriad (Compact)	7.00	0.00	1560.32	97.25
Myriad	8.75	0.96	1333.53	29.47
MDDBST	14.00	2.00	1347.32	39.61
KLS w/ Myriad	9.75	1.26	1584.30	155.75
KLS	8.75	0.96	3079.97	401.97
OMNI	9.75	0.96	1520.05	143.88

Table 4.4: Normalized values for maximum depth, average-latency, and the product of these two metrics.

Algorithm	|Max Depth|	|Average-Latency|	Product
Myriad (Compact)	0.500	0.507	0.253
Myriad	0.625	0.433	0.271
MDDBST	1.000	0.437	0.437
KLS w/ Myriad	0.696	0.514	0.358
KLS	0.625	1.000	0.625
OMNI	0.696	0.494	0.344

the closest competitor being MDDBST. By the way that MDDBST forms a degree-bounded spanning tree, there is no attempt to minimize depth in the tree. Even in its non-compact form *Myriad*, reduces the height of the tree by promoting subtrees where average-latency can be improved, yielding a lower maximum depth. These factors combine to produce values that compete well when considering the normalized values of maximum depth and average-latency, and the product of these two measures.

1300 nodes, Out-degree 6, Random 2D

The depth-latency tradeoffs are magnified by increasing the number of nodes as well as the out-degree. For this larger network of 1300 nodes and an out-degree bound of 6, and two experimental runs *Myriad* (non-compact) still produces the lowest average-latency at 1297.33. This low average-latency measurement can not be held as the tree is reduced to a minimum height, and the compact tree product by *Myriad* averages an average-latency

of 1707.87. Table 4.5 shows the depth, average-latency measurements, and the standard deviation of these two metrics. The normalized depth, and average latency appear in Table 4.6 along with the product of these two metrics. Figure 4.3 (Page 65) plots the depth versus the average-latency, with the radius of each circle defined by the product of the normalized depth and average-latency values (as shown in Table 4.6).

Myriad exhibits the lowest average-latency measurement and with the least deviation over the input sets. Without forcing a compact tree, *Myriad* still produces trees with the lowest depth measurement when compared to the other algorithms in the study. Consistently, MDDBST is competitive in the measurement of average-latency, but with trees containing more levels then necessary, allowing for the possibility of more failure points from the root to node paths.

Table 4.5: Results for the depth and average-latency measurement, for random 2D networks, with 1300 nodes and out-degree 6. Results are averaged over 2 runs.

Algorithm	Max Depth	Std. Deviation	Average-Latency	Std. Deviation
Myriad (Compact)	5.0	0.00	1707.87	318.08
Myriad	6.5	0.71	1297.33	1.34
MDDBST	14.5	0.71	1325.55	59.23
KLS w/ Myriad	8.0	1.41	1385.25	134.29
KLS	7.0	0.00	2057.03	62.75
OMNI	7.0	0.00	1336.92	121.34

Table 4.6: Normalized values for maximum depth, average-latency, and the product of these two metrics.

Algorithm	|Max Depth|	|Average-Latency|	Product
Myriad (Compact)	0.345	0.830	0.286
Myriad	0.448	0.631	0.283
MDDBST	1.000	0.644	0.644
KLS w/ Myriad	0.552	0.673	0.372
KLS	0.483	1.000	0.483
OMNI	0.483	0.650	0.314

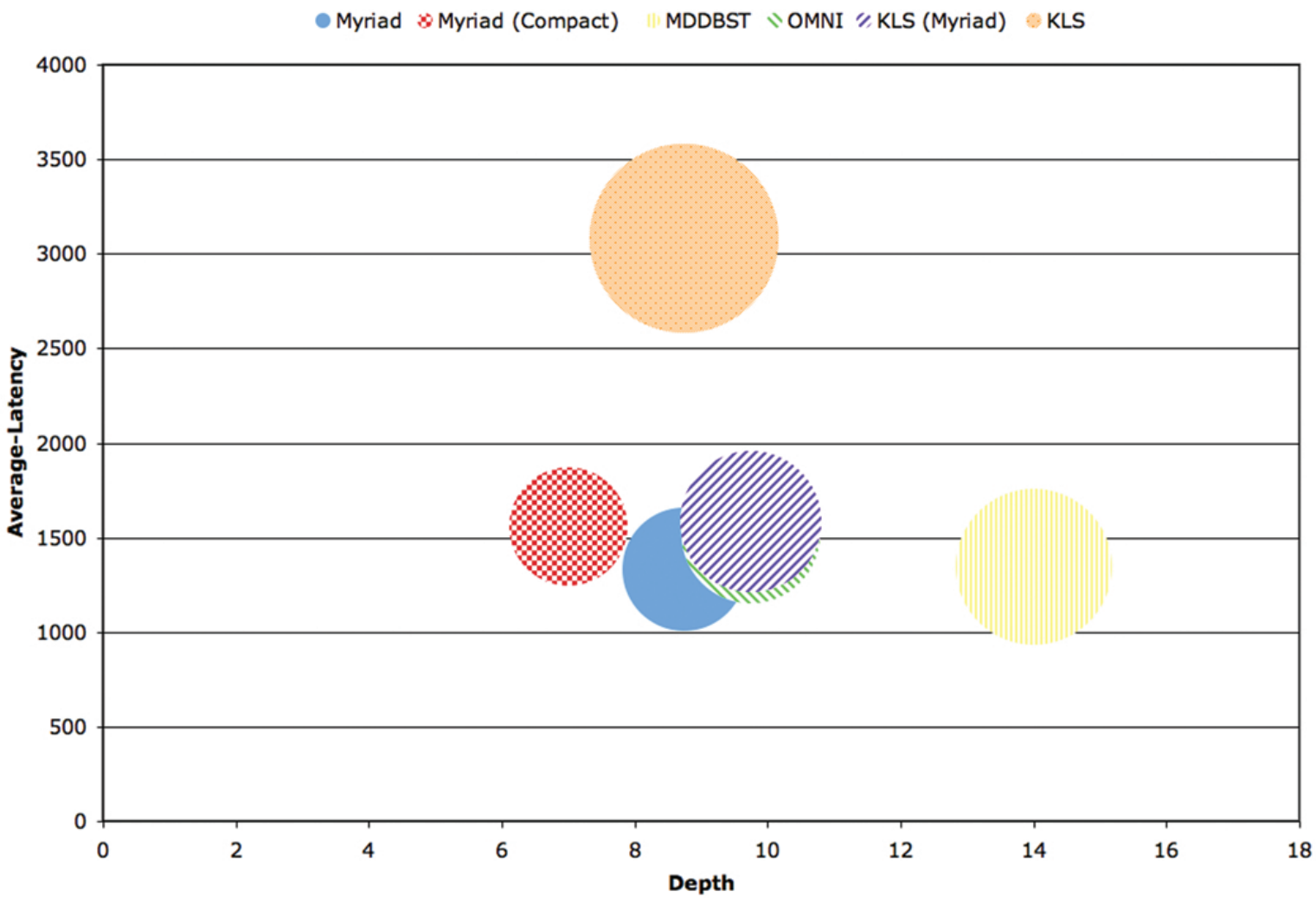


Figure 4.2: Plot of depth versus average-latency, n=550, B=3, random 2D topology.

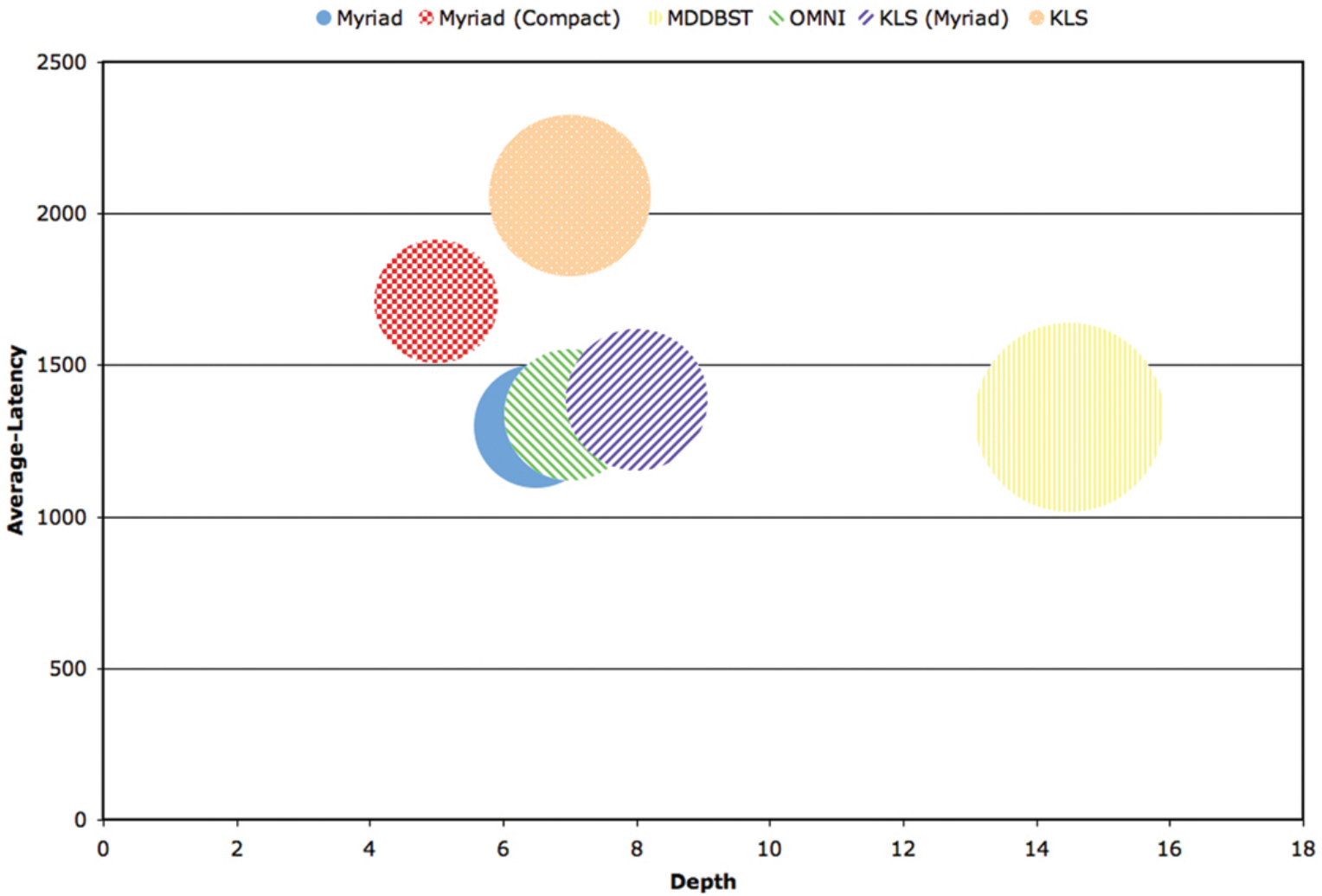


Figure 4.3: Plot of depth versus average-latency, n=1300, B=6, random 2D topology.

Trends in Random 2D

In order to summarize the remaining results for two dimensional random topologies, we present some trends as observed in the results of the experiments. First, we present some trends for out-degree 3 experiments for networks of size 100, 250, 400, 550, and 700.

Table 4.7 shows the how the average latency measurements change with the addition of extra nodes. Average-latency values tend to go down as the network gets larger since the area becomes more saturated and the length of edges required to span the graph become shorter.

Table 4.7: Results for the average-latency, averaged over multiple runs (different for each network size) of each network size, for out-degree 3 and network sizes of 100, 250, 400, 550, and 700.

	Myriad	Myriad (Compact)	MDDBST	OMNI	KLS (Myriad)	KLS
100	1530.15	1831.86	1589.06	1633.53	1568.85	4126.29
250	1435.68	1724.26	1480.36	1802.31	1426.55	3166.41
400	1374.75	1631.60	1388.41	1559.97	1544.29	3099.96
550	1333.53	1560.32	1347.32	1520.05	1584.30	3079.97
700	1337.55	1718.48	1353.92	1751.00	1570.43	2960.82

In Table 4.8 maximum depth is show for each algorithm using the same experimental runs as shown in Table 4.7. The compact version of *Myriad* shows the minimum possible maximum depth for the network sizes shown. The average-latency and depth are brought together in Table 4.9, showing the product of normalized depth and normalized average-latency.

As a visual reference, we provide charts for all three of these tables. Figure 4.4 shows the average-latency trend for these networks, with a fixed out-degree of 3. In this chart the *Myriad* plot line is mostly obscured by the line for MDDBST. In terms of average-latency only, *Myriad* delivers results just under what MDDBST provides. The difference in these algorithms is in the depth of the corresponding trees. Figure 4.5 is a plot of the maximum depth in the result tree for the same networks.

Table 4.8: Results for maximum depth, averaged over multiple runs (different for each network size) of each network size, for out-degree 3 and network sizes of 100, 250, 400, 550, and 700.

	Myriad	Myriad (Compact)	MDDBST	OMNI	KLS (Myriad)	KLS
100	6.25	5.00	8.15	6.13	6.75	7.65
250	7.83	6.00	11.33	8.17	7.50	8.00
400	8.33	7.00	12.67	8.83	8.67	8.17
550	8.75	7.00	14.00	9.75	9.75	8.75
700	9.50	7.00	13.50	9.75	9.75	9.00

Table 4.9: Results for the product of the normalized maximum depth and average-latency, averaged over multiple runs of each network size, for out-degree 3 and network sizes of 100, 250, 400, 550, and 700.

	Myriad	Myriad (Compact)	MDDBST	OMNI	KLS (Myriad)	KLS
100	0.285	0.273	0.385	0.298	0.316	0.938
250	0.313	0.288	0.466	0.410	0.298	0.706
400	0.292	0.291	0.448	0.351	0.340	0.644
550	0.271	0.253	0.437	0.344	0.358	0.625
700	0.318	0.301	0.457	0.427	0.383	0.667

To bring everything, together Figure 4.6 is a plot of the product of the normalized average-latency and normalized maximum depth. This is the same source data as Table 4.9.

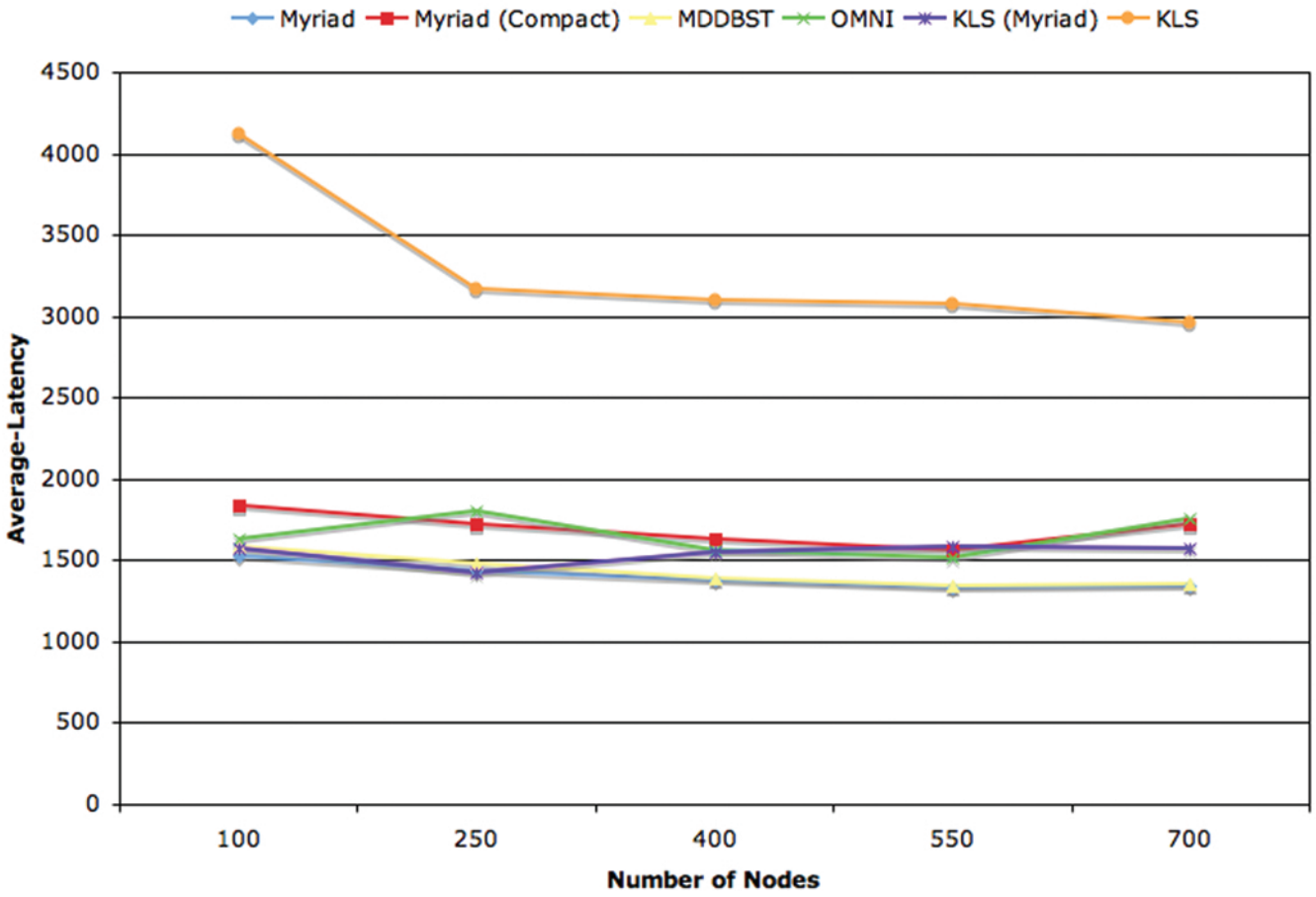


Figure 4.4: Plot of average-latency, n=(100,250,400,550,700), B=3, random 2D topology.

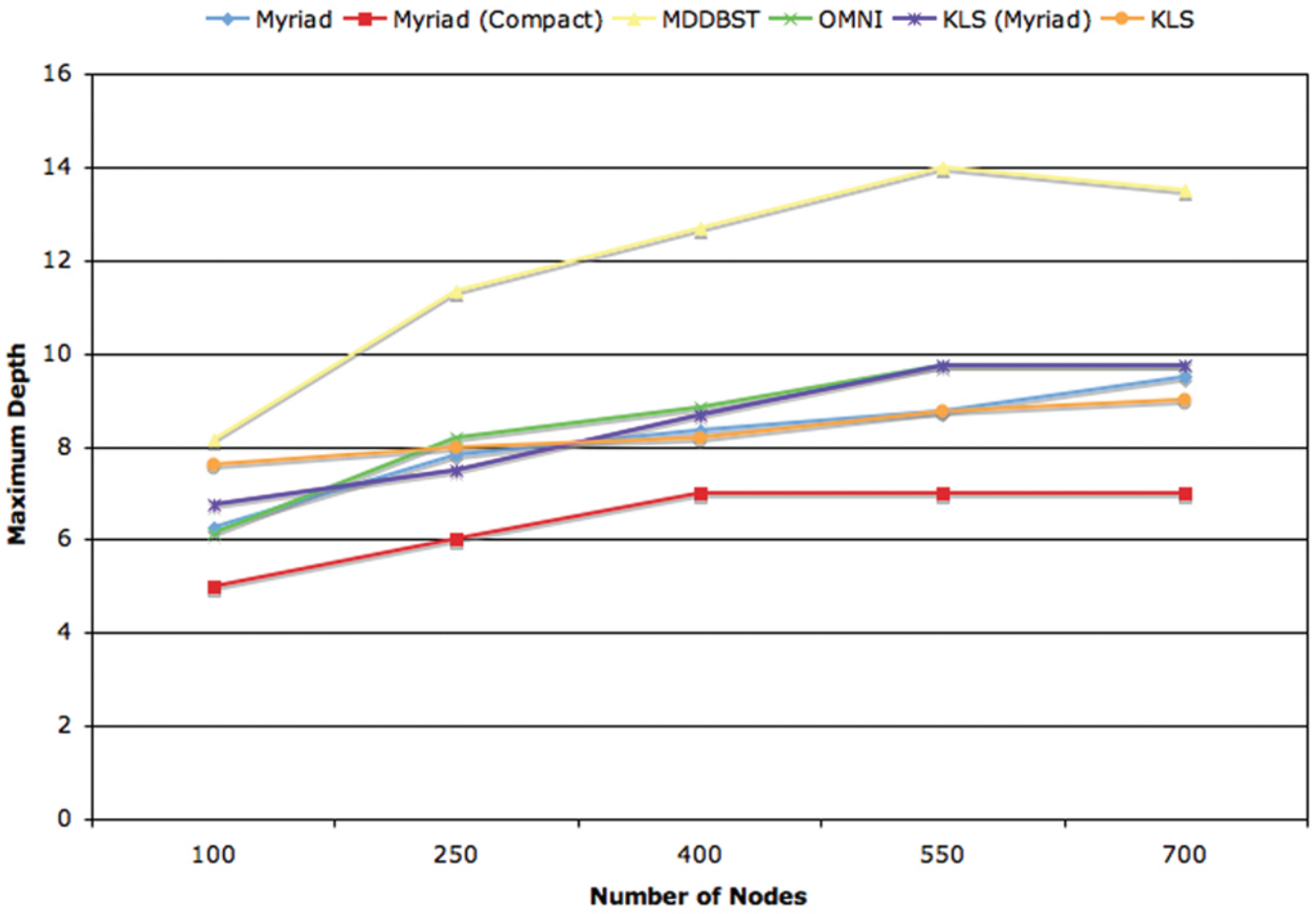


Figure 4.5: Plot of maximum depth, n=(100,250,400,550,700), B=3, random 2D topology.

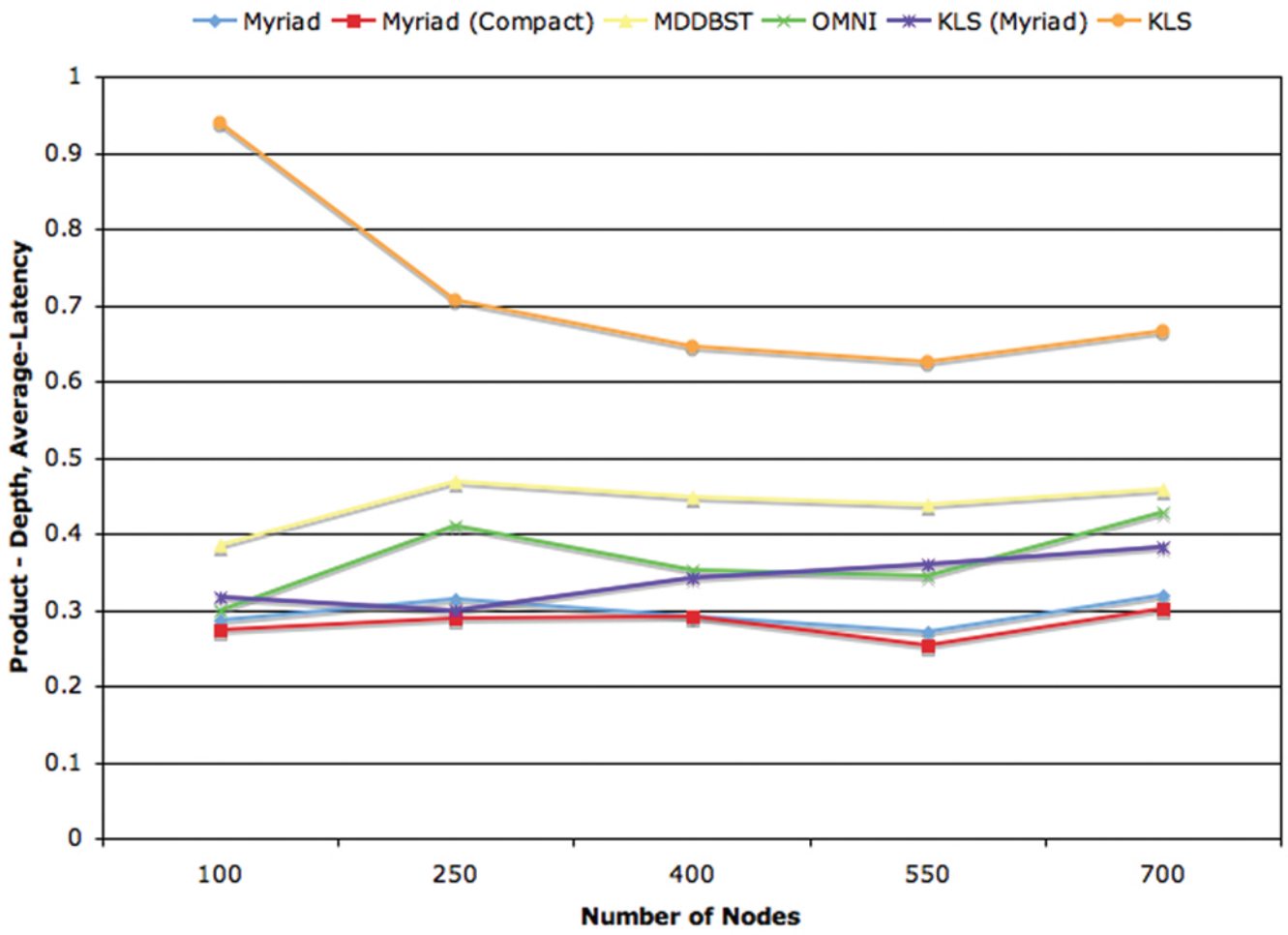


Figure 4.6: Plot of normalized product, depth and average-latency trend, n=(100,250,400,550,700), B=3, random 2D topology.

4.5.2 Internetwork Topologies

Experiments for GT-ITM [67] generated transit-stub topologies are conducted over networks of both 600 and 1000 nodes. Each network size has 100 generated topologies, of which 70 are randomly chooses for the experimental runs. Out-degree constraints are then imposed, ranging from 3 through 7. Experiments for transit-stub topologies follow the OMNI [16] model by organizing only a subset of the network, or multicast service nodes (MSNs). We select MSN nodes by first preferring "transit" nodes in the topology, and then rounding out the set by selecting "stub" nodes. A total of 128 MSNs are chosen for both network sizes examined.

600 nodes, Out-degree 3, Transit-Stub

For 600 node internetwork topologies (Table 4.10), *Myriad* delvers the best average-latency values in its non-compact form while delivering the best maximum depth for the compact form. When we consider the dual metric of the product of the normalized depth and the normalized average-latency (Table 4.11), *Myraid* (both forms) is the best performing algorithm with MDDBST and OMNI close behind. Figure 4.7 (Page 73) shows a visual plot of the data from this experiment. The center of a circle is the plot of depth versus average-latency, with the radius of the circle represented by the product of the normalized depth and normalized average-latency. In this figure, the plot of the non-compact version of *Myriad* is partially obscured by the plot for MDDBST.

Table 4.10: Results for the depth and average-latency measurement, for GT-ITM transit-stub networks, with 600 nodes and out-degree 3. Results are averaged over 70 runs.

Algorithm	Max Depth	Std. Deviation	Average-Latency	Std. Deviation
Myriad (Compact)	6.00	0.00	279.35	49.35
Myriad	8.27	0.99	212.87	35.73
MDDBST	10.61	1.45	220.76	41.74
KLS w/ Myriad	8.27	1.05	216.23	37.78
KLS	7.67	0.93	630.68	91.11
OMNI	7.17	0.87	253.50	40.97

Table 4.11: Normalized values for maximum depth, average-latency, and the product of these two metrics. GT-ITM transit-stub topology, 600 nodes, out-degree 3.

Algorithm	|Max Depth|	|Average-Latency|	Product
Myriad (Compact)	0.565	0.443	0.250
Myriad	0.774	0.338	0.261
MDDBST	1.000	0.350	0.350
KLS w/ Myriad	0.779	0.343	0.267
KLS	0.723	1.000	0.723
OMNI	0.676	0.402	0.272

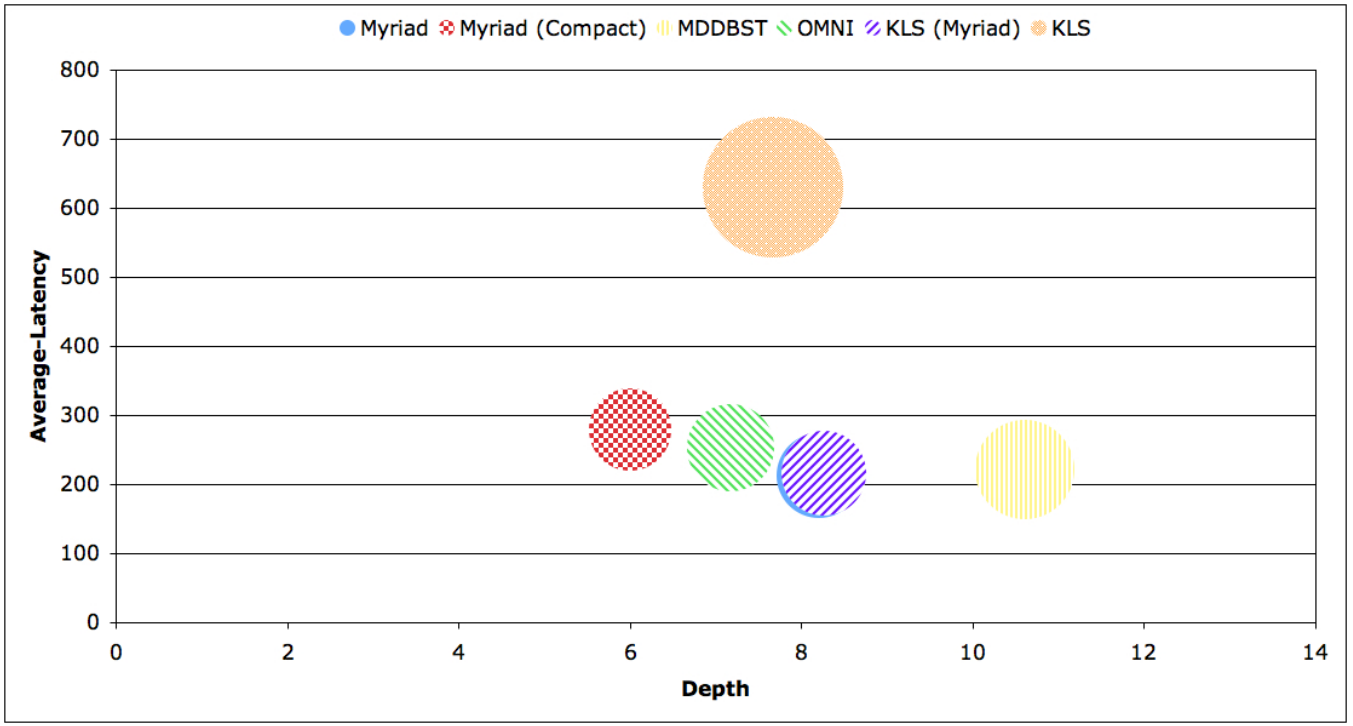


Figure 4.7: Plot of depth versus average-latency, n=600, B=3, transit-stub topology.

600 nodes, Out-degree 6, Transit-Stub

Since the ultimate input set of vertices is reduced to 128 MSNs, simply raising the out-degree bound produces a drop in minimum required depth from 6 to 4. When producing a compact tree, *Myriad* is the only algorithm that produces this tree of minimum height. OMNI does very well in this scenario, with the lowest average-latency value over the input set, but with an average maximum depth of 5.93.

The full depth and average-latency data for these experiments in presented in Table 4.12. This table shows that OMNI produces the lowest average-latency value. OMNI does better as the out-degree bound goes up, but has been found to have scalability limitation in terms of the number of vertices in the input graph. These results are plotted in Figure 4.8.

Table 4.12: Results for the depth and average-latency measurement, for GT-ITM transit-stub networks, with 600 nodes and out-degree 6. Results are averaged over 70 runs.

Algorithm	Max Depth	Std. Deviation	Average-Latency	Std. Deviation
Myriad (Compact)	4.00	0.00	244.47	58.64
Myriad	6.00	0.82	190.06	42.27
MDDBST	8.77	1.26	201.38	51.49
KLS w/ Myriad	6.00	0.82	190.94	42.30
KLS	5.11	0.94	475.13	68.15
OMNI	5.93	0.75	178.79	30.53

When considering the product of the normalized depth and normalized average-latency values (Table 4.13, the compact version of *Myriad* produces the best value. As we have noted previously, trees of minimum depth are important in reducing the number of failure points in any root to node path and for networks where a TTL constraint is meaningful.

Over all of the experiments conducted, *Myriad* has shown to be consistent in producing out-degree constrained multicast trees that exhibit average-latency measurements at or near the top (in this case). While OMNI is extremely competitive even for the combined metric of depth and average-latency in this case, *Myriad* has been found to scale to larger network sizes and produce better results for lower out-degree bounds. We believe that on the Internet

that smaller out-degree bound values will need to be used to power end-system multicast and that cluster sizes will be somewhat larger than what is examined in this experimental setup.

Table 4.13: Normalized values for maximum depth, average-latency, and the product of these two metrics. GT-ITM transit-stub topology, 600 nodes, out-degree 6.

Algorithm	|Max Depth|	|Average-Latency|	Product
Myriad (Compact)	0.456	0.515	0.235
Myriad	0.684	0.400	0.274
MDDBST	1.000	0.424	0.424
KLS w/ Myriad	0.684	0.402	0.275
KLS	0.583	1.000	0.583
OMNI	0.676	0.376	0.254

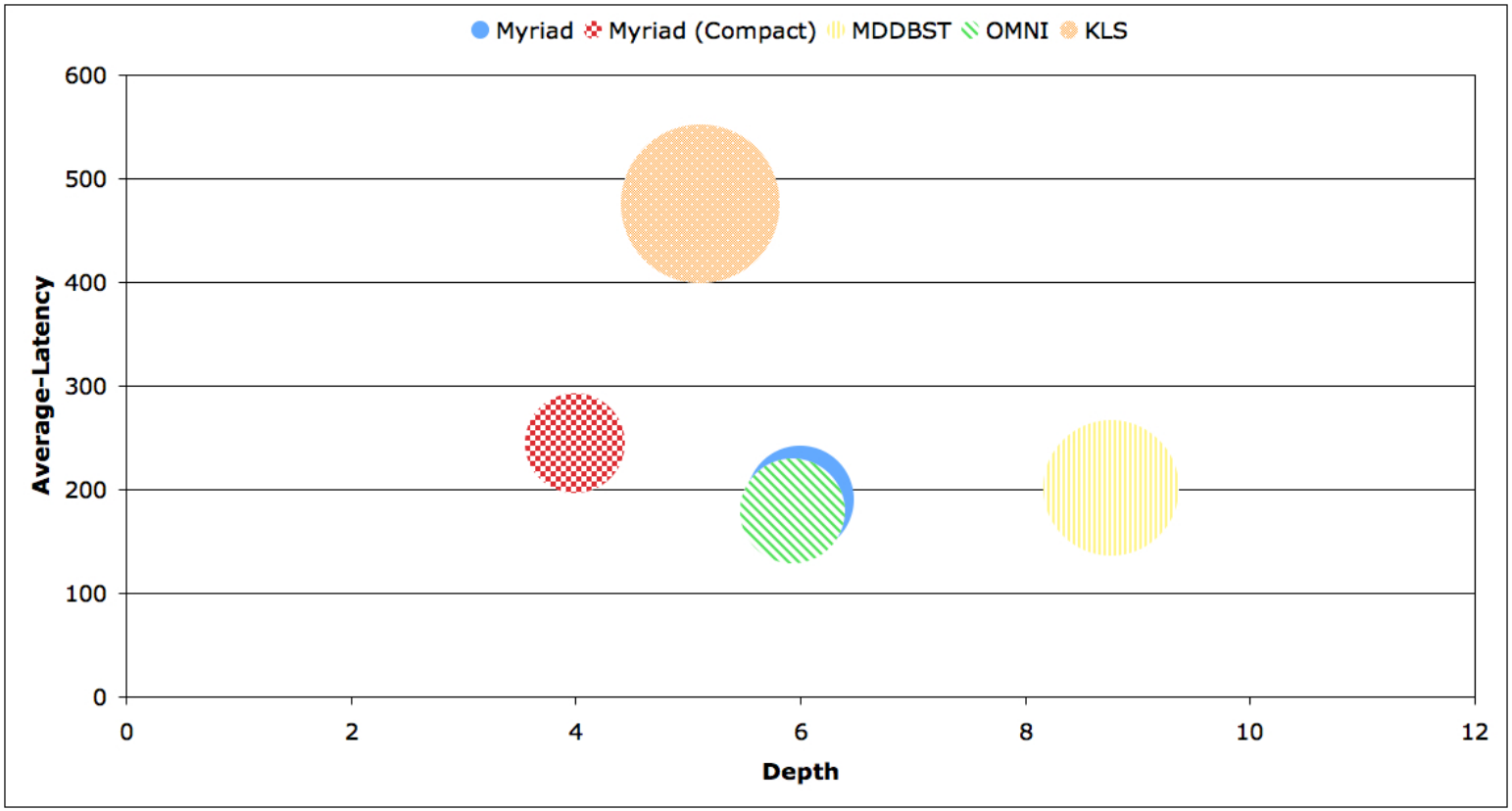


Figure 4.8: Plot of depth versus average-latency, n=600, B=6, transit-stub topology.

4.5.3 Hypercube Typologies

For hypercube topologies, results have been gathered for the parameter combinations of: 8 to 12 bits of representation (256 to 4096 nodes), out-degree bounds of 3 through 7, and both the ℓ_1 and ℓ_2 metrics. This produces a total of 50 different network setups, with 5 algorithms run on each network. The algorithms are KLS [42], KLS with *Myriad* clusters, Myriad (without compactness), Myriad (with compactness), and MDDBST [60]. OMNI is not examined for hypercube topologies since initial exploration determined that OMNI did not lend itself to this type of network topology. A general summary and analysis of the overall data for hypercube topologies is presented at the end of this section.

We highlight results from the 8 bits (256 node) experiment with out-degree 3, and the 12 bit (4096 node) experiment by examining several factors from this data. Since two both the ℓ_1 and ℓ_2 norms are used as distance metrics for the hypercube networks, we present the results for both metrics. Since the algorithms employed all give the same result when the same exact input graph is given, all experiments are executed exactly one time.

To facilitate easy comparison, Table 4.14 brings in all of the data for 8 bit hypercube networks with out-degree 3. This data includes the actual and normalized values for maximum depth and average-latency, as well as the product of the two normalized measures. In the ℓ_1 metric space, *Myriad* produces the lowest average-latency measure, while having only 1 level in the tree over optimal. As with the other topologies, the MDDBST algorithm is competitive in terms of average-latency, but not in terms of maximum depth in the tree

We also highlight a result from a larger hypercube network, using 12 bits of representation (4096 nodes) and an out-degree bound of 6. Results in this area are similar to the smaller network with *Myriad* providing the lowest average-latency measurement and the lowest product of the normalized maximum depth and average-latency values. The result for this experiment are shown in Table 4.15.

In summary of all hypercube experiments, we refer to Figures 4.9 and 4.10. This chart,

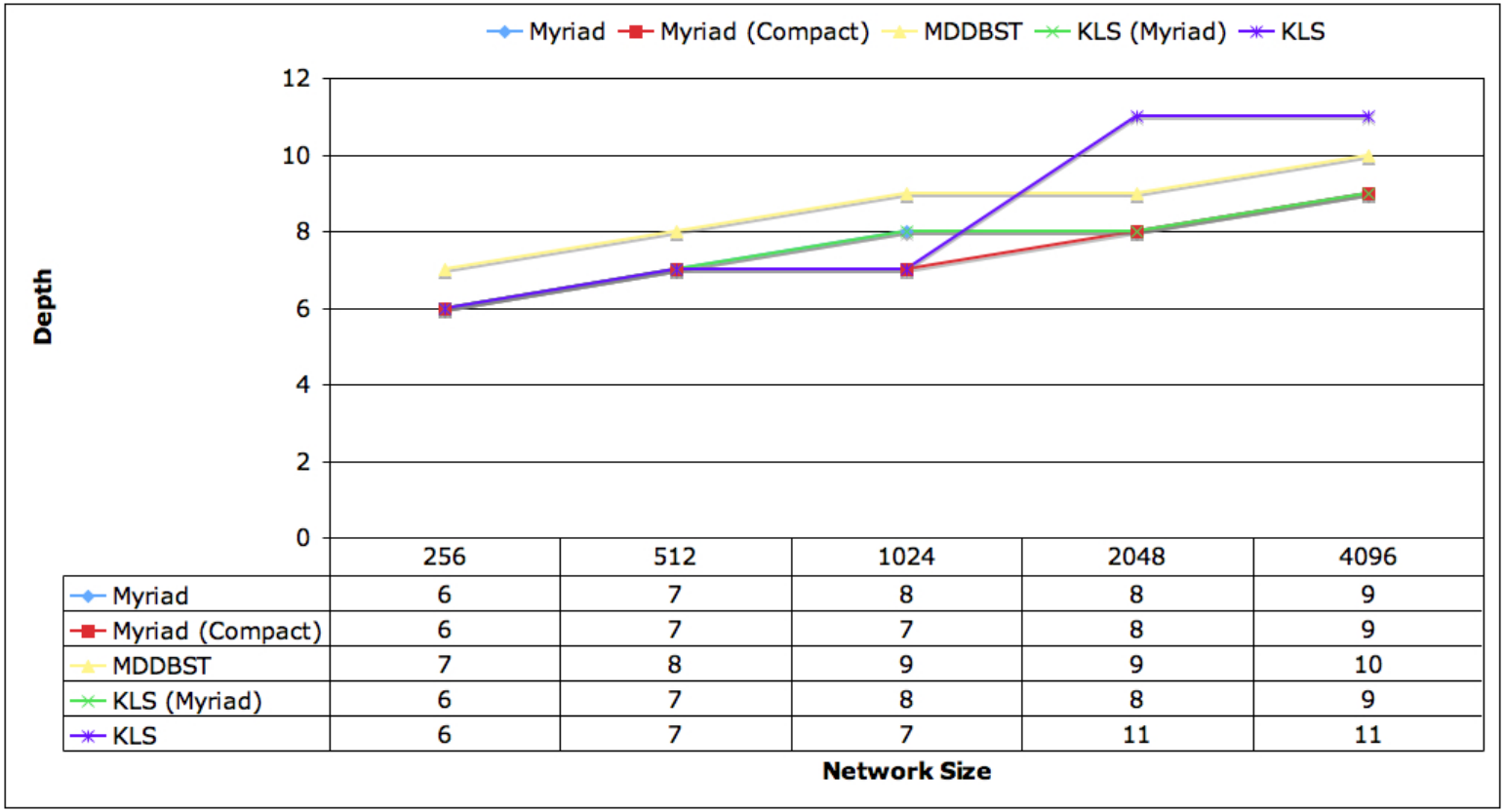
respectively, the growth of depth as network sizes increases and the average-latency measurements as network size increases. All experiments captured in these figures are for out-degree 3 trees.

Table 4.14: Depth and average-latency for all algorithms, 8 bit hypercube network, out-degree 3, showing ℓ_1 and ℓ_2 measurements. The normalized depth, normalized average-latency, and the product of these two measures is also shown.

Metric	Algorithm	Max Depth	Average-Latency	|Max Depth|	|Average-Latency|	Product
$\ell 1$	Myriad	7	5.20	0.70	0.51	0.35
$\ell 1$	Myriad (Compact)	6	5.39	0.60	0.52	0.31
$\ell 1$	MDDBST	10	6.73	1.00	0.66	0.65
$\ell 1$	KLS (Myriad)	7	5.25	0.70	0.51	0.36
$\ell 1$	KLS	8	10.28	0.80	1.00	0.80
$\ell 2$	Myriad	6	4.40	0.86	0.55	0.47
$\ell 2$	Myriad (Compact)	6	4.47	0.86	0.56	0.48
$\ell 2$	MDDBST	7	5.48	1.00	0.69	0.69
$\ell 2$	KLS (Myriad)	6	4.46	0.86	0.56	0.47
$\ell 2$	KLS	6	7 .96	0.86	1.00	0.85

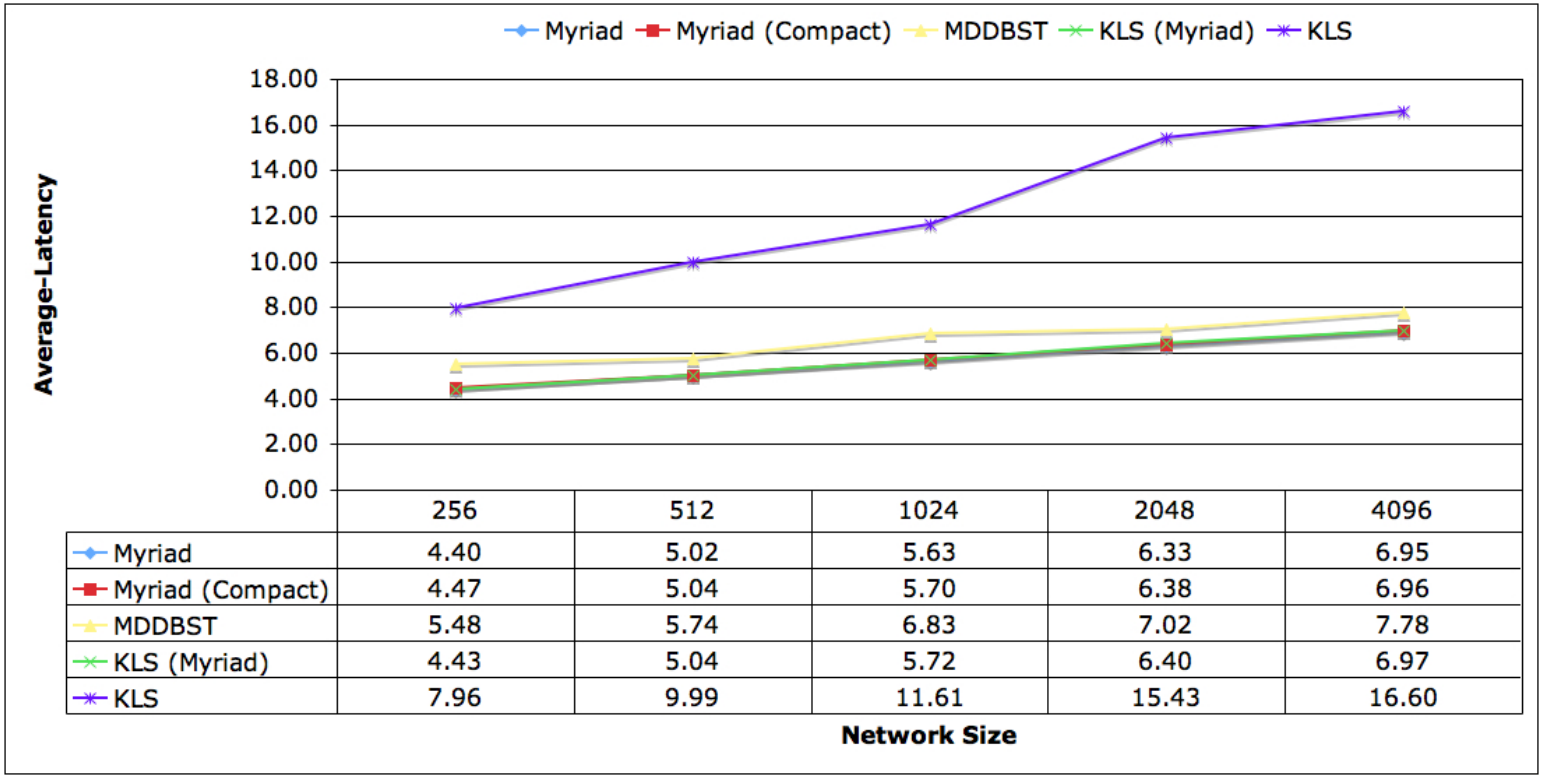
Table 4.15: Depth and average-latency for all algorithms, 12 bit hypercube network, out-degree 6, showing ℓ_1 and ℓ_2 measurements. The normalized depth, normalized average-latency, and the product of these two measures is also shown.

Metric	Algorithm	Max Depth	Average-Latency	|Max Depth|	|Average-Latency|	Product
$\ell 1$	Myriad	10	6.11	0.67	0.44	0.29
$\ell 1$	Myriad (Compact)	6	6.78	0.40	0.49	0.20
$\ell 1$	MDDBST	15	8.88	1.00	0.64	0.64
$\ell 1$	KLS (Myriad)	8	6.12	0.53	0.44	0.24
$\ell 1$	KLS	8	13.84	0.54	1.00	0.53
$\ell 2$	Myriad	6	4.81	0.67	0.39	0.26
$\ell 2$	Myriad (Compact)	6	4.86	0.67	0.39	0.26
$\ell 2$	MDDBST	7	5.59	0.78	0.45	0.35
$\ell 2$	KLS (Myriad)	6	4.82	0.67	0.39	0.26
$\ell 2$	KLS	9	12.34	1.00	1.00	1.00



	256	512	1024	2048	4096
Myriad	6	7	8	8	9
Myriad (Compact)	6	7	7	8	9
MDDBST	7	8	9	9	10
KLS (Myriad)	6	7	8	8	9
KLS	6	7	7	11	11

Figure 4.9: Plot of depth growth for hypercube experiments.



	256	512	1024	2048	4096
Myriad	4.40	5.02	5.63	6.33	6.95
Myriad (Compact)	4.47	5.04	5.70	6.38	6.96
MDDBST	5.48	5.74	6.83	7.02	7.78
KLS (Myriad)	4.43	5.04	5.72	6.40	6.97
KLS	7.96	9.99	11.61	15.43	16.60

Figure 4.10: Plot of average-latency growth for hypercube experiments.

4.6 Results Summary

Results for all three types of examined topologies (2 dimensional random, transit-stub internetwork, and high dimensional hypercube), *Myriad* is among the best performing in all cases. When considering the hybrid metric of the product of normalized depth and normalized average-latency, *Myriad* clearly outperforms all other algorithms tested.

The ability of *Myriad* to deliver the best results across all categories is a measure of how well *Myriad* adapts to the network topology given as input. This leads us to believe that there is the possibility of future adaptation of *Myriad* for different network topologies such as wireless sensor networks as well as ad hoc and mobile networks.

Chapter 5

Analysis for Real-World Systems

In this section we place this research in the context of existing real world systems and provide forward looking statements on the future for theory related to degree-bound shortest path / spanning trees and their relation to peer-to-peer overlay networks and the content dissemination on the Internet.

Of particular interest are existing peer-to-peer overlay networks including the purely decentralized loosely organized networks (Gnutella [2, 3, 22], Pastry [54], Tulip [11]), and systems that use intermediate "supernodes" (Kazaa) [14]. We focus on the possibilities of multicasting in several systems of the next few sections.

5.1 Multicasting and Peer-to-peer

Multicast routing involves determining how to send a source message (we can generalize to the single TCP[1]/IP packet or UDP[2]/IP level without loss of generality) from a host r to a subset of the hosts $u \in V$ where the hosts in the set u are defined by a logical group identifier. Broadcasting is the upper-bound case of multicasting where the message destination

[1]Transmission Control Protocol
[2]User Datagram Protocol

is the entire network ($u = V$) so we will refer to both scenarios as multicasting where the size of the multicast group u satisfies $1 \leq |u| \leq |V|$. A distinction must be made between between network supported multicast and application-level (or software multicasting). Multicasting can be useful whenever there is a large group of users who want access to the same information, and can possibly alleviate the effect of flash crowds [55].

Network level support for IP multicasting has been described in RFC 1112 [7] and the Internet Group Management Protocol (IGMP - RFC 3376) [9]. With network layer multicasting hosts are able to inform their local routers of their desire to join a multicast group, via IGMP. At the router level, messages are duplicated and delivered to all subscribing hosts [43]. This allows for a close to optimal (minimum) number of messages in order to deliver the message and avoids multiplexing (reducing traffic). Network level (IP) multicasting has not been widely adopted [32], so we will consider software solutions that implement multicasting using only unicast messages.

Chu et al. [24] set forth important arguments for implementing multicast in end systems, rather than at the IP layer. They provide four major drawbacks to network layer multicast:

1. Maintaining group memberships introduces state into a currently stateless system.

2. Greatly complicates network management since IP unicast is already loosely defined and does not always take the optimal route.

3. Concerns with the scalability and efficiency for distributing and managing internet-wide unique group identifiers.

4. The changes required to infrastructure (routers) increases cost and delays implementation.

Given those constraints, application-level multicast is not entirely without faults. In particular, application-level multicast can not ensure that a single physical link is not traversed

twice, and is thus sub-optimal for latency and bandwidth saturation. When building an application-level multicast solution, we must consider and measure the performance impact of constructing the multicast routes in an overlay network that does not have global knowledge [24].

Multicasting is a valuable tool and is a well studied problem at the network level and at the application level [17, 20, 23, 45, 59] for distribution of all sorts of applications. Potential multicast applications include multimedia distribution [58, 61, 63, 65] and news/event notification [57], as well as many other applications [1, 23, 37].

In examining application level multicast networks, we consider the possibilities for delivery of such services. Peer-to-peer (P2P) systems cover a range of distributed systems using the fringes of the Internet. These systems vary in their organization from the purely distributed (Gnutella [2], Pastry [54], Tulip [11], Chord [62]), systems that user intermediate "supernodes" (Kazaa), and even extends to systems with a centralized server (original Napster) [14].

P2P overlay networks provide a logical alternative routing structure than what is used for all internet traffic, creating a virtual network that overlays the physical network. P2P overlay networks are a natural fit for application-level multicast routing since protocols already exist for the construction of general routing tables and for the actual routing of individual messages. So we must find new ways to leverage the infrastructure of the P2P network in order to provide efficient multicast delivery. In this Chapter, case studies are presented on how existing P2P overlay networks can be equipped with (or gain improved) multicast functionality through application of the work presented here.

5.2 Multicasting in Unstructured Peer-to-peer

Gnutella [2] is a totally decentralized peer-to-peer overlay network [14] that performs all of its maintenance and query operations via broadcasting [3]. A peer joins the network by contacting a known member and broadcasting a `PING` message to all of its directly connected nodes, who in turn broadcast the message to their connected nodes. Queries are also sent in this manner, broadcasting a query to each of a node's neighbors who then broadcast to their neighbors. Responses are then sent back to the host (allowing for it to traverse the same route in reverse, plus up to two additional hops). Each message is given a time to live (TTL) which is (usually) set to seven hops. This allows any given node to reach a subset of nodes in the network, but is truly based on who you know. This construct amounts to flooding for nearly all communication that deals with network setup and searching (file transfers are done directly between clients via HTTP), but provides no visible structure that can be exploited for routing purposes.

The earliest version of the Gnutella protocol defined all peers to be truly that, equals in the system regardless of resources available (link speed). Current versions of Gnutella protocol (version 0.6 [3]) employ an "ultrapeer" scheme, effectively implementing a star topology. The elected ultrapeers are connected to each other in the same manner as earlier versions of Gnutella, and other nodes (called leaves) are connected to the ultrapeers. In this manner, ultrapeers act as a proxy to the entire network for their leaf set. This is beginning to introduce structure into the traditionally unstructured Gnutella overlay network.

It seems that a guaranteed multicast is not immediately possible in the Gnutella network without adding an even more rigorous structure to the network and it has been shown that introducing structure to Gnutella greatly reduces the number of messages required [22]. Based on the current direction of the Gnutella protocol specification, we will make the assumption that each ultrapeer represents 75 leaf nodes and has connections to 6 other

ultrapeers in the Gnutella network [3, 5]. Based on a strict interpretation of the Gnutella protocol with ultrapeers enabled, a given peer, in the best case, can reach a maximum of 3,499,199 other peers. This is a best case scenario and assumes that all ultrapeers know distinctly different ultrapeers (i.e. network can be represented with a directed acyclic graph - DAG), and is not likely to be reality. Given this data, we conjecture that there is not an effective solution for building a reliable multicast structure for Gnutella, however it is possible to build one for closely connected localities. This simply allows for the possibility that some users might not be able to reach the multicast group.

In order to implement multicast in Gnutella, the ultrapeers concept is essential. We will prescribe the additional responsibility of managing multicast groups to ultrapeers. Each ultrapeer will manage multicast groups created by its leaf nodes, providing an even distribution given the assumption that leaf nodes are evenly distributed among ultrapeers and that every leaf node is equally likely create multicast groups, thus providing a well distributed load for hosting the multicast groups. Thus, creation of multicast groups is a one hop operation assigning the group to a given node's ultrapeer (or itself if the originating node is an ultrapeer).

If any other node in the network (leaf or ultrapeer) wishes to join the multicast group, that peer sends a join request to its connecting ultrapeer who then floods the network in the same manner as content query requests are processed today. Therefore, if a node's ultrapeer is within six or seven hops of the ultrapeer hosting the multicasts group, the join should be successful. If a join message reaches the multicast groups originating ultrapeer, that ultrapeer responds with a hit response. The multicast group owning ultrapeer keeps track of the joining peer's representative ultrapeer's IP address and port number, and the joining node's ultrapeer must keep track of the fact that the joining node has joined the group. In addition, the joining ultrapeer will remember which node responded with the hit message for direct routing the in the future.

To send a message to a given multicast group, a peer begins by sending a message to its ultrapeer. If the sending peer is know by its ultrapeer to be a member of the multicast group, the send request is forwarded (directly) to the ultrapeer that owns the multicast group. The group owning ultrapeer then forwards the multicast message content to its connected ultrapeers (only if it knows that they represent subscribers), who in turn do the same (with the given TTL). In this scenario, it is possible that the multicast message is routed through ultrapeers that do not represent subscribers, but does not route along eventually dead edges. There is also not a guarantee that a given ultrapeer will not receive any given packet twice, but should discard duplicates as with other Gnutella messages.

The multicast system proposed here as an addition to the Gnutella partially satisfies the base criteria for peer-to-peer multicast networks. Any given multicast group will be allowed to live beyond its originator, but not beyond its originator's ultrapeer. This is because Gnutella does not provide semantics for maintaining content beyond the existence of it's owner, and we have preserved that. Scalability seems to be reasonable in this network as it builds upon the existing Gnutella query flooding process, but does slightly reduce the number of overall messages by taking advantage of the existing of ultrapeers and having them record routes as they are created. For now, we presume that the existing Gnutella network can handle this additional load for certain multicast traffic (stock tickers, chat, and sports scores) but does not seem to offer the efficiencies and bandwidth that would be required (additional experimentation and measurement would be required to fully describe the requirements). Efficiencies are clearly gained when membership is concentrated on a small subset of the ultrapeers that are fewer hops from the message source, and in a worst case scenario a multicast group with one sender and on subscriber could require seven hops to deliver the payload, routing through five ultrapeers who are simply relaying the message and not processing anything. In the situation of Gnutella, *Myriad* could be used for scheduling between the ultrapeers and also for a subnetwork of subscribers that is served by an ultrapeer.

5.3 Multicasting in Structured Peer-to-peer

Structured peer-to-peer networks present a friendlier framework within which we may design application-level multicast protocols. We will concentrate on various networks that implement distributed hash table (DHT) semantics: CAN [51], Chord [62], Pastry [54], and Tulip [11]. Each of these networks builds an overlay network in a different way and we examine how each might provide some usefulness towards designing a multicast protocol for the network.

Since each of these networks provides key based content publishing and lookup, we can assume that a special multicast object will be able to be stored in the system using the well known consistent hash function for that particular network. This allows for consistent storage and ownership of the keys, as well as possible migration of the key when a node leaves the network. Specific qualities of each network will be addressed individually.

5.3.1 Multicasting in CAN

The content-addressable network (CAN) described by Ratnasamy et al. [51] in it's purest form divides a two dimensional space (represented by $[0,1] \times [0,1]$) by splitting the heaviest loaded area (in terms of managed content) in half when a new node joins. Thus when the network is being bootstrapped, the first node owns the entire space and when a second node joins they each represent half of the space. Items are located in the network by hashing the object to a unique coordinate and making the node whose area the point falls in responsible for that object. Routing is then done by relaying messages from a given node to that nodes immediate neighbors only. The initial design of CAN assigned the nodes space in the order they arrived, but attempts have been made to make the representative space more accurately reflect the distance between nodes as determined by round trip times (RTT) between the nodes, thus providing some degree of locality awareness. The average routing path in a CAN network is of length $\frac{d}{4}n^{\frac{1}{d}}$ there n is the number of nodes in the network (equal partitions)

and d is the dimensionality of the space [51]. If we reduce to a strictly two dimensional network, the average routing path becomes $\frac{\sqrt{n}}{2}$.

While CAN provides the necessary functionality of being able to label and consistently address multicast groups, the ability to efficiently route multicast messages is not inherent to the network. We can propose a system for CAN by where a node subscribing to a multicast group will leave that information with every node along the path from the joiner to the node owning the multicast group. This will allow the reverse route to be traversed when delivering the multicast message. Any receiving a request duplicates the packet sends it to all it's subscribing neighbors. This organization should minimize the number of physical links that cary the same multicast packet (assuming nodes enter the network using the locality awareness). The chosen value for d affects how quickly a given multicast message traverses the network. Higher dimensionality (higher values for d) all a message to be duplicated more times, thus potentially reaching more subscribers in less hops. For example when $d = 1$ any node receiving the message can only forward to at most 1 additional node, for $d = 2$ the message can be duplicated 3 times, and for $d = 3$ the message can be duplicated up to 5 times. This choice for the parameter d will determine the effectiveness of multicast message delivery in CAN. CAN lends itself to reverse path forwarding solutions and does not seem to easily adaptable to the inclusion of an algorithm such as *Myriad* for multicast route calculation.

5.3.2 Multicasting in Chord and Pastry

Chord [62] provides the standard DHT service of keyed lookup, but organizes the logical layout of nodes differently than CAN. In a Chord network, nodes are organized in a 1 dimensional space (a line from 0 to x with wrap around - usually drawn as a circle). Again, because of the DHT semantics, labeling and assigning ownership to a multicast group is trivial as long as we utilize to a well known consistent hash function for the labeling.

Since Pastry [54] is closely related to Chord, we will consider the Scribe [20] application-level multicasting infrastructure already built for Pastry. Scribe does well in respect to route maintenance link stress but falls short in the efficient delivery of multicast messages. Scribe functions by reversing the path of normal Pastry traffic and forms trees by taking the union of these paths and reversing them at the join points (a technique known as reverse path forwarding). The problem is that locality is not taken into consideration and two computers sitting next to each other might relay a message through a third host that is much farther away. The lower dimensionality and loss of locality inhibits efficient scheduling of multicast routes on Chord and Pastry.

Reverse path forwarding has some ill effects including the inability to guarantee an out-degree bound at any particular node in the overlay network. Borg is hybrid multicast distribution scheme [68] that is also built upon Pastry and alleviates some of the impact when using reverse-path forwarding. As with Scribe, Borg can not guarantee that out-degree constrains are satisfied. This further complicates the distribution of real-time multimedia streaming content with a high bandwidth and low latency requirements. So we consider the possibility of implementing the *Myriad* algorithm for a unit circle DHT such as Pastry.

In order to execute *Myriad* more information about the network is needed than what is normally provided in a Pastry routing table. The obvious solution is to simply increase the routing table information required, which would cause routing table to become to large to manage for a sufficiently large network. Instead we consider a hybrid approach allowing the network to be segmented into an artificial hierarchy. This hierarchical approach will allow for full routing information to be exchanged between subsets of nodes in the network, rather than all members of the network.

For Pastry, it is possible to implement *Myriad* by expanding either the routing table or the neighborhood set. In order to guarantee multicasting ability, any node in the network has to be able to initiate a multicast stream that can reach the rest of the network. We therefore

propose a hierarchy that is based on the node ID structure. While it might be better for overall latency to leverage the locality properties of the neighborhood set, it becomes difficult to build a reliable hierarchical structure that guarantees inclusion of all participants.

Therefore we propose segmentation of the network upon logical node ID strings. Each participant in the pastry network would be required to maintain routing information about some fraction of the node ID ring from their ID forward. This allows for the calculation of multicast routes using locality information within a segment of the node ID circle. Meta-scheduling would need to be done to ensure that each segment of the circle is covered, using standard Pastry routing.

The only remaining item to allow for *Myriad* based scheduling is the determining of which nodes in any given segment wish to participate or have subscribed to a selected multicast group. This is additional information that would need to be brought into the routing table and exchanged on refreshes. It would be possible to use the same techniques as discussed in the next section.

5.3.3 Multicasting in Tulip

The Tulip peer-to-peer overlay network described in [11] introduces a well described and mathematically sound overlay network that formally provides stretch-2 routing, and maximum of 2 network hops for routing [10, 11].

Stretch-2 routing has a maximum ratio of 2 for cost the routing path from source to destination as compared to the minimum (optimal) routing path. Tulip achieves the ability to do 2 hop routing by maintaining a high out degree of $O(2\sqrt{n}\log n)$, where n is the number of nodes in the network. Here, out degree represents information that is kept current in any given participant's routing table. This high redundancy allows for routing around failures with minimal overhead, at the expense of higher memory usage at each peer, but still within an acceptable range for memory consumption.

There is a subset of the formal foundations of Tulip whose definitions are useful when constructing the multicast portion of the overlay. The most important of which is the distance function $d(r, t)$ that represents the communication cost from source node r to the destination node t. The vicinity of any given node $u \in V$ is defined as the $\sqrt{n}\ \log n$ closest nodes according to $d()$ [11]. It is necessary to measure round trip times and estimate bandwidth to avoid the problem of users misrepresenting their bandwidth to discourage other hosts to download from them [38].

In Tulip, every node is assigned a label that is the result of a consistent hash function. The first $\log \sqrt{n}$ bits of the hash are taken to be the *color* of the node, this is the color function $c(u)$. Because of relatively even distribution of the SHA-1 hash function, this leads to $\sqrt{n}$ color sets with at most $2\sqrt{n}$ nodes in each color set [11]. Every node maintains 2 sets of information; (1) the vicinity list consists of the closest $\log n$ nodes of each color and (2) the color list, consisting of information about ever other node of the same color $c(u) = c(x) \mid \forall x \in V$.

Tulip's keyed lookup routing is for objects whose exact key is known, which is the case for multicast groups since their identifier (or group name) is known. Each object is stored by assigning it a color (through the same method as coloring peer nodes) and locating it on the host of the same color with the longest matching label prefix (similar to the key assignment in Pastry [54]). Locating the stored object can be done in one or two hops, depending on the color of the object and the color of the requesting node. The first hop will go directly to the object (if the object is the same color as the requesting object), otherwise the requesting node will route the request to the closest node (according to the prefix of their hashes) that is the same color as the object (the normal Tulip routing scheme). The second routing hop goes directly to the node holding the object since the holder will be the same color as the receiver of the first routing hop and all nodes of the same color maintain information about each other.

In order to provide multicasting that uses *Myriad* to calculate routes, Tulip needs to be modified so that multicast is natively supported without additional support at the application level. In addition to the normal routing information kept at each host, nodes must maintain certain data to keep track of multicast routes. Each node begins by maintaining its own subscription list which consists of a version number and 1024 bytes of subscription data. The subscription data portion is a representation of a counting Bloom filter (where the multicast group IDs are mapped into the filter). By using a Bloom filter, we are accepting the possibility that a false positive might be reported when testing for membership in a multicast group. When the subscription information is shared, an alternative representation (using non-counting Bloom Filter) is used, this allows the same subscription data for a single node to be stored in 128 bytes (and is read-only). The power of nodes in the network would affect their ability to use either the counting filter or non-counting filter.

During the color list refresh [11], each node sends accompanying multicast subscription version number that is known for each node. If the node receiving the data sees a newer version for a given node, it will request the subscription data for that specific node. This will put initial stress of a node joining the network, but provides a much greater benefit for the network as a whole since the subscription data for all nodes is not sent on every refresh request. Nodes that enter a steady state as far as their subscriptions go, will see almost no additional overhead during the color list refresh. This mechanism also helps to prevent nodes from creating a fake multicast subscription filter for other nodes by always retrieving the correct copy of the data from the owning host.

An additional data structure is also passed around the network for each host in the color group. This consists of a version number and a vector for a given node contains scaled (0-255) latency values from that node to all other nodes of the same color, thus the length of the vector is $2\sqrt{n}$ and is broken up into individual elements and passed along with the known routing information for that node. The overhead introduced by this is $(2\sqrt{n})^2$ in

terms of communication costs. This allows any given node in a color group to calculate the all pairs shortest paths and schedule the color group multicast tree.

After $\sqrt{n}$ color list refreshes, any given node will have all of the data necessary needed to schedule any given multicast route (local to that node's color group). Before the data has reached a steady state, messages can still be routed, but the tree may need to be rebuilt later. As a node disappears from the network its multicast routing information disappears along with it's standard routing information, no special communication is needed for this.

The source node is the orchestrator of the first level of scheduling (where the source node is the node which stores the identify of the group - not the actual sender of the message). This source node will seed the message into each color group where internal color group scheduling will take place.

Since color groups are used as the primary network segmentation device, the hierarchy imposed is 2 level. Requiring the source node to physically disseminate the message into each color group could be costly, so *Myriad* would need to be run over a set contains the source node and one representative from each color group. Each source node will maintain multicast routing information through the first node of each color in it's vicinity list.

Once the message has been seeded into each color group, these network segments can be scheduled. Since subscription information is shared among members of a color group, any node receiving a multicast communication from outside the color group has the authority and information to use *Myriad* to construct a multicast delivery tree for the color group.

5.4 Summary

While this work has provided a new algorithm for the construction of minimum average-latency degree-bounded directed spanning trees, it is important to assess the potential impact on real-world systems. Since the algorithmic solution for *Myriad* has been reduced to a graph

problem, it should be possible to use the algorithm to schedule multicast communications on any peer-to-peer overlay network where the necessary routing information is either available or can be made available to the application.

Chapter 6

Conclusion

In this body of research we have examined the problem of multicast based group communication through algorithm development in relation to the *degree-constrained minimum average-latency spanning tree problem* (*DC-MAL* as defined in Section 1.1). This, and continued research on this problem is warranted because of the large body of existing and future multicast applications [1, 24, 26, 40, 49, 58, 61, 64]. We believe that the research in this area will continue to grow, giving way to practical systems that are used seamlessly alongside existing Internet applications.

Since IP multicast is facing slow adoption, the consensus of the community seems to point to application level multicasting as an enabling technology [18, 20, 23, 24, 45, 50]. Our research is positioned in response to this trend of delivering multicast content through peer-to-peer overlay networks and aims to make use existing (or expanded) routing information available at end systems participating in the peer-to-peer overlay network. This extended routing information and accompanying locality calculations can be used to feed into a centralized algorithm such as *Myriad*, which is presented here.

When organizing nodes in a network with full routing information, it is possible to generalize to a graph structure. This allows study of *DC-MAL* in relation to multicast com-

munication. The constraint of *minimum average-latency* is motivated by the desire to reach all subscribers in the minimum amount of time possible. Out-degree (degree-bounded) constraints are imposed by the physical capabilities of the network in relation to the bandwidth requirements of the multicast communication stream. Our work has also shown the impact of requiring a compact tree to be formed for the multicast routes [41]. Reducing the average number of hops required to reach all subscribes in turn reduces the potential points of failure along the communication paths.

Myriad provides an approximate solution to *DC-MAL* by adhering to heuristic constraints developed through extensive experimentation and visualization of new and existing algorithms. *Myriad* operates on the property that any node in a multicast tree should connect to the root through a parent node that is closer to the root than itself. If a compact tree is desired, this algorithm performs well for uniformly random network distributions. For non-compact trees, *Myriad* consistently outperforms other algorithms in the area (Section 4.5).

We believe that this research can serve as the foundation for future research in the area of application level multicasting. A first step is to move *Myriad* out of the simulator and deploy the technology as a real world multicast system positioned for delivery of (near) real time multimedia content. Content with high bandwidth requirements will naturally yield low out-degree constraints at end systems and extends well from the experiments conducted in the examination of *Myriad*.

The use of application level multicasting can also allow for participation in multicast communication by end systems that are obstructed from public internet communication by network address translation (NAT) [8], an item that has been somewhat ignored [66]. Additional research is needed to construct systems capable of originating and delivering multicast communication involving hosts behind a network address translator. We believe

that *Myriad* is a candidate algorithm for the organization of multicast overlay networks positioned in this space.

Our plans for future work on *Myriad* include exploring methods for constructing a distributed version of the algorithm. Initial work on this problem indicates that we might be able to employ the same heuristic, of only connecting through vertices that are closer to the root, in a distributed manner. In this model, each vertex would be responsible for picking its own parent out of its neighbor set, using coordinate approximation to select a parent that is closer to the root than itself. In order to handle the dynamism and churn of a running system, we consider an incentive system, providing a vertex with better placement in the tree if they agree to host children in the same multicast stream. Given the results for centralized experimentation with *Myriad*, we believe that a distributed algorithm formulated in a similar fashion can provide competitive results for solving DC-MAL and powering application-level multicasting on the Internet.

By introducing the new *Myriad* algorithm and investigating the problem of multicasting and the related *minimum average-latency degree-bounded directed spanning tree problem* we have contributed to the field and advanced a position and framework within to undertake future research on this subject. A common vocabulary for comparing and classifying multicast tree has been proposed (Section 1.2) in order to fill an apparent gap in the discourse on multicast and degree-bounded spanning trees. This work also demonstrates through empirical evaluation the performance improvements when comparing *Myriad* to other competitive algorithms aimed at solving the same or closely related problems. Lastly we show experimental results that indicate a depth-latency tradeoff when constructing multicast trees [41] and believe that this insight will aid in future development and advancement in graph theory and multicasting technologies.

Bibliography

[1] An internet multicast system for the stock market. *ACM Trans. Comput. Syst.*, 19(3): 384–412, 2001. ISSN 0734-2071. doi: http://doi.acm.org/10.1145/380749.380771.

[2] http://www.gnutella.com/, visited october 7, 2006, .

[3] http://rfc-gnutella.sourceforge.net/src/rfc-0_6-draft.html, visited october 7, 2006, .

[4] http://www.graphviz.org/, checked march 23, 2007.

[5] URL `http://www.limewire.com/english/content/glossary.shtml`.

[6] Distance vector multicast routing protocol, http://www.faqs.org/rfcs/rfc1075.html, visited october 7th, 2006, .

[7] Host extensions for ip multicasting, http://www.faqs.org/rfcs/rfc1112.html, visited october 7th, 2006, .

[8] Network address translator (nat) - friendly application design guidelines, http://www.faqs.org/rfcs/rfc3235.html, .

[9] Internet group management protocol, version 3, http://www.faqs.org/rfcs/rfc3376.html, visited october 7th, 2006, .

[10] I. Abraham, C. Gavoille, D. Malkhi, N. Nisan, and M. Thorup. Compact name-independent routing with minimum stretch. In *SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 20–24, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-840-7. doi: http://doi.acm.org/10.1145/1007912.1007916.

[11] I. Abraham, A. Badola, D. Bickson, D. Malkhi, S. Maloo, and S. Ron. Practical locality-awareness for large scale information sharing. In *4th International Workshop on Peer-To-Peer Systems*, 2005.

[12] Foto N. Afrati, Stavros S. Cosmadakis, Christos H. Papadimitriou, George Papageorgiou, and Nadia Papakostantinou. The complexity of the travelling repairman problem. *ITA*, 20(1):79–87, 1986.

[13] Mark Allman. On the performance of middleboxes. In *IMC '03: Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 307–312, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-773-7. doi: http://doi.acm.org/10.1145/948205.948246.

[14] Stephanos Androutsellis-Theotokis and Diomidis Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Comput. Surv.*, 36(4):335–371, 2004. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/1041680.1041681.

[15] Fred S. Annexstein, Kenneth A. Berman, and Mihajlo A. Jovanovic. Latency effects on reachability in large-scale peer-to-peer networks. In *SPAA '01: Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, pages 84–92, New York, NY, USA, 2001. ACM Press. ISBN 1-58113-409-6. doi: http://doi.acm.org/10.1145/378580.378594.

[16] S. Banerjee, C. Kommareddy, K. Kar, B. Bhattacharjee, and S. Khuller. Construction of an efficient overlay multicast infrastructure for real-time applications. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies. IEEE*, volume 2, pages 1521–1531, 2003.

[17] Suman Banerjee, Bobby Bhattacharjee, and Christopher Kommareddy. Scalable application layer multicast. In *SIGCOMM '02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 205–217, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-570-X. doi: http://doi.acm.org/10.1145/633025.633045.

[18] Suman Banerjee, Seungjoon Lee, Bobby Bhattacharjee, and Aravind Srinivasan. Resilient multicast using overlays. In *SIGMETRICS '03: Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 102–113, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-664-1. doi: http://doi.acm.org/10.1145/781027.781041.

[19] Avrim Blum, Prasad Chalasani, Don Coppersmith, Bill Pulleyblank, Prabhakar Raghavan, and Madhu Sudan. The minimum latency problem. In *STOC '94: Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, pages 163–171, New York, NY, USA, 1994. ACM Press. ISBN 0-89791-663-8. doi: http://doi.acm.org/10.1145/195058.195125.

[20] M. Castro, P. Druschel, A-M. Kermarrec, and A. Rowstron. Scribe: A large-scale and decentralised application-level multicast infrastructure. In *IEEE Journal on Selected Areas in Communications (JSAC) (Special issue on Network Support for Multicast Communications)*, 2002.

[21] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. Splitstream: high-bandwidth multicast in cooperative environments. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating*

systems principles, pages 298–313, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-757-5. doi: http://doi.acm.org/10.1145/945445.945474.

[22] Miguel Castro, Manuel Costa, and Antony Rowstron. Should we build gnutella on a structured overlay? *SIGCOMM Comput. Commun. Rev.*, 34(1):131–136, 2004. ISSN 0146-4833. doi: http://doi.acm.org/10.1145/972374.972397.

[23] Woan Sun Chang and Robert Simon. End-host multicasting in support of distributed real-time simulation systems. In *ANSS '04: Proceedings of the 37th annual symposium on Simulation*, page 7, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2110-X.

[24] YangHua Chu, Sanjay G. Rao, and Hui Zhang. A case for end system multicast (keynote address). In *SIGMETRICS '00: Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 1–12, New York, NY, USA, 2000. ACM Press. ISBN 1-58113-194-1. doi: http://doi.acm.org/10.1145/339331.339337.

[25] Luís Henrique M. K. Costa, Serge Fdida, and Otto Carlos M. B. Duarte. Incremental service deployment using the hop-by-hop multicast routing protocol. *IEEE/ACM Trans. Netw.*, 14(3):543–556, 2006. ISSN 1063-6692. doi: http://dx.doi.org/10.1109/TNET.2006.876157.

[26] Yi Cui and Klara Nahrstedt. High-bandwidth routing in dynamic peer-to-peer streaming. In *P2PMMS'05: Proceedings of the ACM workshop on Advances in peer-to-peer multimedia streaming*, pages 79–88, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-248-8. doi: http://doi.acm.org/10.1145/1099384.1099395.

[27] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: a decentralized network coordinate system. In *SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 15–26, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-862-8. doi: http://doi.acm.org/10.1145/1015467.1015471.

[28] S. E. Deering. Multicast routing in internetworks and extended lans. In *SIGCOMM '88: Symposium proceedings on Communications architectures and protocols*, pages 55–64, New York, NY, USA, 1988. ACM Press. ISBN 0-89791-279-9. doi: http://doi.acm.org/10.1145/52324.52331.

[29] Stephen E. Deering and David R. Cheriton. Multicast routing in datagram internetworks and extended lans. *ACM Trans. Comput. Syst.*, 8(2):85–110, 1990. ISSN 0734-2071. doi: http://doi.acm.org/10.1145/78952.78953.

[30] Stephen Edward Deering. *Multicast routing in a datagram internetwork*. PhD thesis, Stanford University, 1991.

[31] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[32] C. Diot, B. N. Levine, B. Lyles, H. Kassem, and D. Balensiefen. Deployment issues for the ip multicast service and architecture. In *IEEE Network Magazine*, January/February 2000.

[33] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[34] Aditya Ganjam and Hui Zhang. Connectivity restrictions in overlay multicast. In *NOSSDAV '04: Proceedings of the 14th international workshop on Network and operating systems support for digital audio and video*, pages 54–59, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-801-6. doi: http://doi.acm.org/10.1145/1005847.1005860.

[35] Michael R. Garey and David S. Johnson. *Computer and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[36] Michel Goemans and Jon Kleinberg. An improved approximation ratio for the minimum latency problem. In *SODA '96: Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 152–158, Philadelphia, PA, USA, 1996. Society for Industrial and Applied Mathematics. ISBN 0-89871-366-8.

[37] Björn Grönvall, Ian Marsh, and Stephen Pink. A multicast-based distributed file system for the internet. In *EW 7: Proceedings of the 7th workshop on ACM SIGOPS European workshop*, pages 95–102, New York, NY, USA, 1996. ACM Press. doi: http://doi.acm.org/10.1145/504450.504469.

[38] P. Krishna Gummadi, Stefan Saroiu, and Steven D. Gribble. A measurement study of napster and gnutella as examples of peer-to-peer file sharing systems. *SIGCOMM Comput. Commun. Rev.*, 32(1):82–82, 2002. ISSN 0146-4833. doi: http://doi.acm.org/10.1145/510726.510756.

[39] M. D. Hamilton, P. McKee, and I. Mitrani. Distributed systems with different degrees of multicasting. In *WOSP '02: Proceedings of the 3rd international workshop on Software and performance*, pages 68–74, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-563-7. doi: http://doi.acm.org/10.1145/584369.584379.

[40] Mohamed Hefeeda, Ahsan Habib, Boyan Botev, Dongyan Xu, and Bharat Bhargava. Promise: peer-to-peer media streaming using collectcast. In *MULTIMEDIA '03: Proceedings of the eleventh ACM international conference on Multimedia*, pages 45–54, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-722-2. doi: http://doi.acm.org/10.1145/957013.957022.

[41] Michael T. Helmick and Fred S. Annexstein. Depth-latency tradeoffs in multicast tree algorithms. In *AINA-07: Proceedings of The IEEE 21st International Conference on Advanced Information Networking and Applications*, 2007.

[42] Jochen Kanemann, Asaf Levin, and Amitabh Sinha. Approximating the degree-bounded minimum diameter spanning tree problem. *Algorithmica*, 41(2):117–129, 2004.

[43] James F. Kurose and Keith W. Ross. *Computer Networking: A top-down approach featuring the internet*. Addison Wesley, third edition, 2005.

[44] Tetsuya Kusumoto, Yohei Kunichika, Jiro Katto, and Sakae Okubo. Tree-based application layer multicast using proactive route maintenance and its implementation. In *P2PMMS'05: Proceedings of the ACM workshop on Advances in peer-to-peer multimedia streaming*, pages 49–58, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-248-8. doi: http://doi.acm.org/10.1145/1099384.1099392.

[45] Minseok Kwon and Sonia Fahmy. Topology-aware overlay networks for group communication. In *NOSSDAV '02: Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video*, pages 127–136, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-512-2. doi: http://doi.acm.org/10.1145/507670.507688.

[46] Frank Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann Publishers, 1991.

[47] Madhav V. Marathe, R. Ravi, Ravi Sundaram, S. S. Ravi, Daniel J. Rosenkrantz, and III Harry B. Hunt. Bicriteria network design problems. *J. Algorithms*, 28(1):142–171, 1998. ISSN 0196-6774. doi: http://dx.doi.org/10.1006/jagm.1998.0930.

[48] Martin Modahl, Bikash Agarwalla, Gregory Abowd, Umakishore Ramachandran, and T. Scott Saponas. Toward a standard ubiquitous computing framework. In *Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing*, pages 135–139, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-951-9. doi: http://doi.acm.org/10.1145/1028509.1028515.

[49] Walid Mostafa and Mukesh Singhal. A reliable multicast session protocol for collaborative continuous-feed applications. In *SAC '97: Proceedings of the 1997 ACM symposium on Applied computing*, pages 35–39, New York, NY, USA, 1997. ACM Press. ISBN 0-89791-850-9. doi: http://doi.acm.org/10.1145/331697.331706.

[50] Andrea Passarella, Franca Delmastro, and Marco Conti. Xscribe: a stateless, cross-layer approach to p2p multicast in multi-hop ad hoc networks. In *MobiShare '06: Proceedings of the 1st international workshop on Decentralized resource sharing in mobile computing and networking*, pages 6–11, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-558-4. doi: http://doi.acm.org/10.1145/1161252.1161255.

[51] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer commu-*

nications, pages 161–172, New York, NY, USA, 2001. ACM Press. ISBN 1-58113-411-8. doi: http://doi.acm.org/10.1145/383059.383072.

[52] R. Ravi. Rapid rumor ramification: approximating the minimum broadcast time. In *Foundations of Computer Science, 1994 Proceedings., 35 Annual Symposium*, pages 202 – 213, 1994.

[53] R. Ravi, Madhav V. Marathe, S. S. Ravi, Daniel J. Rosenkrantz, and Harry B. Hunt III. Approximation algorithms for degree-constrained minimum-cost network-design problems. *Algorithmica*, 31(1):58–78, 2001.

[54] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, 2001.

[55] Dan Rubenstein and Sambit Sahu. Can unstructured p2p protocols survive flash crowds? *IEEE/ACM Trans. Netw.*, 13(3):501–512, 2005. ISSN 1063-6692. doi: http://dx.doi.org/10.1109/TNET.2005.845530.

[56] Sartaj Sahni and Teofilo Gonzalez. P-complete approximation problems. *J. ACM*, 23 (3):555–565, 1976. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/321958.321975.

[57] D. Sandler, A. Mislove, A. Post, and P. Druschel. Feedtree: Sharing web micronews with peer-to-peer event notification. In *Proceedings of the 4th International Workshop on Peer-to-Peer Systems (IPTPS'05)*, Ithaca, New York, February 2005.

[58] Eric Setton, Jeonghun Noh, and Bernd Girod. Rate-distortion optimized video peer-to-peer multicast streaming. In *P2PMMS'05: Proceedings of the ACM workshop on Advances in peer-to-peer multimedia streaming*, pages 39–48, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-248-8. doi: http://doi.acm.org/10.1145/1099384.1099390.

[59] S. Shi and J.S. Turner. Routing in overlay multicast networks. In *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1200 – 1208, 2002.

[60] Sherlia Y. Shi, Jonathan S. Turner, and Marcel Waldvogel. Dimensioning server access bandwidth and multicast routing in overlay networks. In *NOSSDAV '01: Proceedings of the 11th international workshop on Network and operating systems support for digital audio and video*, pages 83–91, New York, NY, USA, 2001. ACM Press. ISBN 1-58113-370-7. doi: http://doi.acm.org/10.1145/378344.378357.

[61] Hwangjun Song, Dong Sup Lee, and Hyung Rai Oh. Application layer multicast tree for real-time media delivery. In *Computer Communications, Volume 29, Issue 9, ICON 2004 - 12th IEEE International Conference on Network 2004*, 2004.

[62] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003. ISSN 1063-6692. doi: http://dx.doi.org/10.1109/TNET.2002.808407.

[63] D. A. Tran, K. A. Hua, and T. T. Do. Scalable media streaming in large peer-to-peer networks. In *MULTIMEDIA '02: Proceedings of the tenth ACM international conference on Multimedia*, pages 247–250, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-620-X. doi: http://doi.acm.org/10.1145/641007.641056.

[64] Song Ye and Fillia Makedon. Collaboration-aware peer-to-peer media streaming. In *MULTIMEDIA '04: Proceedings of the 12th annual ACM international conference on Multimedia*, pages 412–415, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-893-8. doi: http://doi.acm.org/10.1145/1027527.1027625.

[65] Chun-Chao Yeh and Lin Siong Pui. On the frame forwarding in peer-to-peer multimedia streaming. In *P2PMMS'05: Proceedings of the ACM workshop on Advances in peer-to-peer multimedia streaming*, pages 1–10, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-248-8. doi: http://doi.acm.org/10.1145/1099384.1099386.

[66] Chad Yoshikawa, Brent Churn, Amin Vahdat, Fred Annexstein, and Ken Berman. The lonely nated node. In *11th ACM SIGOPS European Workshop*, 2004.

[67] Ellen W. Zegura, Kenneth L. Calvert, and Samrat Bhattacharjee. How to model an internetwork. In *IEEE Infocom*, volume 2, pages 594–602, San Francisco, CA, March 1996. IEEE.

[68] Rongmei Zhang and Y. Charlie Hu. Borg: a hybrid protocol for scalable application-level multicast in peer-to-peer networks. In *NOSSDAV '03: Proceedings of the 13th international workshop on Network and operating systems support for digital audio and video*, pages 172–179, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-694-3. doi: http://doi.acm.org/10.1145/776322.776349.

[69] Shelley Q. Zhuang, Ben Y. Zhao, Anthony D. Joseph, Randy H. Katz, and John D. Kubiatowicz. Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination. In *NOSSDAV '01: Proceedings of the 11th international workshop on Network and operating systems support for digital audio and video*, pages 11–20, New York, NY, USA, 2001. ACM Press. ISBN 1-58113-370-7. doi: http://doi.acm.org/10.1145/378344.378347.