Efficient and Secure Media Delivery

Justin Goshi

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Washington

2004

Program Authorized to Offer Degree: Computer Science & Engineering

University of Washington

Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Justin Goshi

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Chair of Supervisory Committee:

Richard E. Ladner

Reading Committee:

Richard E. Ladner

John Zahorjan

Radha Poovendran

Date: August 12, 2004

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to Bell and Howell Information and Learning, 300 North Zeeb Road, Ann Arbor, MI 48106-1346, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature Justin Gosh

Date August 12, 2004

University of Washington

Abstract

Efficient and Secure Media Delivery

by Justin Goshi

Chair of Supervisory Committee:

Professor Richard E. Ladner
Computer Science & Engineering

With the increasing power of computers and communication networks, interest has grown in a whole new class of multimedia applications such as media-on-demand, subscription services to live sporting events, and online distributed games. Being both popular and bandwidth-intensive, these applications can impose significant load on both the network and individual servers. As these applications continue to mature, companies are beginning to sell access to high-quality versions of these applications. This makes it necessary to protect the data so that only paying customers have access. One way to accomplish this is to encrypt the data with a group key held only by the set of paying customers. When this set changes, the group key must be changed and distributed in an efficient and secure way.

In our work we study solutions to improve the efficiency and security of media delivery. Stream merging is a technique for efficiently delivering popular media using multicast and client buffers. We perform a comprehensive comparison of the recently proposed stream merging solutions to gain a deeper understanding of their complexity and performance trade-offs. We also show how to apply stream merging to the live broadcast with time-shifting model. This is a model where a client can join the broadcast at time t and receive the broadcast of time $t - w$ for some offset parameter $w \geq 0$. Finally, we study the problem of efficient and secure group key distribution.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

GLOSSARY

ANCESTOR WEIGHT OF U: the sum of the degrees of all nodes in a key tree on the path from u to the root.

ARTIFICIAL ARRIVAL TIME: the time that client X would have arrived had it requested segment 0.

AUXILIARY KEY: a key shared by a subset of group members.

BACKWARD SECURITY: a new member is not able to decrypt past group communications.

B-TREE OF ORDER T: a tree where all internal nodes except the root have $[\lceil t/2 \rceil, t]$ children (where $t \geq 3$), and all leaves are at the same depth in the tree. The root can have $[2,t]$ children.

CLIENT RECEIVE PROGRAM: the information from a stream merging server informing a client of which streams to listen to and for how long.

COMMUNICATION COST: the number of encrypted messages required by the key server to update the keys due to group membership change.

FORWARD SECURITY: a departing member is not able to decrypt future group communications.

GLOBAL RESTRUCTURING: key tree restructuring that requires a global view of the tree structure.

GROUP KEY: a secret shared by all members of a secure group that is used to encrypt group data.

HEIGHT-BALANCED NODE: a node whose subtrees differ in height by at most 1.

HEIGHT-BALANCED 2-K TREE: a tree where all internal nodes have degree $2 \leq d \leq k$, and all nodes are height-balanced.

INDIVIDUAL KEY: a key known to only one group member (shared with the key server).

KEY SERVER: the server responsible for the secure management and distribution of keys used to maintain group security.

KEY TREE: a logical data structure used to represent the set of keys held by each group member.

LIVE BROADCAST WITH TIME-SHIFTING: a system that allows clients to arrive at time t and ask to join the broadcast at time $t - w$ for some offset value $w \geq 0$.

LOCAL RESTRUCTURING: key tree restructuring using rules that can be applied while considering only a small portion of the tree.

MEDIA-ON-DEMAND: the demand by clients to playback, view, listen, or read various types of media such as video, audio, or large files with little or no start-up delay and with no interruptions.

MERGE TREE: a tree data structure used to represent stream merging patterns.

MULTICAST KEY DISTRIBUTION: the problem of managing and distributing the group and auxiliary keys for a secure group.

NODE WEIGHT: the tree weight of the sub-tree rooted at the node.

OFFLINE ALGORITHM: an algorithm that operates with full knowledge of all inputs.

ONLINE ALGORITHM: an algorithm that must process inputs as they arrive (without knowledge of the future).

OPTIMAL TREE WEIGHT FOR N MEMBERS: the minimum tree weight over all possible trees consisting of n members.

PREORDER TRAVERSAL PROPERTY: a property of merge trees where a preorder traversal yields the client arrival times in order.

RECEIVE-TWO MODEL: a stream merging model where a client can receive data from at most two streams simultaneously.

RE-KEYING: to change keys on group membership change to ensure that only the remaining members hold the group key.

RESTRUCTURING COST: the additional cost required to dynamically maintain a desired key tree structure.

START-UP DELAY: the delay between the time that a client requests a media object and the time that the client starts receiving the media object.

STREAM LENGTH: the length of time that a stream is active before it is not needed by any clients.

STREAM MERGING: a technique for efficient media delivery that requires clients to simultaneously receive two or more streams and buffer data for future use.

TIME-SEGMENT DIAGRAM: a visual diagram for representing stream merging patterns.

TOTAL BANDWIDTH: the sum of all stream lengths required by a stream merging algorithm to satisfy a set of client requests.

TREE STRUCTURE COST: the cost to update a key tree if we simply add or delete the specified node without worrying about trying to maintain a specific tree structure.

TREE WEIGHT: the maximum ancestor weight of any node in a given key tree.

WEIGHT-BALANCED NODE: a node whose subtrees differ in node weight by at most 1.

WEIGHT-BALANCED TREE: a tree where all nodes are weight-balanced.

ACKNOWLEDGMENTS

This dissertation would not have been possible without the help of many people. First, I would like to express my sincere appreciation to the Department of Computer Science and Engineering for their support, and especially to Professor Richard E. Ladner for his great advising over the past four years. This dissertation would never have been completed without his constant guidance and direction. He provided inspiration and helped me develop the ability and confidence to finish my dissertation. I would like to thank Professor Amotz Bar-Noy for the many projects we collaborated on, and especially for his ideas and inspiration regarding the live broadcast with time-shifting model. I would like to thank Professor Eve A. Riskin for all her help and support in getting me started with research early in my graduate school career. I would like to thank Professor John Zahorjan and Professor Radha Poovendran for providing great feedback and helping to improve my dissertation. Finally, I would like to thank my family and friends. I would not have made it this far without their encouragement and support.

DEDICATION

To my parents, Clifford and Kay Goshi, for their constant encouragement and support over the years. They stood by me through all the tough times, and never gave up on me. Also to my wife Tara, who believed in me and provided me with emotional and material support during my rough first few years in graduate school.

Chapter 1

INTRODUCTION

With the explosive growth of the Internet and the increasing power of computers and communication networks, interest has grown in a whole new class of multimedia applications. Some examples that are rapidly growing in popularity are media-on-demand, subscription services to live and on-demand media, and online distributed games.

The popularity of multimedia applications coupled with their high bandwidth requirements can impose significant load on both the network and individual servers. One technique to reduce the load on individual servers is to cache content at various nodes in the network [44]. Another technique is to replicate servers with the same content at various locations. Both of these approaches attempt to achieve better scaling by decentralizing the location of the media object. An alternative to caching and replication is to use multicast [20] and client buffers. This is the technique we use in our work. Although multicast is supported by IPv4, it is not currently deployed over the global Internet. Despite this, we believe that the assumption of multicast support is already valid in certain situations. For example, enterprise and local area networks (LANs) either have multicast support or can deploy it without too much trouble, the MBone (multicast backbone) uses tunneling to enable multicast communication over the public Internet, and other application level multicast solutions that use tunneling have also been developed [42]. Finally, enabling full multicast support from the Internet is made more likely by techniques like those described in this paper that can use it to dramatically reduce individual server and network load.

As multimedia applications continue to mature, companies are beginning to sell access to high-quality versions of these applications. For example, Major League Baseball offers a subscription service to view their baseball games over the Internet, and RealNetworks

Inc. [1] offers subscription services that provide access to movies and songs. This makes it necessary to protect the data so that only paying customers have access. One way to accomplish this is to encrypt the data with a key held only by the set of paying customers. When this set changes, the shared key must be changed and distributed in an efficient and secure way. This is usually accomplished for point-to-point communications by using a unicast security protocol such as SSL [26] to perform mutual authentication and to establish a shared key. However, this technique does not scale to deal with groups because of the need to establish the shared key with all members of the group, coupled with the fact that the group can be large and highly dynamic.

The above discussion outlined two important issues for media delivery. First, the server bandwidth can quickly become a bottleneck under heavy load. Second, securing data delivery is important. Thus, our work focuses on the problem of providing solutions for the efficient and secure delivery of media. Before moving on, it is important to distinguish between offline and online solutions. An *offline algorithm* is one that can consider all of its inputs when making decisions. While this type of algorithm cannot be used in practice, it can be used to derive an optimal offline solution, which provides insight into the best possible performance of an algorithm. An *online algorithm* is one that must make decisions as the inputs arrive (with no knowledge of the future), and can be used in the actual implementation of real systems.

1.1 Efficient Media Delivery

Stream merging [24, 22, 7, 6, 17, 14, 15] is a technique for efficient media delivery that is built on top of multicast. It reduces the server bandwidth required to satisfy the clients requesting a particular media object. This is accomplished by having the clients simultaneously receive two or more streams, and by having them buffer data for future use. The most commonly used model is called the *receive-two model* where clients listen to two streams simultaneously. As clients buffer *future data* (data not used for immediate viewing purposes) while at the same time receiving data for immediate use, the server can start dropping streams whose data are already in the clients' buffers. Bandwidth is saved as streams are terminated.

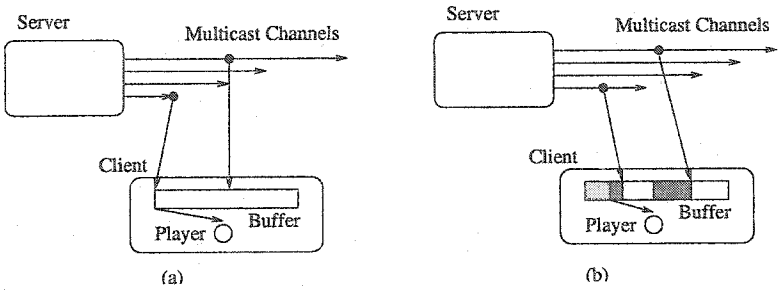

Figure 1.1: A system where clients can receive data from two streams (multicast channels) simultaneously. Each stream represents the same media object sent at a different time. Part (a) shows the initial client state where the client is granted a new stream from the server and is also instructed to buffer data from an existing stream. Part (b) shows the system at some later time when the client is simultaneously receiving, buffering, and viewing the data.

Figure 1.1 shows how stream merging works for the receive-two model, where clients can receive data from two streams (multicast channels) simultaneously. Each channel streams the same media object at a different time. The initial state of the system is shown in Figure 1.1(a) where the client is granted a new stream from the server and is also instructed to buffer future data from the current position of a stream that has been started earlier. The choice of which streams to listen to is different for each of the various approaches that have appeared in the literature [24, 22, 7, 6, 17, 14, 15]. Figure 1.1(b) shows the system after some time, where the client continues to play from the new stream, and has a later portion of the media buffered. The client can stop listening to the new stream when it has enough data buffered up from earlier streams. If no other clients need the stream, then the server can terminate it. This model is called stream merging because when a stream is terminated, the clients "merge" with clients listening to an earlier stream and the two streams become one.

We study the use of stream merging for two different models: the media-on-demand model, and the live broadcast with time-shifting model.

The Media-on-Demand Model: Media-on-Demand is the demand by clients to playback, view, listen, or read various types of media such as video, audio, or large files with little or no start-up delay and with no interruptions. Enjoying increasing use, media-on-

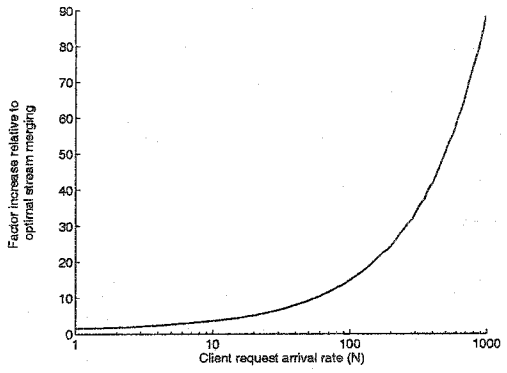

Figure 1.2: The theoretical performance gain achieved by using optimal stream merging as opposed to unicast for the media-on-demand model. The x-axis is the average number of client requests per media length on a log-scale, and the y-axis is the factor increase in required server bandwidth when using unicast as opposed to optimal stream merging.

demand is capturing the attention of many, from movie industries serving movie trailers to education providers who serve video-taped lectures across the Internet. The simplest (and most commonly used) solution dedicates a stream (channel) to each client request. This solution is simple, but very costly in terms of required server bandwidth. The potential benefit of using the receive-two stream merging model instead of the unicast solution is shown in Figure 1.2. The x-axis plots on a log-scale the client request arrival rate (N) measured as the average number of client requests in the length of time it takes to play the entire media object. The client arrival times follow a Poisson distribution. The y-axis plots the factor increase in total server bandwidth usage for unicast compared to optimal stream merging. For example, when $N = 100$, unicast is roughly 15 times worse than optimal stream merging. The figure implies that the benefits of stream merging are more pronounced as the client request intensity increases. We show later that the performance of the best online stream merging algorithms are in fact very close to optimal, so these large bandwidth savings can be achieved in practice.

The Live Broadcast with Time-Shifting Model: The broadcast systems of today (such as radio or TV), do not support clients who wish to join a broadcast started at an earlier time. For example, clients who wish to view a program at 10 pm must receive the

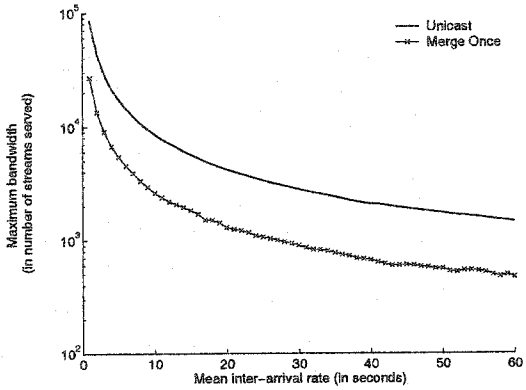

Figure 1.3: Potential performance gain achieved by using the simple merge-once stream merging solution as opposed to the unicast solution for the live broadcast with time-shifting model.

program at 10 pm, unless they record it. We propose a system in which clients may arrive at time t and ask to join the broadcast at time $t-w$ for some offset value $w \geq 0$. This feature is called *live broadcast with time-shifting*. Although it seems like the media-on-demand model and live broadcast with time-shifting model are similar enough that solutions to one would apply to the other, we will show in Chapter 3 that subtle differences make the time-shift problem more difficult. This is especially true when trying to design an optimal algorithm.

The simple unicast solution allocates a dedicated channel for every request. This solution is very simple because a client can just tune to its dedicated channel to receive the entire transmission. However, this technique is very costly in terms of required server bandwidth. By adapting the stream merging solution, we can achieve a huge improvement over the unicast solution. As an example, Figure 1.3 shows a comparison of the unicast solution and the merge-once algorithm. The merge-once algorithm has each client merge at most once, and is the simplest stream merging solution. The graph shows the results of running a 24 hour experiment where the client arrival times follow a Poisson distribution and the first requested segment falls at the start of an hour (simulating the start of a TV program). The x-axis plots the mean inter-arrival rate, and the y-axis plots the maximum number of concurrent channels supplied by the server over the sequence of client requests. Notice that the y-axis is on a log scale, so the simple merge-once stream merging solution significantly

outperforms the unicast solution. This demonstrates the potential benefits of applying the stream merging technique. We show later that we can also achieve a huge performance gain over the simple merge-once algorithm by adapting other stream merging algorithms.

We note that digital video recorder (DVR) systems like Replay TV [2] and TiVo [3] provide similar time-shifting capabilities. The major difference is that these systems require the client to plan ahead and specify which programs (or program types) should be recorded. Time-shifting capabilities are provided for these recorded programs by accessing the recorded data when necessary. Also, note that this approach does not alleviate the load on the server because each client must still download its own copy of a program. The focus of our work is on a technique that eliminates the burden of specifying in advance which programs should be recorded, while at the same time reducing the server load required to provide clients with access to live broadcast with time-shifting.

1.1.1 Related Work

The solution of dedicating a private channel to each client for the required media is problematic even with the ever growing availability of network bandwidth. A commonly proposed technique to reduce server bandwidth is to utilize multicast. The first and most natural way to exploit the advantage of multicast is to batch client requests and serve them with the same multicast stream. The trade-off involved with this solution is the start-up delay and the bandwidth saved.

The revolutionary pyramid broadcast scheme pioneered by Viswanathan and Imielinski [51] introduced additional trade-offs with two other resources: client receive bandwidth and client buffer size. The novelty of the pyramid broadcast paradigm is that clients are capable of receiving more bandwidth than they need for playback. This allows them to buffer later portions of the required transmission, and to play them on time. Using these additional resources, the pyramid broadcast scheme demonstrated a huge improvement over the simple batching solution.

Following the pyramid broadcast scheme, many solutions emerged to explore the trade-offs of the four resources: server bandwidth, client bandwidth, client buffer, and start-up

delay. Solutions like skyscraper broadcasting [31] achieved dramatic reductions in required server bandwidth by using the receive-two model. The skyscraper model assumed a static allocation of bandwidth per media object. Newer models like patching [10, 27, 11], tapping [13], piggybacking [4, 28, 29, 32], and stream merging [24, 22, 7, 6, 17, 14, 15] use dynamic allocation of bandwidth to media objects to allow the flexibility needed for serving multiple objects. We focus on stream merging in our work since it seems to incorporate all the advantages of the pyramid broadcasting paradigm and is very useful in designing and implementing efficient offline and online solutions.

The original stream merging algorithms of Eager *et al.* [24, 22] were event-driven, decisions for which streams to listen to were made at the time of an event. The specific events were the arrival of a client, the merge time of two streams, and the termination of a stream. The papers reported good results compared to the optimal algorithm using simulations with Poisson arrivals. Bar-Noy and Ladner were the first to provide a worst-case analysis of stream merging algorithms. They designed algorithms based on merge trees and a provided worst-case analysis for both their optimal offline algorithm [7] and their dynamic Fibonacci tree online algorithm [6]. They showed that their online algorithm has $O(\log n)$ competitive performance, and also achieved good results using simulations with Poisson arrivals. Coffman *et al.* presented the dyadic algorithm [17], and performed the first average-case stream merging analysis. They also reported good performance on Poisson arrivals. In recent years, Chan *et al.* performed an in-depth study of stream merging algorithms. First, they described a new algorithm called the the connector algorithm [14] that operates using rectilinear trees, and performed the first constant competitive analysis of a stream merging algorithm. In particular, they showed that their connector algorithm is 5-competitive (bandwidth usage no more than 5 times that of the optimal). They went on to derive a general framework for analyzing stream merging algorithms, and used it to show that the dyadic algorithm is 3-competitive [15]. All of the previous work on stream merging analysis attempted to minimize the total bandwidth usage of the server. Chan *et al.* showed that both their connector algorithm and the dyadic algorithm are 4-competitive with respect to maximum bandwidth usage.

1.1.2 Contributions

We present a comparative study of four recently proposed stream merging algorithms for the media-on-demand model in Chapter 2. In particular, we study the event-driven [24, 22], dynamic Fibonacci tree [6], dyadic [17], and connector [14] algorithms. It is important to evaluate these competing algorithms according to their strengths and weaknesses. Chan *et al.* performed very good comparisons using competitive analysis. They showed that both their connector algorithm and the dyadic algorithm are constant competitive. Because the performance of these algorithms are so close using a competitive analysis, we use a different approach in our study. In particular, we attempt to identify and classify the algorithms according to different inherent stream merging properties, and also perform a comprehensive study using simulations. The result of our work is a deeper understanding of the trade-off between stream merging complexity and performance for these stream merging solutions. The main contributions of this comparative study are:

1. *We extend the dyadic algorithm allowing us to achieve better performance.* We provide analysis and empirical evidence to show that our extension performs better than the original algorithm. We call our extension the α-dyadic algorithm. Chan *et al.* also studied an α-dyadic extension using a competitive analysis [15]. We study the α-dyadic algorithm using a different analysis, and an empirical study using simulations.

2. *We qualitatively compare the stream merging algorithms.* We illustrate the design principles and operation of each of the stream merging algorithms to gain an understanding of how they differ. We use this information to classify them according to stream merging complexity.

3. *We provide an in-depth empirical comparison of the stream merging algorithms.* We use simulations to compare the performance of the stream merging algorithms with respect to total server bandwidth usage (the metric they were designed to minimize). We also compare their performance using alternative measures of bandwidth usage. In particular, we look at their maximum and time-varying bandwidth usage, and consider the performance implications of servers with fixed maximum bandwidth (bounded

bandwidth). These alternative performance measures have significant implications for the design of real servers.

4. *We use a peer-to-peer file-sharing model [30] to represent a server that serves multiple media objects.* Although previous work has looked at a multiple object server [22], we are the first to use a more realistic model which is the result of an in-depth study on peer-to-peer multimedia workloads conducted at the University of Washington.

All of the prior work with algorithms based on the pyramid broadcasting paradigm are for the media-on-demand model. In other words, they all assume that the media "exists" and that clients demand it as a whole. In Chapter 3 we show that stream merging can be applied to the live broadcast with time-shifting model. We are the first to propose the incorporation of the pyramid broadcasting paradigm in such an environment.

1.2 Secure Media Delivery

The ability to securely deliver data to a group is important for emerging applications like subscription services to live and on-demand multimedia. Secure group communication systems typically rely on a *group key*, a secret shared by all members of the group. Access to group data is provided by encrypting all data with the group key. The *key management system* controls access to the group key, ensuring that only authenticated members receive the key. To facilitate this processes, the key management system also manages a set of *auxiliary keys* that are shared by some subset of group members, and *individual keys* that are assigned one per group member. When group membership changes, it becomes necessary to change the group key and some of the auxiliary keys to ensure that only the current group members have access to the group data. This operation is known as *re-keying*, and the main challenge is to perform re-keying in an efficient and secure fashion. A key management system is composed of the following three logical components (we focus on the second component):

1. *Mutual authentication and key exchange.* This component is responsible for authenticating new members and distributing their individual keys. The authenticated mem-

ber then joins/leaves the group by sending requests to the key management and distribution component.

2. *Key management and distribution.* This component is responsible for managing the set of keys in the system, and for generating re-key messages to change keys that are compromised due to group membership change.

3. *Re-key message transport delivery.* This component is responsible for the reliable delivery of re-key messages to group members.

The problem of managing and distributing the group and auxiliary keys is known as the *multicast key distribution* problem, and is the focus of our work. The key graph approach was proposed to efficiently solve this problem [52, 53]. Key graphs are logical data structures that represent the set of keys held by each group member. Key trees are an important class of key graphs that are sufficient for the scalable management of a group key. In a key tree, the group key is the root of the tree, auxiliary keys are internal nodes, and group members with their individual keys are leaves. We refer to both a group member and its individual key using the same name (it will be clear from context if we are talking about a member or its key). An example key tree for a group with nine members is shown in Figure 1.4. In this example, k_M is the group key, k_1, k_2, and k_3 are the auxiliary keys, and $u_1 \ldots u_9$ represent the nine members and their individual keys. A group member holds a key if the key is an ancestor of the member. For example, member u_1 holds key k_1, but member u_4 does not.

We should distinguish between a multicast tree that enables the efficient broadcast of data to all members in a multicast group and a key tree that is a logical structure for maintaining multicast group security. Each internal node in a key tree holds a group key for the subgroup consisting of all the members at the leaves of the subtree rooted at the node. A broadcast to this group is implemented by a multicast tree and is encrypted with the key held at the node.

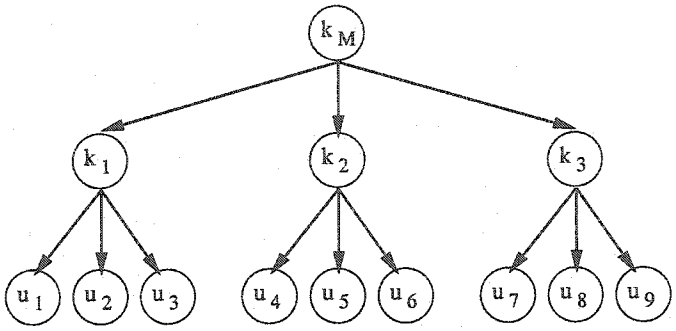

Figure 1.4: An example key tree with 9 members. The root of the tree contains the shared group key k_M.

1.2.1 Related Work

There has been a lot of work on designing group security solutions. These include secure multicast routing [48], reliable group re-key message delivery [54], and cryptographic schemes [50]. For an overview of the issues in multicast security, refer to the paper by Moyer *et al.* [38]. In our work, we focus on one important component of a group security system: designing scalable key distribution schemes for group communications. A common theme for solutions to this problem are hierarchies or multi-way trees.

In Iolus [36], scalable key distribution is handled by partitioning the group into multiple subgroups. The subgroups are arranged in a hierarchy, with each subgroup operating like an independent multicast group. A subgroup is managed by a Group Security Agent (GSA), which is responsible for communication with other GSAs and for handling local subgroup management (handling local joins/leaves, message delivery, etc.). A similar approach introduced a novel clustering algorithm with bounds on the subgroup sizes that allowed for better scalability [5].

Both Iolus and the approach using clustering require multiple trusted entities for key distribution. The key graph technique was independently developed by both Wallner *et al.* [52] and Wong *et al.* [53]. Like the Iolus and clustering approaches, key graphs also use a hierarchical structure. However, key graphs use a central key server, use a logical as opposed to a physical hierarchy, and only allow group members to be at the lowest level in the

hierarchy. Wallner *et al.* studied binary key trees. Wong *et al.* performed an extensive study, and concluded that the most efficient key graph is a *k-ary* tree (one where all internal nodes have k or fewer children). They also showed that the optimal degree is 4 when assuming equally likely re-key operations. Heuristics were used to guide the placement of new group members into the tree, but because there is no control over which members leave the group, the tree may become unbalanced. Moyer *et al.* [37] stated that maintaining balanced trees is desirable in practice because membership updates can be performed with logarithmic communication costs provided that the tree is balanced. They maintained binary key trees with a novel algorithm to keep the tree balanced, where a balanced tree was defined as one where the distance from the root to any two leaf nodes differs by at most 1. Rodeh *et al.* [45] presented an algorithm to maintain AVL trees, which are balanced binary trees where the height of the two subtrees (children) of a node differs by at most one.

There has also been work on optimizations to improve scalability. One example is periodic group re-keying [34, 47] as opposed to re-keying immediately after each group membership change. Periodic re-keying reduces the work of the key server, but introduces a period of vulnerability. For example, a deleted member can continue to receive group data until the periodic re-key time.

Fiat and Naor [25] studied the broadcast encryption problem where a center sends encrypted messages over a broadcast channel with the goal of establishing a secret with a chosen subset of possible receivers. Luby and Staddon [35] studied the trade-off between communication and storage using a combinatorial analysis, but had a restriction on the size of the set of possible receivers. Canetti *et al.* [12] studied the same trade-off without restrictions on group size. The work most closely related to ours is a recent paper by Snoeyink *et al.* [49] that derived a lower bound on the worst-case communication costs for any key distribution scheme that uses a central key server and simple private key encryption (such as DES). In particular, they showed that an optimal key distribution tree for n members is a special case of a 2-3 tree (one where all internal nodes have either 2 or 3 children), and has a worst-case communication cost of $\lceil 3 \log_3 n \rceil$ for a re-key operation. Poovendran and Baras [43] used an information-theoretic approach to show that the worst-case communication cost for any key tree distribution scheme is related to the entropy of the

member deletion process. If the members have identical probability of being deleted, then the communication cost is $\Omega(\log n)$. However, if the members have unequal probabilities of being deleted, then the average communication cost can be sub-logarithmic. For our work, we assume that we have no knowledge of member deletion probabilities. Therefore, the lower bound results of Snoeyink *et al.* and Poovendran and Baras are essentially the same.

1.2.2 Contributions

Our study of key distribution algorithms appears in Chapter 4. Previous results showed that if we have no knowledge of member deletion probabilities, then for any key distribution scheme, the lower bound on worst-case communication cost for re-keying a group of n members is $\Omega(\log n)$. For an algorithm that uses key trees to achieve the lower bound, balanced key trees must be used, but no description was given on how to maintain balance over sequences of group membership changes. Moyer *et al.* [37] maintained balance for binary trees, but can incur high cost to maintain their balanced tree structures. We improve on the previous work by addressing the above weaknesses. In particular, we present three new online algorithms for the dynamic maintenance of key trees over sequences of re-key operations (one is based on the AVL tree algorithm of Rodeh *et al.* [45]). We explain that the communication cost for online algorithms can be broken down into two parts: costs due to tree structure, and costs due to the overhead of restructuring required to maintain those tree structures. Our algorithms use local restructuring rules and achieve a good trade-off between tree structure and restructuring costs. We show that our algorithms have decent worst-case tree structure bounds, and show through simulations that this leads to good performance when the key tree would have become highly unbalanced without restructuring.

Chapter 2

COMPARISON OF STREAM MERGING ALGORITHMS

Stream merging is a technique for efficiently delivering popular media-on-demand using multicast and client buffers. Recently, several algorithms for stream merging have been proposed, and we perform a comprehensive comparison of them. We present the differences in philosophy and mechanics among the various algorithms, and illustrate the trade-offs between their stream merging complexity and performance. We measure performance in total, maximum, and time-varying server bandwidth usage under different assumptions for the client request patterns. We also consider the effects on clients when the server has limited bandwidth. The result of this study is a deeper understanding of the complexity and performance trade-offs for the various algorithms.

The remainder of this chapter is organized as follows: in Section 2.1, we define stream merging for the media-on-demand model. We describe the various stream merging algorithms that we use in our study in Section 2.2. We provide our qualitative analysis of the algorithms and discuss their relative complexities in Section 2.3. Our empirical comparison of the various algorithms using simulations is given in Section 2.4. Finally, Section 2.5 presents our conclusions and lists directions for future research.

2.1 Model and Preliminaries

There are four parameters that have been studied in the literature for media-on-demand delivery schemes. They are: server bandwidth, client bandwidth, client buffer, and start-up delay. We focus on the commonly used receive-two model (client bandwidth is twice the playback rate). We assume that the client has a buffer large enough to store half of the media-object being downloaded. The receive-two model is sufficient since it has captured most of the benefit provided by stream merging. It was shown that any further gain by receiving more than two streams simultaneously is small [7]. We think that the receive-two

model is practical for use over local area networks and enterprise networks where sufficient bandwidth is available. We also feel that the client buffer size is not too important since it is not a concern for many Internet applications, and set-top boxes with large buffers have already appeared on the marketplace. We note that client buffer size is a concern for low resource devices (such as mobile devices), however in these situations the client bandwidth would likely be the more constrained resource. Furthermore, previous work has shown that the worst-case analysis for an unbounded client buffer is easily extended for an analysis using a bounded client buffer [7]. We leave the empirical study using a bounded buffer for future research.

Time is slotted and the transmission is segmented such that it takes one slot of time to transmit one segment of the broadcast. Clients that arrive in the same time slot are batched together and served by a stream that starts at the end of the time slot. We assume that these stream start times are $0 \leq t_1 < t_2 < \cdots$. For simplicity, we refer to the t_i as client arrival times since the streams are started to service client requests. These new streams may not need to run until completion since later portions may have already been buffered. The size of the time slot is referred to as the *start-up delay*, and is the delay from the time a client requests a given media object until the time the client starts receiving the media object. This is without loss of generality since we could make the time slot as short as required (shrinking it to 0 for immediate service).

An example operation of stream merging is shown in the time-segment diagrams of Figure 2.1. For simplicity, we refer to both the client and the new stream started for it by the same name. The media object is broken into equal sized segments. The x-axis represents time, the diagonal lines represent the media object sent over different multicast channels, and the numbers above the diagonal lines represent the media segment sent at that time. Each client (except the first) listens to two streams simultaneously. When a client has enough data, it can merge with an earlier stream (represented by the dashed vertical lines). If no clients are listening to a stream, it can be terminated. For example, in Figure 2.1(a) we see that client B listens to streams B and A for three units, receiving parts $0, 1, 2$ from stream B and parts $3, 4, 5$ from stream A. At that point, the client no longer needs stream B so it merges to A (shown by the dashed vertical line). Also, stream

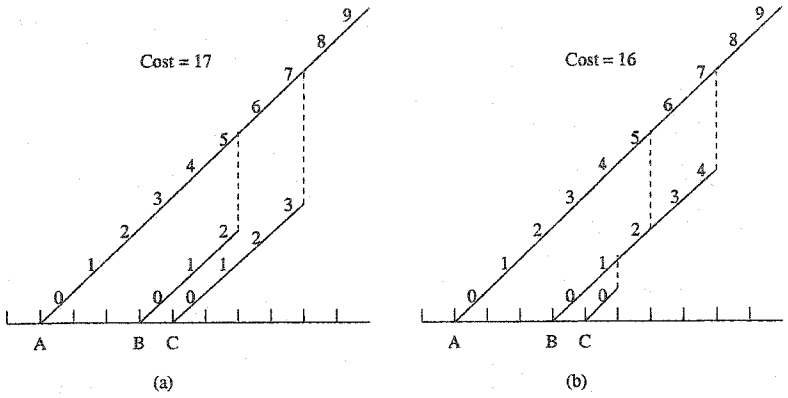

Figure 2.1: Two choices for stream C are illustrated. In (a) the client served by C merges directly to A, while in (b) the client first merges to B then to A.

B is terminated since no other clients are listening to it.

The information from the server informing a client of which streams to listen to and for how long is called the *client receive program*. Figure 2.1 shows two possible client receive programs for client C (clients A and B have the same receive program in both parts of the figure). In (a) the client receives data from streams C and A for four units, then from stream A for two units. In (b) the client receives data from streams C and B for one unit, from streams B and A for three units, and from stream A for the last two units. Note that although client B no longer needs stream B after it merges with A (shown by the dashed vertical line), stream B needs to be extended for 2 more slots so that client C has enough data to merge with A. The stream length represents how long a stream is active before it is not needed by any clients. The total bandwidth used is conveniently represented as the sum of all stream lengths. By adding up the stream lengths we see that if there are no future arrivals, then option (b) uses less total bandwidth.

An alternative representation of the operation of stream merging is the merge tree. Figure 2.2 shows a time-segment diagram and its corresponding merge tree. Nodes in the tree are labeled by their arrival time, and the tree structure indicates the merge patterns. A client represented by a given node labeled x will listen to the stream started at time x and the stream started at time $p(x)$, its parent node in the merge tree. This continues until

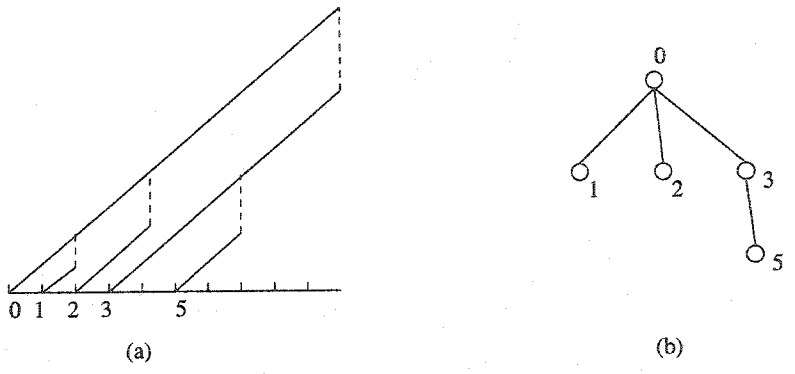

Figure 2.2: A time-segment diagram is illustrated in (a) and its corresponding merge tree is illustrated in (b).

the client no longer needs the stream started at time x. The client will then listen to the streams started at $p(x)$ and $g(x)$, where $g(x)$ is the grandparent of the client in the merge tree. This process continues until the client listens only to the root stream. This implies that all arrivals in the merge tree will eventually merge to the root. Note that if the media object has length L units, then an arrival that comes more than L units after the root of a given merge tree cannot be incorporated into the tree. This means that multiple merge trees may be necessary to represent the entire operation of a stream merging algorithm. The length of a stream started at a given non-root node x is given by

$$\ell(x) = 2z(x) - x - p(x) \tag{2.1}$$

where $p(x)$ is the parent of node x and $z(x)$ is the latest arrival in the subtree rooted at x [7]. The length of the root is simply the media length L. In this way, when looking at the final merge trees representing the operation of any stream merging algorithm, the total bandwidth used can be found by adding the lengths of all streams. The advantage of using the merge tree representation is that it is easier to reason with than the time-segment diagram. In addition, the merge tree is a simple data structure that can be used to keep track of the state of the streams in the system.

Bar-Noy and Ladner say that a merge tree has the *preorder traversal property* if a preorder traversal of the merge tree yields the client arrival times in order [7]. They showed

that every optimal merge tree satisfies the preorder traversal property. Some of the online stream merging algorithms that we examine use this fact and restrict their merge trees to have the preorder traversal property.

2.2 Stream Merging Algorithms

In this section we provide a description of each of the stream merging algorithms that we are studying. Each algorithm attempts to minimize total server bandwidth, but differs in how the server decides which streams a client should listen to and for how long. We also introduce two other media-on-demand solutions that are used as a starting point for comparison: unicast, and merge-once. We include these solutions in our study because unicast is the most commonly used media-on-demand solution, and merge-once is the simplest stream merging solution.

2.2.1 Unicast

The simplest and worst performing media-on-demand delivery solution is unicast, where each client request is met with its own dedicated stream. The main advantage of this approach is its simplicity, the client does not have the complexity of listening to two streams and does not even require a buffer. This is not a stream merging solution, but we include it in our study because it is the most commonly used media-on-demand solution. This makes it a good starting point for comparisons in our complexity and performance studies.

2.2.2 Merge-Once Algorithm

The merge-once model is based on patching [10, 27, 11], and is only slightly more complicated than unicast. This technique is included in our study because it is the simplest form of stream merging. Each client request is assigned a primary stream and receives the prefix of the primary stream from a secondary stream. The primary stream multicasts the full media object, and the secondary stream is a dedicated unicast stream. Once the (one) merge occurs, the client listens to just the primary stream. In both the optimal offline and the online case the issue is to find the full primary streams. The optimal solution can be found

by using dynamic programming. We use an online algorithm that starts a new full stream once the accumulated cost of the secondary streams exceeds the media object length. That is, if L is the length of a full stream and t_i is a full stream, then the next full stream is t_j such that $\sum_{i<k<j}(t_k - t_i) \leq L$ but $\sum_{i<k\leq j}(t_k - t_i) > L$. Because all merge trees for the merge-once algorithm are star trees with depth at most 2, by Equation (2.1) the length of a non-root stream is simply $x - p(x)$.

2.2.3 Event-Driven Algorithm

The original stream merging algorithm used an event-driven model where the decisions for which two streams a client listens to are made by the server at the time of an event [24, 22]. There are three types of events: the arrival of a client, the merge time of two streams, and the termination of a stream. The server treats all clients that are listening to the same two streams as a group. When an event occurs, the server must decide which two streams each group of clients should listen to. The particular technique used in our study is the earliest reachable merge target (ERMT) heuristic since it was the best reported heuristic in a previous study [22]. The ERMT heuristic attempts to merge a group of clients with the closest stream that it can successfully merge with. The event-driven algorithms were the first to introduce the stream merging model and showed good performance in simulations.

The operation of the algorithm is best illustrated through an example. Consider the operation of the algorithm for the arrival times $t = [0, 3, 4]$ and a media object length of $L = 7$ units. Several snapshots of the time-segment diagram as the algorithm progresses are provided in Figure 2.3. The solid lines represent the past while dashed lines represent the possible future (in the absence of any new events). The first event occurs with the client arrival at time 0. Since there are no other streams in the system, the client only listens to one stream. The next event occurs at time 3 with the arrival of a new client request. A new stream is created to serve this request. At this point, there is only one additional stream (the stream started at time 0), which can be caught before it terminates. This means that the client arriving at time 3 listens to streams 0 and 3 simultaneously as shown in Figure 2.3(a). The dashed lines show that stream 3 will merge to stream 0 at

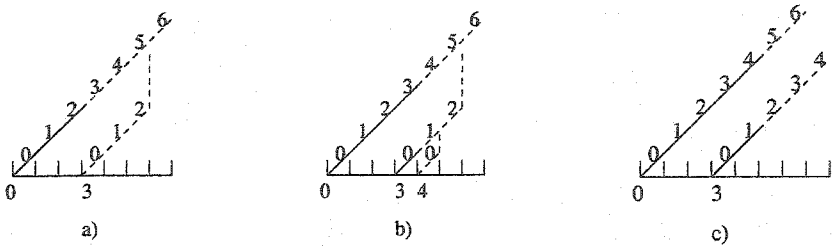

Figure 2.3: Operation of the ERMT algorithm that illustrates the dynamic nature of the server decisions. Part (a) shows the state of the system at time 3, part (b) shows the state of the system at time 4, and part (c) shows the state of the system at time 5 after streams 3 and 4 have merged. Dashed lines show the merge patterns from the point of view of each client in the absence of any other events.

time 6 if there are no earlier events. The next event is the arrival of a new client at time 4. This client listens to its own stream, and listens to the earliest reachable target which is the stream started at time 3 as shown in Figure 2.3(b). The next event is the merge at time 5 between streams 3 and 4 leaving the stream that started at time 3. At this point, all the clients listening to stream 3 need another stream to listen to. This includes both the clients that listened to stream 3 from when it began, and the clients that just finished listening to stream 4. There is only one other stream in the system (stream 0). However, the clients cannot catch (merge with) stream 0 before it terminates as shown in Figure 2.3(c). This is because the first segment received from stream 0 after the merge point is segment 5. This causes stream 3 to be lengthened to a point where it can no longer merge with stream 0. As a result, the clients of stream 3 only listen to stream 3 from this point on.

2.2.4 Dynamic Fibonacci Tree Algorithm

The dynamic Fibonacci tree algorithm uses the infinite Fibonacci tree structure to determine client receive programs [6]. The main contribution of this work was showing that the algorithm is $O(\log n)$ competitive. We provide a short description, but omit most of the details of the algorithm because it is lengthy and can be found in [6].

As mentioned above, the algorithm determines client receive programs by using the infinite Fibonacci tree structure. Recall that merge trees are a data structure that contains

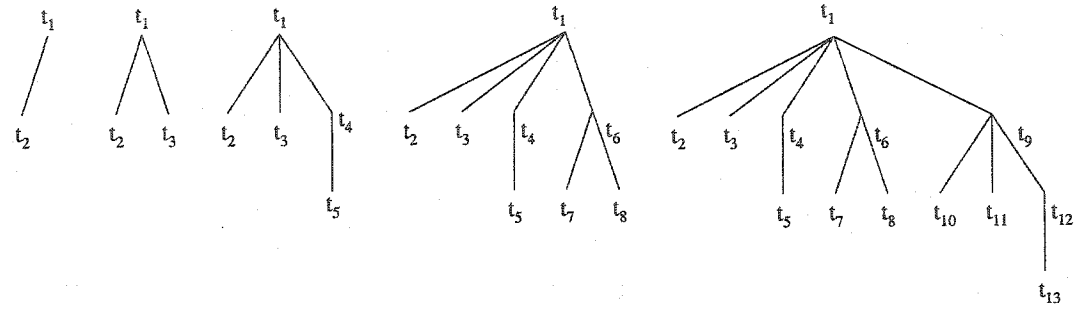

Figure 2.4: Illustration of the static Fibonacci trees $\mathcal{F}_3$, $\mathcal{F}_4$, $\mathcal{F}_5$, $\mathcal{F}_6$, and $\mathcal{F}_7$. The static trees $\mathcal{F}_1$ and $\mathcal{F}_2$ each consist of a single node (and are not shown). The static trees for $\mathcal{F}_k$ are constructed by making the root of $\mathcal{F}_{k-2}$ the last child of the root of $\mathcal{F}_{k-1}$.

the structure of all client receive programs. The algorithm starts a new merge tree when the arrival time of a client is more than $L/2$ units after the root of the current merge tree, where L is the number of units in the entire media object. Define F_k to be the kth Fibonacci number, and $\mathcal{F}_k$ to be a static Fibonacci tree with F_k nodes. The main properties of a static Fibonacci tree are: the number of nodes in the kth tree is F_k, and the kth tree is created from the $k-1$st and $k-2$nd trees. In particular, $\mathcal{F}_1$ and $\mathcal{F}_2$ each consist of a single node, and $\mathcal{F}_k$ is constructed by making the root of $\mathcal{F}_{k-2}$ the last child of the root of $\mathcal{F}_{k-1}$. Figure 2.4 shows the static Fibonacci trees $\mathcal{F}_3$, $\mathcal{F}_4$, $\mathcal{F}_5$, $\mathcal{F}_6$, and $\mathcal{F}_7$ for 2, 3, 5, 8, and 13 nodes respectively. Each tree is labeled in preorder fashion. Note that the recursive structure allows the maintenance of an infinite Fibonacci tree. The algorithm holds on to a static tree $\mathcal{F}_k$ such that there are currently $n \leq F_k$ clients in the merge tree. The algorithm can always extend the size of the Fibonacci tree as needed, and can maintain the preorder labeling of the tree.

The dynamic Fibonacci tree algorithm uses the infinite Fibonacci tree as its merge tree with clients being inserted into the tree in a preorder fashion. Let $t_1 = y_0, y_1, \ldots, y_k = t_{n-1}$ be the path from the root to the last arrival so far. It could be the case that the next arrival t_n cannot be inserted in the desired position (because the parent has already terminated), or it might be cheaper to choose another parent y_i. Heuristics are used to decide which y_i should be the parent for the new arrival t_n, and the static Fibonacci tree is modified [6]. Tree

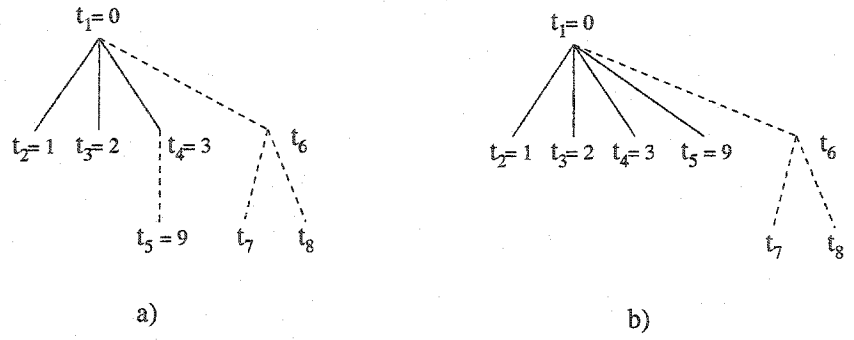

Figure 2.5: An example of a dynamic Fibonacci tree structure. Part (a) shows where arrival $t_5 = 9$ should go, but because it cannot merge with $t_4 = 3$, the tree is restructured as shown in part (b).

modifications are done to ensure that the resulting tree maintains the preorder property and has at least as good performance as the current tree for all future arrivals. As an example, consider the set of arrival times $t = [0, 1, 2, 3, 9]$, and a stream length of $L = 20$ units. Figure 2.5(a) shows part of the infinite Fibonacci tree after processing the first 4 arrivals. The dashed lines indicate the future, and the current size of the infinite Fibonacci tree is 8 nodes. Part (a) shows where the arrival $t_5 = 9$ should go, but because the stream started at $t_4 = 3$ has already terminated, the tree is restructured as shown in part (b). Note that the preorder traversal property is preserved, and the construction of the remainder of the infinite Fibonacci tree can proceed as described earlier.

2.2.5 Dyadic Algorithm

The dyadic algorithm [17] uses recursive dyadic interval partitioning to determine client receive programs. The main contribution of this work was the first average-case analysis of a stream merging algorithm. Like the dynamic Fibonacci tree algorithm, the dyadic algorithm starts a new merge tree when the arrival time of a client is more than $L/2$ units after the root of the current merge tree. The merge tree is constructed in a recursive fashion. First, the algorithm determines the children of the root. Next, each child recursively becomes the root for the subtree rooted at it. The process of deciding which arrivals become the children of a given root is done using dyadic interval partitioning as shown in Figure 2.6. For a root

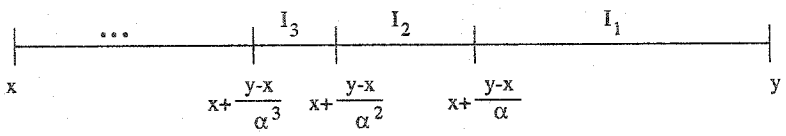

Figure 2.6: Illustration of dyadic interval partitioning over the interval $(x, y]$, where x is the root stream for all arrivals in the interval and y is the time after which no arrivals are allowed to merge to x. The first arrival in each dyadic interval I_i merges directly to x and becomes a root for all remaining arrivals in I_i.

arriving at time x, the initial interval $(x,y]$ is created where y is the time after which no arrivals are allowed to merge to the root. This interval is partitioned using the parameter α as shown in Figure 2.6, where α can take on any value greater than one. In this way, each partition is recursively split as many times as necessary depending on the actual arrivals.

The original paper [17] used $\alpha = 2$ and we call this the 2-dyadic algorithm. We consider a variant that divides the intervals by the golden ratio ϕ, where $\phi = (1 + \sqrt{5})/2$. Chan *et al.* [15] also studied the α-dyadic algorithm using a competitive analysis, and found that the algorithm is 3-competitive when $\alpha = 1.5$. Our analysis of a functional identity related to the "continuous" version of the problem yields ϕ as an optimal solution. Consider the case of almost instantaneous arrivals in the interval $(0, x]$ with a root stream at 0, where the arrivals before x/α are handled recursively and the arrivals after x/α are handled recursively with a second root stream at approximately x/α, which itself eventually merges to the root. The total merge cost $M_\alpha(x)$ (cost excluding the root stream [7]), approximately satisfies the equation

$$M_\alpha(x) = M_\alpha\left(\frac{x}{\alpha}\right) + M_\alpha\left(\left(1 - \frac{1}{\alpha}\right)x\right) + \left(2 - \frac{1}{\alpha}\right)x. \tag{2.2}$$

The first term is the merge cost of streams initiated before x/α (not counting the root stream at 0), the second term is the merge cost of streams initiated after x/α, and the third term is the cost of the stream started at x/α (see Equation (2.1)). A solution to Equation (2.2) is

$$M_\alpha(x) = \frac{\frac{1}{\alpha} - 2}{\frac{1}{\alpha}\ln\left(\frac{1}{\alpha}\right) + \left(1 - \frac{1}{\alpha}\right)\ln\left(1 - \frac{1}{\alpha}\right)} x \ln(x)$$

which is minimized when $\alpha = \phi$. Furthermore, we provide empirical motivation for using

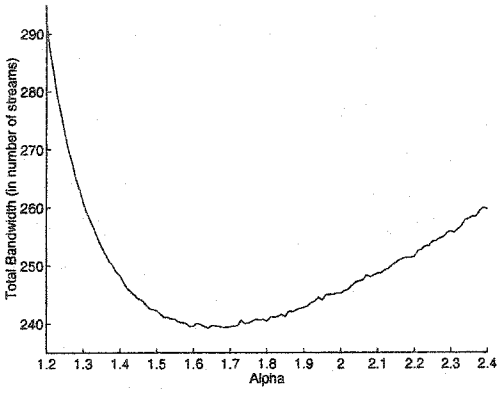

Figure 2.7: Variation of α for the α-dyadic algorithm using Poisson arrivals with an average of 1,000 requests in the length of time it takes to serve the entire media object.

the ϕ-dyadic algorithm in Figure 2.7. We ran a simulation where the client arrivals times follow a Poisson distribution, and the intensity of arrivals is $N = 1,000$, where N is the average number client requests in the amount of time it takes to serve the entire media object. We used different values of α from 1.2 to 2.4 in increments of 0.01. We see that the best α is 1.65, which is close to ϕ. We also ran this experiment with other client arrival distributions and saw the same curve shape, with the minimum always near or equal to ϕ.

The dyadic algorithm as described above is an offline algorithm since it requires complete knowledge of the entire arrival sequence to do the dyadic interval partitioning. However, it is simple to construct an online algorithm using a stack [17]. We present a simple example to illustrate how the algorithm processes arrivals in an online fashion. We consider the operation of the 2-dyadic algorithm for a media object length of $L = 20$ units and the arrival times $t = [0, 4, 5, 6, 9]$. The first arrival is at time 0, so the stream started at time 0 becomes a root. The interval from 0 to $L/2 = 10$ is then partitioned into dyadic intervals as illustrated in Figure 2.8(a) where $x = 0$, $y = 10$, and $\alpha = 2$. The next arrival is at time 4, and since it is the first arrival in the interval I_2 it becomes a child of the stream started at time 0. The stream started at time 4 then becomes a root for any arrivals between times 4 and the end of the interval I_2. The next arrival is at time 5, and merges to the stream started at time 4 since it is contained in the interval shown in Figure 2.8(b). The next

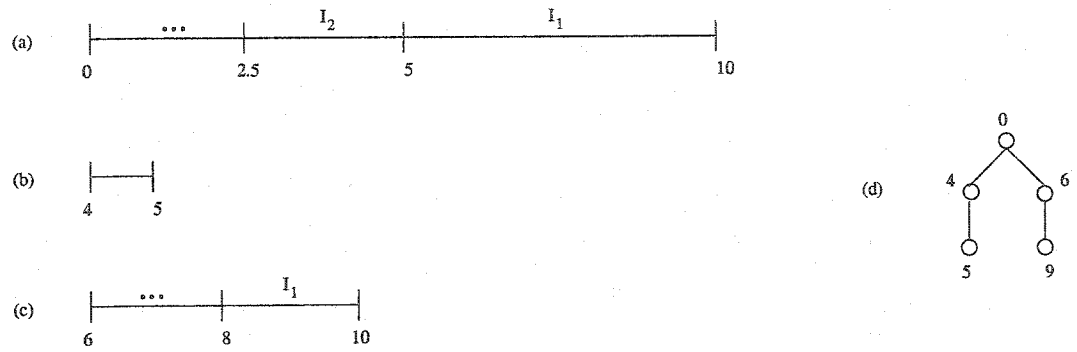

Figure 2.8: Parts (a), (b), and (c) show the dyadic intervals created for the arrivals $t = [0, 4, 5, 6, 9]$. Part (d) shows the final merge tree constructed by the dyadic algorithm for the given arrivals.

arrival at time 6 does not fit into the interval shown in Figure 2.8(b). However, it is the first arrival in the interval I_1 shown in Figure 2.8(a), so it becomes a child of the stream started at time 0, and is used as the root for any arrivals from time 6 to the end of the interval I_1. The final arrival is contained in the interval shown in Figure 2.8(c) so it merges to the stream started at time 6. The merge tree for this example is shown in Figure 2.8(d).

2.2.6 Connector (Rectilinear Tree) Algorithm

The connector algorithm [14] introduced a data structure called a rectilinear tree to represent merge patterns. A rectilinear tree is defined as a binary tree on a triangular grid with the following properties:

1. The root of the rectilinear tree is the top right corner of the triangle, and the leaves are on the hypotenuse of the triangle.

2. The leaves of the rectilinear tree represent client arrivals, while the internal nodes represent times when a client changes which stream(s) it is listening to.

3. Every edge in the rectilinear tree is represented by a horizontal or vertical grid line. In other words, a node and its parent are either in the same row or same column.

The main contribution of this work was providing the first constant competitive analysis of a stream merging algorithm. In particular, they showed that their connector algorithm is 5-competitive. Both the dynamic Fibonacci tree and dyadic algorithms start a new merge tree when the arrival time of a client is more than $L/2$ units after the root of the current merge tree. The connector algorithm uses the same criteria to decide when to start a new rectilinear tree. We provide a short description of how the connector algorithm constructs rectilinear trees. For more details, please refer to the paper by Chan *et al.* [14].

A rectilinear tree lies in the triangle bounded by the three points $a = (0, 0)$, $b = (0, K-1)$, and $c = (K-1, K-1)$, where $K = L/2 + 1$. The root is at point $(0, K-1)$. The online connector algorithm $\mathcal{A}$ constructs its rectilinear tree by making reference to another rectilinear tree $\mathcal{R}$. The reference tree $\mathcal{R}$ is constructed using a simple recursive structure, and is used because it provides a good estimate of an optimal rectilinear tree. Define $f(n) = \lfloor (2/3)(n-1) \rfloor + 1$. The left subtree of the root contains the top $f(K)$ leaves along the hypotenuse, and the right subtree contains the remainder of the leaves along the hypotenuse. The roots of these subtrees are located at grid points $(0, f(K)-1)$ and $(f(K), K-1)$. Each subtree is divided using the same recursive structure until it contains only one leaf. An example of a reference rectilinear tree for $L = 22$ is shown in Figure 2.9(a).

The online connector algorithm $\mathcal{A}$ assumes that the first client arrives at time $t = 0$. This client listens to its own stream (does not merge to any other stream). A new stream is started for each subsequent request i. The reference rectilinear tree is used to determine when the client i should start buffering data from an earlier stream. Let the point (i, i) represent the new client i. Define a to be the least common ancestor of i and the previous arrival in the reference tree $\mathcal{R}$. Using a, new grid paths are added to the online rectilinear tree as follows.

1. If a is on column i, add a new vertical grid path running up from (i, i) that stops at the first line it intersects in the current online rectilinear tree. This means that i immediately listens to two streams.

2. If a is not on column i, then it must be to the right of column i because of the monotone property [14]. Let a lie on column c_a. The new grid path is a horizontal

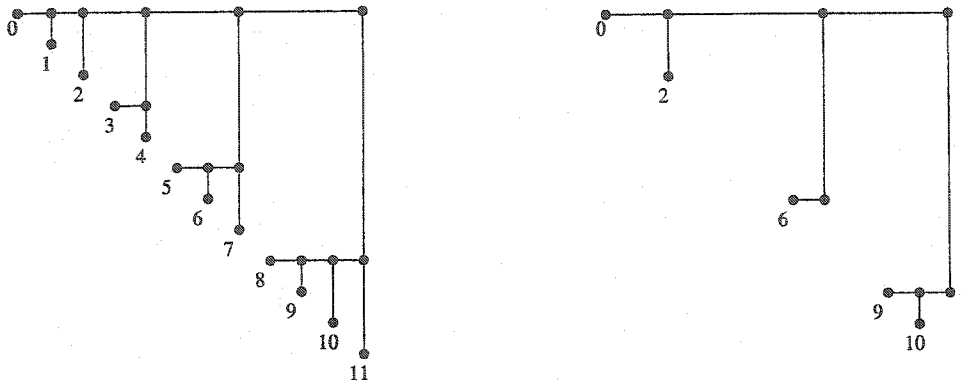

Figure 2.9: Illustration of the rectilinear tree data structure. Part (a) shows a reference rectilinear tree for $L = 22$. Part (b) shows the corresponding rectilinear tree constructed by the online algorithm for the arrivals $t = [0, 2, 6, 9, 10]$.

line from (i, i) to (i, c_a) and a vertical line from (i, c_a) to the first line it intersects in the current online rectilinear tree. This means that i first starts listening to only its stream, then listens to two streams at a later time.

As an example, consider the client arrival times $t = [0, 2, 6, 9, 10]$, and a stream length of $L = 22$. Figure 2.9(a) shows the reference rectilinear tree $\mathcal{R}$. Part (b) shows the rectilinear tree constructed by the online algorithm. Note that the trees are very similar, but that there is a slight deviation from the reference tree $\mathcal{R}$ for the arrivals 6, 9, and 10. Although the arrival at time 6 could have started listening to the root stream immediately, it delayed merging for one unit (represented by the horizontal line). The connector algorithm does this so that it can closely follow the reference tree. For example, the arrival at time 6 delays merging so that it can be used as a merge target if an arrival comes at time 7.

2.3 Stream Merging Complexity

In a way there exists a hidden hierarchy of models that trades off complexity for performance. It is important to note that we are not talking about implementation complexity, but rather

about the inherent differences between the algorithms which we describe below. Stream merging systems can be viewed as being composed of three components: the server, the multicast channels, and the clients. The server gives clients receive programs and schedules streams and prefixes of streams on the channels. The clients follow the receive program given by the server, possibly buffering data for future use. Unfortunately, we cannot quantify stream merging complexity. We can only order the systems intuitively from the simplest to the most complex. We use the following three criteria in our classification:

1. *Client buffer required.* This is the only feature on the client-side that differs among the various media-on-demand solutions that we consider in this paper. Note that a buffer is always required if a client is allowed to listen to more than one stream simultaneously.

2. *Dynamic stream lengths.* When the server creates a new multicast stream, it can either decide to fix the amount of time that this stream will be broadcast, or to allow the length to change depending on future arrivals (lengthen the stream if a client that arrives in the future can use it). We refer to the ability to change stream lengths as dynamic stream lengths.

3. *Dynamic receive programs.* When the server receives a new client request, it can either decide to tell the client its entire receive program, or to tell the client only parts of the receive program and possibly change it in the future. We refer to the ability to change the receive program as dynamic receive programs.

These three criteria define inherent properties of a media-on-demand solution. Intuitively, a solution that requires a client buffer is more complex than one that does not. All stream merging solutions require a client buffer since a client must have the ability to listen to two or more streams simultaneously. A solution that allows dynamic stream lengths is more complicated than one that does not because the server does not know in advance when a stream will terminate. Finally, a solution that allows dynamic receive programs is more complicated than one that does not because the server must constantly contact

clients to tell them which streams to listen to. Using our three criteria, we now classify the various media-on-demand solutions used in our study by increasing order of stream merging complexity.

No client buffer, static stream lengths, static receive programs. The simplest media-on-demand solution is the unicast solution. This solution does not require a client buffer since the client only listens to one stream. The server never changes stream lengths or receive programs since it sends each client a full dedicated stream.

Client buffer, static stream lengths, static receive programs. The merge-once and connector algorithms fall into this class. Using either of these solutions, the client requires a buffer since it listens to two streams simultaneously. However, the server still does not use dynamic stream lengths or dynamic receive programs. For the merge-once algorithm, the primary streams are full streams and the secondary streams are unicast to one client, so the server has complete knowledge of stream lengths and the receive programs for each client at the time of its arrival. The connector algorithm also never changes its stream lengths or client receive programs. When a client arrives, the server generates a new stream and determines the stream length using grid paths in the rectilinear tree structure. These grid paths never change, and may be longer than necessary because the algorithm tries to follow the reference rectilinear tree $\mathcal{R}$ as closely as possible.

Although both the merge-once and connector algorithms fall into this class, we can say that the merge-once algorithm is simpler because it only allows one merge per client.

Client buffer, dynamic stream lengths, static receive programs. The dyadic and dynamic Fibonacci tree algorithms both fall into this class. They are both based on building merge tree structures to determine receive programs. As a result, both require a client buffer and dynamic stream lengths. However, receive programs never change because it is completely determined by a node's spot in the merge tree.

Client buffer, dynamic stream lengths, dynamic receive programs. Finally, the most complex technique is the event-driven solution. It requires client buffers and allows dynamic stream lengths and dynamic receive programs. Allowing dynamic receive programs

was illustrated in Figure 2.3, each client only knows the two streams that it is currently listening to.

We show through simulations in Section 2.4 that these differences in stream merging properties are reflected in actual performance. It does make sense that the more flexible an algorithm is (the more it can change or delay its decisions), the better the performance should be. But how much better? We investigate this trade-off between complexity and performance in the next section.

2.4 Simulation Results

We present a comparison of the stream merging algorithms using simulations. We describe our simulation setup in Section 2.4.1. We study how the algorithms compare with respect to total server bandwidth (the metric they were designed to minimize) in Section 2.4.2. In Section 2.4.3, we study how the various algorithms compare with respect to alternative server bandwidth measures (time-varying, minimum, maximum). Finally, we present our study of a server with bounded bandwidth (fixed maximum bandwidth) in Section 2.4.4.

For the remainder of this section, we present empirical results comparing only the ERMT, dynamic Fibonacci tree, ϕ-dyadic, and connector algorithms. The reason for choosing $\alpha = \phi$ was shown in Figure 2.7, and the reason for eliminating the unicast and merge-once algorithms is shown in Figure 2.10. We see that there is a huge difference in performance between the unicast, merge-once, and connector algorithms. For this graph, performance is measured in total server bandwidth. We use Poisson client arrivals where the x-axis plots the client request arrival rate N ranging from 100 to 1,000, where N is the average number of client requests in the length of time it takes to play the entire media object. The y-axis plots the factor increase in total server bandwidth compared to the optimal algorithm on a log-scale. The optimal solution can be computed using dynamic programming [7]. It is obvious that the simplicity of both the unicast and merge-once algorithms hurts their performance. Similar behavior was observed for the other models that we describe in Section 2.4.1. Therefore, we eliminate the unicast and merge-once algorithms from the remainder of our study.

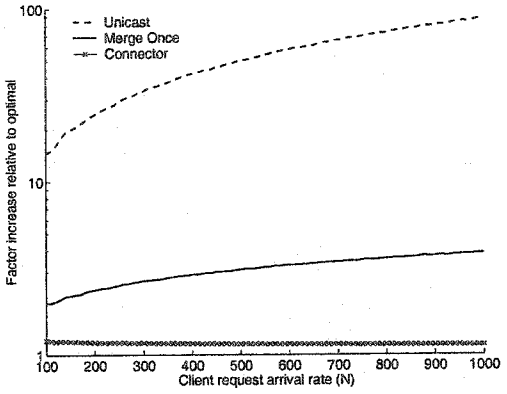

Figure 2.10: Illustration of the huge performance benefits of using the connector stream merging algorithm over the simple unicast and merge-once algorithms.

2.4.1 Simulation Setup

We study both a server that serves a single object and server that serves multiple objects. For the single object server, we experiment with both constant-rate and Poisson client arrival distributions. The motivation behind a constant-rate arrival distribution is that it arises in a heavily loaded system with batching where the start-up delay is equal to the constant inter-arrival time. As long as there is at least one arrival every unit (where a unit is as long as the start-up delay) we get the constant-rate arrival distribution. While Poisson arrivals is not the most realistic assumption, this is the distribution used in previous work with the stream merging algorithms. The client arrival intensities are chosen to represent requests for popular objects since Figure 1.2 showed that the benefits of stream merging are more pronounced as the client arrival intensity increases.

Using a single object server helps in understanding the performance of the various stream merging algorithms. However, a server normally serves more than one object. This motivates the multiple object server model. There are a few additional parameters involved when considering a server with multiple objects. These are: distribution of object sizes, object popularity, and temporal locality in requests for individual objects. We assume for simplicity that all objects have the same size, and that client arrival intensities follow a Poisson distribution. To generate the distribution of object popularity and temporal local-

ity of requests, we use the peer-to-peer file sharing workload model of Gummadi *et al.* [30]. This model was created to reflect observed properties of multimedia file sharing during a 200-day study at the University of Washington. The model was also used to verify various hypotheses about the driving forces behind such a workload, so that it could be used to predict future trends. For our experiments, we choose parameter values that reflect the trends observed in the study at the University of Washington. In particular, we assume that the server holds $40,000$ objects, and that the clients observe a fetch-at-most-once behavior. The system initially has $1,000$ clients in its population, and the simulation runs for $100,000$ requests. It might seem that fixing the number of requests limits the usefulness of our results. For example, what if we had $1,000,000$ requests instead? However, we note that the total number of requests were chosen together with the other parameters to generate observed trends in object popularity and temporal locality. Once these are fixed, the performance of the stream merging algorithms depends on the arrival rate rather than the total number of requests.

Note that each media object is independent, that is, streams for one media object cannot be shared (merged) with streams for another media object. Therefore, when measuring total server bandwidth, the quantity measured for the multiple object server is the same as adding up the total server bandwidth used for each individual media object. However, this is not true for the maximum and time-varying bandwidth metrics, or the bounded bandwidth study. For these studies, there is an interaction between the time that requests are made and the media object that is requested. This is the reason that the multiple object server study is interesting.

2.4.2 Total Server Bandwidth

In this section we present our comparison of the various algorithms with respect to their total server bandwidth usage. We represent the client arrival intensity N as the average number of client requests in the length of time it takes to play the entire media object. The performance numbers are presented as the factor increase in total server bandwidth relative to the optimal algorithm.

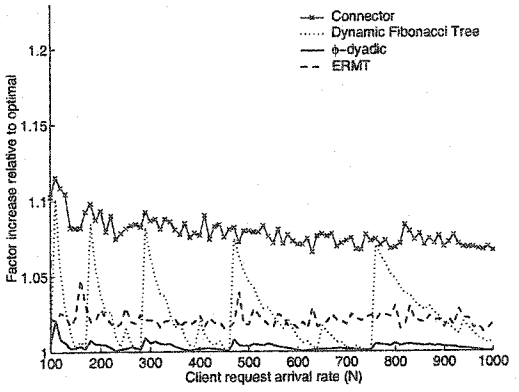

Figure 2.11: Total server bandwidth comparison for the connector, dynamic Fibonacci tree, ϕ-dyadic, and ERMT algorithms. Single object server, constant-rate arrivals.

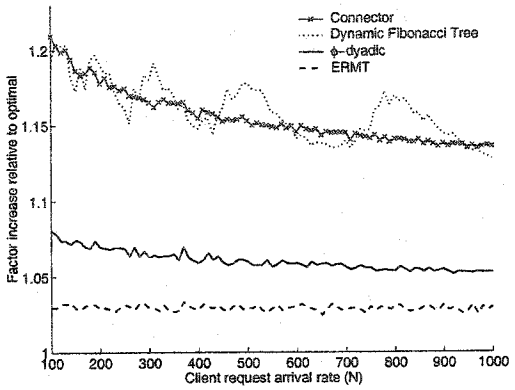

Figure 2.12: Total server bandwidth comparison for the connector, dynamic Fibonacci tree, ϕ-dyadic, and ERMT algorithms. Single object server, Poisson arrivals.

Figure 2.11 shows our results for a single object server where we have constant-rate client arrivals. Figure 2.12 shows the results for a single object server where the client arrival times are Poisson distributed. For both experiments, we run an experiment that is 25 times the length of the media object. The x-axis plots the client arrival intensity from $N = 100$ to $N = 1,000$ and the y-axis plots the factor increase in total server bandwidth relative to optimal.

Looking at the results for constant-rate arrivals in Figure 2.11, we see that all of our algorithms perform very well, with bandwidth usage less than 1.1 times that of the optimal over most of the experiment. One thing that stands out is the periodic performance of the dynamic Fibonacci tree algorithm. When N is a Fibonacci number, then with constant-rate arrivals the Fibonacci tree constructed for each root stream in the dynamic Fibonacci tree algorithm is optimal. So we see that the performance of the dynamic Fibonacci tree algorithm greatly improves as N approaches a Fibonacci number.

All of the algorithms also perform very well for Poisson client arrivals (Figure 2.12). Note that the results for constant-rate and Poisson arrivals are shown with the same scale on the y-axis. The ERMT algorithm has roughly the same performance. However, the connector, dynamic Fibonacci tree, and ϕ-dyadic algorithms all perform noticeably worse

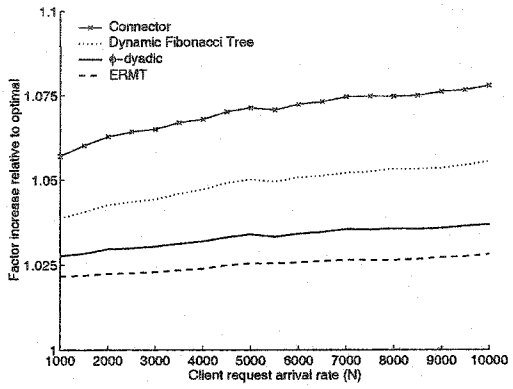

Figure 2.13: Total server bandwidth comparison for the connector, dynamic Fibonacci tree, ϕ-dyadic, and ERMT algorithms. Multiple object server using the peer-to-peer file sharing workload model.

for Poisson arrivals. These algorithms all impose some structure on the merge pattern (they use static receive programs). This causes them to perform better when the arrival pattern is predictable (constant-rate arrivals). On the other hand, since the ERMT algorithm uses dynamic receive programs, it can easily adjust to the client arrival pattern, and thus performs equally well for both constant-rate and Poisson arrivals.

Figure 2.13 shows the results for our multiple object server. The x-axis plots the client arrival intensity from $N = 1,000$ to $N = 10,000$ and the y-axis plots the factor increase in total server bandwidth relative to optimal. Because our simulation runs for 100,000 client requests, the experiment ranges from 100 to 10 times the media length depending on the value of N. Looking at the results, we see that all of our algorithms perform very well, with bandwidth usage less than 1.075 times that of the optimal over most of the experiment.

2.4.3 Alternative Server Bandwidth Measures

In this section we present our comparison of the algorithms with respect to minimum, maximum, and time-varying server bandwidth usage. These alternative server metrics are important for real servers and have not been studied in as much depth as the total server bandwidth metric. The graphs in this section show histograms of bandwidth usage where the x-axis plots the number of concurrent streams that are being served and the y-axis plots the

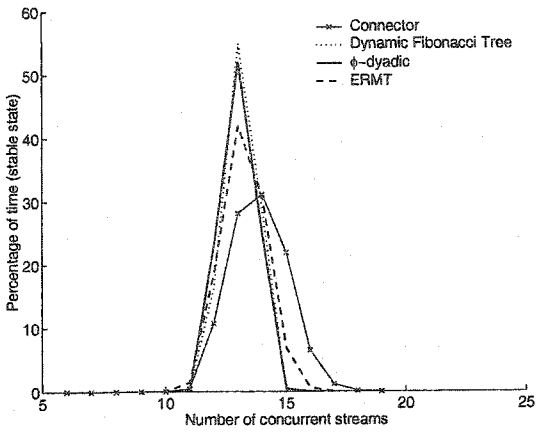

Figure 2.14: A histogram of server bandwidth usage for the connector, dynamic Fibonacci tree, ϕ-dyadic, and ERMT algorithms. Single object server, constant-rate arrivals.

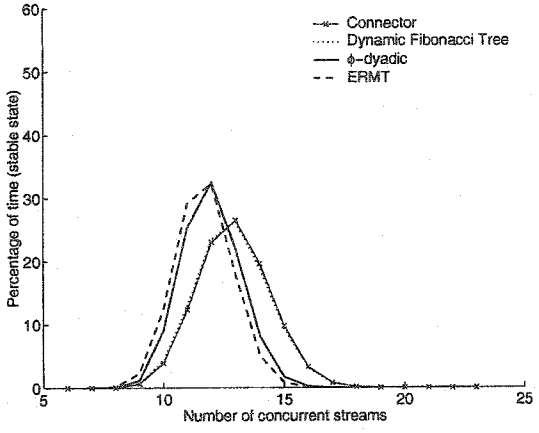

Figure 2.15: A histogram of server bandwidth usage for the connector, dynamic Fibonacci tree, ϕ-dyadic, and ERMT algorithms. Single object server, Poisson arrivals.

percentage of time that the server is serving a given number of concurrent streams. These graphs also allow us to see the minimum and maximum number of concurrent streams served over our experiments. The stream merging algorithms were not designed to minimize any of these metrics, so we did not use the optimal algorithm for this study. Note that for these histograms, points farther to the left are better since they represent lower bandwidth usage, and a narrow distribution of points is also good since that represents low variance.

We look at the server bandwidth usage over time for a single object server when $N = 1,000$ with constant-rate client arrivals in Figure 2.14, and with Poisson client arrivals in Figure 2.15. The experiments were run for a length of time that is 102 times the media length. We cut out the data for the first and last media length to prune out the simulation start-up and termination effects, allowing us to use data that is representative of the system in steady-state. The performance of our algorithms are very similar. There is a lot of overlap among the curves, and the minimum and maximum points are all within a few streams of each other. Note that the results for both the constant-rate and Poisson arrival experiments show the same range on the y-axis. We notice that the algorithms have a narrower distribution for constant-rate arrivals. This makes sense because the arrival times

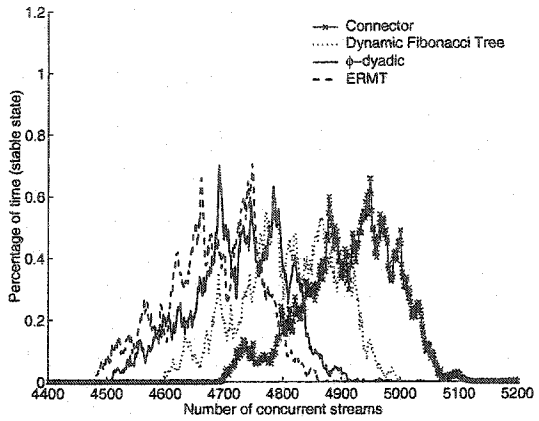

Figure 2.16: A histogram of server bandwidth usage for the connector, dynamic Fibonacci tree, ϕ-dyadic, and ERMT algorithms. Multiple object server using the peer-to-peer file sharing workload model.

are structured, meaning that the server constantly serves a roughly stable number of clients.

We look at the server bandwidth usage over time for our multiple object server when $N = 10,000$ in Figure 2.16. The experiment was run for a length of time that is 12 times the media length. We cut out the data for the first and last media length to prune out the simulation start-up and termination effects, allowing us to use data that is representative of the system in steady-state. The performance of our algorithms are similar. However, it is clear that the ERMT algorithm performs the best, followed by the ϕ-dyadic, dynamic Fibonacci, and connector algorithms.

2.4.4 Server with Bounded Bandwidth

In this section we present our comparison of the algorithms when we have a server with bounded bandwidth (fixed maximum bandwidth). We study two client behaviors: client balking frequency where the clients leave if they do not get immediate service, and average client waiting time where clients wait until the server is able to satisfy their requests. The available server bandwidth is expressed as the number of streams per client that arrives in the length of time it takes to stream the media object, and is calculated by taking the maximum server bandwidth (in number of streams) and dividing it by N.

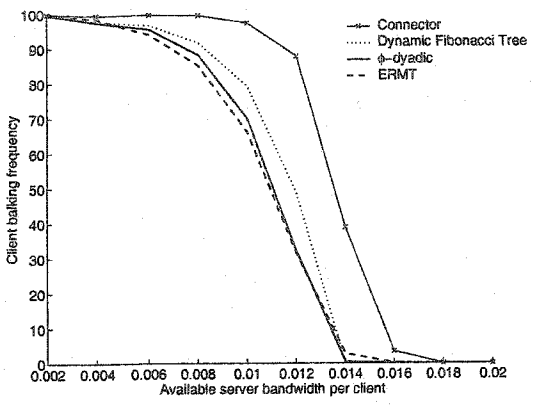

Figure 2.17: Client balking frequency for the connector, dynamic Fibonacci tree, ϕ-dyadic, and ERMT algorithms. Single object server, constant-rate arrivals.

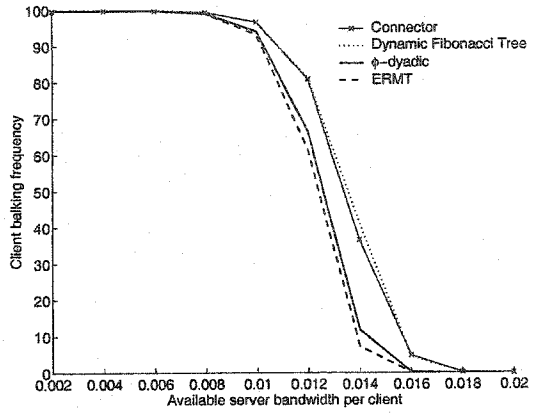

Figure 2.18: Client balking frequency for the connector, dynamic Fibonacci tree, ϕ-dyadic, and ERMT algorithms. Single object server, Poisson arrivals.

Client Balking Frequency: For all of these results, the x-axis plots the available server bandwidth per client, and the y-axis plots the percentage of clients that are denied service. Curves that are pushed over to the left represent better performance since this means that fewer streams are needed to avoid client balking.

The client balking frequency for a single object server when $N = 1,000$ is shown in Figure 2.17 for constant-rate arrivals and Figure 2.18 for Poisson arrivals. The graphs show the average of 100 trials. The algorithms all have very similar performance. We see that all of the algorithms have a critical point where the client balking frequency rapidly drops from close to 100% to near 0% as we increase the available number of streams. Both graphs show that the ERMT algorithm performs the best, followed closely by the ϕ-dyadic algorithm. The dynamic Fibonacci tree and connector algorithms have slightly worse performance.

The client balking frequency for our multiple object server when $N = 10,000$ is shown in Figure 2.19. The experiment was run for a length of time that is 10 times the media length. Similar to what we observed for the single object server, we see that all of the algorithms have a critical point where the client balking frequency rapidly drops to near 0% as we increase the available number of streams. We see that the ERMT algorithm performs the best, followed by the ϕ-dyadic, dynamic Fibonacci tree, and connector algorithms.

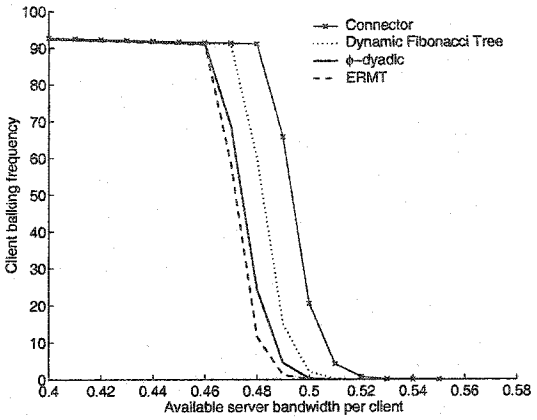

Figure 2.19: Client balking frequency for the connector, dynamic Fibonacci tree, ϕ-dyadic, and ERMT algorithms. Multiple object server using the peer-to-peer file sharing workload model.

Mean Client Waiting Times: For all of these results, the x-axis plots the available server bandwidth per client, and the y-axis plots the mean waiting time for clients (expressed as a percentage of the full media object). For these experiments the server attempts to minimize client waiting times by selecting the queued client that has been waiting the longest. Curves that are pushed over to the left represent better performance since this means that fewer streams are needed to avoid delaying service to clients.

The mean client waiting times for a single object server when $N = 1,000$ is shown in Figure 2.20 for constant-rate arrivals and Figure 2.21 for Poisson arrivals. The graphs show the average of 100 trials. The algorithms all have very similar performance. The ERMT algorithm performs the best by a slight margin, followed closely by the ϕ-dyadic, dynamic Fibonacci tree, and connector algorithms.

The mean client waiting times for our multiple object server when $N = 10,000$ is shown in Figure 2.22. The experiment was run for a length of time that is 10 times the media length. The ERMT, ϕ-dyadic, and dynamic Fibonacci tree algorithms have almost identical performance, with the connector algorithm having the worst performance among the four.

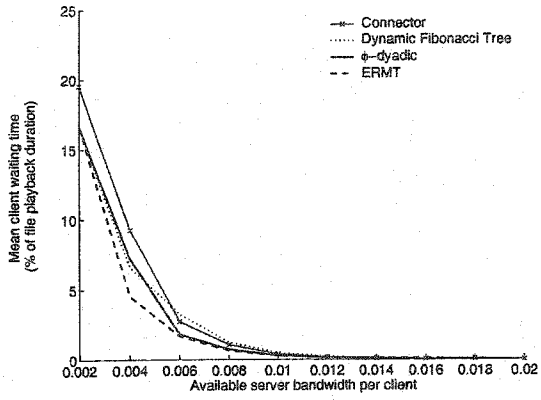

Figure 2.20: Mean client waiting times for the connector, dynamic Fibonacci tree, ϕ-dyadic, and ERMT algorithms. Single object server, constant-rate arrivals.

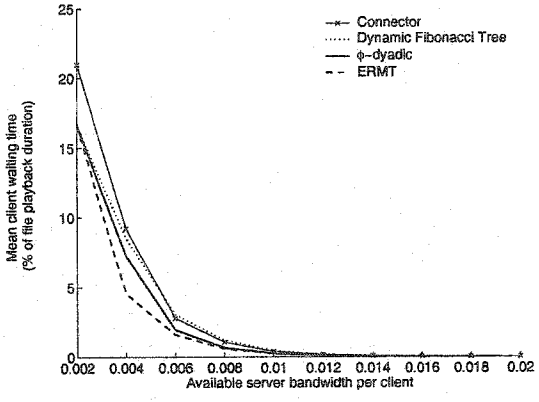

Figure 2.21: Mean client waiting times for the connector, dynamic Fibonacci tree, ϕ-dyadic, and ERMT algorithms. Single object server, Poisson arrivals.

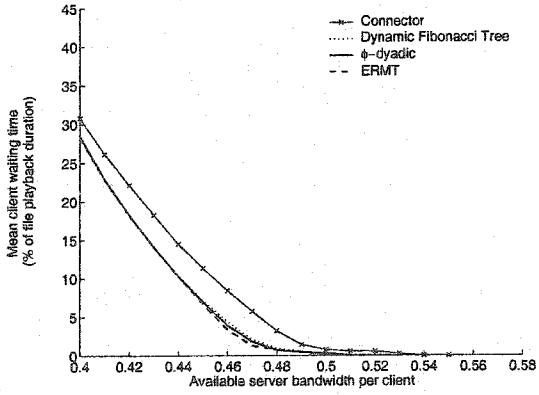

Figure 2.22: Mean client waiting times for the connector, dynamic Fibonacci tree, ϕ-dyadic, and ERMT algorithms. Multiple object server using the peer-to-peer file sharing workload model.

Table 2.1: Summary of the stream merging complexity and performance trade-off for the various media-on-demand solutions that we studied.

	Algorithm	*Buffer Required*	*Dynamic Stream Lengths*	*Dynamic Receive Programs*
	Unicast			
⇓	Merge Once	√		
Increasing	Connector	√		
performance	Fibonacci Tree	√	√	
⇓	Dyadic	√	√	
	ERMT	√	√	√

⇒ Increasing stream merging complexity ⇒

2.4.5 Discussion of the Simulation Results

Our experiments showed that the unicast and merge-once algorithms perform poorly compared to the other stream merging algorithms that we studied. Our experiments also showed that the ERMT, dynamic Fibonacci tree, ϕ-dyadic, and connector algorithms have similar performance and all perform very well for all of the various performance metrics that we studied. It is interesting to note that even though these algorithms were designed to minimize total server bandwidth usage, they all perform well with respect to the alternative server bandwidth measures and for the bounded bandwidth server.

One goal of our study was to understand the trade-off between stream merging complexity and performance for each of the algorithms. We summarize our findings in Table 2.1. We note that the algorithms increase in both stream merging complexity and performance as we move down the table. It seems that the ϕ-dyadic algorithm achieves the best trade-off among the studied algorithms because it does not require dynamic client receive programs, yet has performance that is very close to the ERMT algorithm. The connector and dynamic Fibonacci algorithms take a larger hit in performance. The ERMT algorithm usually performs the best, and is less influenced by the type of client arrival distribution because of its flexibility in making merge decisions on the fly.

2.5 *Conclusions and Future Work*

Media-on-demand is enjoying increasing popularity. However, the existing solution of serving a dedicated stream for each request is not scalable. Multicast is a commonly proposed technique to reduce the required server bandwidth used by many media-on-demand solutions including the recent stream merging model, and is a feasible technology for enterprise and local area networks. We demonstrated that by using stream merging huge bandwidth savings is possible over the traditional unicast solution.

We studied various stream merging solutions and identified the differences in their complexity and performance. Unicast is the simplest and worst performing solution where each client request is met with a dedicated stream. The merge-once solution exhibits a huge performance gain over unicast while introducing the modest complexity of handling a buffer and downloading two streams simultaneously. There is another huge gain in performance between the merge-once solution and any of the recently proposed stream merging algorithms. These algorithms have similar performance using all of the performance metrics that we studied, and all perform very well. In general, ERMT performs the best followed closely by ϕ-dyadic, with the dynamic Fibonacci tree and connector algorithms having the worst performance among the four. Of these stream merging algorithms, the connector algorithm is the only one with static stream lengths and static client receive programs. Both the ϕ-dyadic and dynamic Fibonacci tree algorithms have similar complexity (both use dynamic stream lengths). The ERMT algorithm is the most complicated stream merging solution because it postpones decisions for which streams a client should listen to until the last possible moment. This added flexibility explains why it usually performs the best.

We list a few of the interesting possibilities for further research.

1. Refine the stream merging hierarchy by considering other models with different combinations of client buffer, client receive bandwidth, and start-up delay. A particularly interesting parameter to study is the client receive bandwidth. The k-receive model has been examined in earlier stream merging papers. A value of $k > 2$ might be used where high client bandwidth is available. A more interesting case called bandwidth skimming [23] has $1 < k < 2$, and corresponds to having low client bandwidth.

2. Consider dynamic access patterns such as a prime-time TV program, increasing intensity of arrivals, and decreasing intensity of arrivals. For example, the popularity of an interesting video clip increases dramatically as more people hear about it.

3. Study other measures of performance like network bandwidth instead of server bandwidth. This may give new perspectives toward better stream merging decisions, however, it is a major challenge to come up with suitable models to measure such a cost metric.

Chapter 3

STREAM MERGING FOR LIVE BROADCAST WITH TIME-SHIFTING

We consider live broadcast (such as radio or TV) to which users can join with time-shifting. That is, they can join the broadcast at time t and receive the broadcast of time $t - w$ for some offset parameter $w \geq 0$. The simplest implementation that supports such a feature allocates a dedicated channel to each request. Using such a technique, the server bandwidth quickly becomes a bottleneck. We adapt the stream merging technique to the time-shifting model, which allows us to greatly reduce the required server bandwidth.

The remainder of this chapter is organized as follows: in Section 3.1, we define stream merging in the live broadcast with time-shifting model. We discuss the difficulties in trying to develop a polynomial time optimal offline algorithm for this model in Section 3.2. We present our modification of the dyadic stream merging algorithm to allow it to operate for the time-shift model in Section 3.3. We study the performance of our modified dyadic algorithm using simulations in Section 3.4. Finally, we conclude and provide directions for future work in Section 3.5.

3.1 Model and Preliminaries

We assume the existence of a live broadcast that lasts forever. Time is slotted and the transmission is segmented such that it takes one slot of time to transmit one segment of the broadcast. This is without loss of generality since we could make the time slot as short as required. The root transmission R broadcasts segment t of the live broadcast at time t for $t \geq 0$. A client requesting the broadcast arrives at time t, and starts receiving the broadcast at an arbitrary offset $t - w$, $w \geq 0$. The following formally defines a client request in the live broadcast with time-shifting model.

Definition 1 *Each client request X is represented by the pair $(t(X), f(X))$, where*

1. *$t(X)$ is the arrival time.*

2. *$f(X) \leq t(X)$ is the first requested segment.*

In addition, the artificial arrival time $a(X) = t(X) - f(X)$ is the time that client X would have arrived had it requested segment 0.

We assume that all clients that arrive during a time slot wait for the beginning of the next time slot. In other words, clients may arrive at any time, but all clients arriving within the same time slot are batched together and treated as one client. A new stream is initiated for each client X (or batched group of clients), unless the client can immediately merge with an existing stream. This happens if there is a stream Y such that $a(X) = a(Y)$. For simplicity, we denote a stream by the name of the client that initiated it. Clients can simultaneously receive and buffer data from two streams, and can play the data back from their buffers when needed. Each client must receive all segments of the broadcast that it wishes to play before or at the playback time. This allows each client to play the broadcast without any interruptions. An example operation of stream merging in the live broadcast with time-shifting model is shown in Figure 3.1 for the seven requests

$A = (6, 2)$ $B = (8, 3)$ $C = (12, 3)$ $D = (13, 1)$ $E = (15, 2)$ $F = (19, 5)$ $G = (20, 9)$.

We will refer to this figure throughout this section as we describe the live broadcast with time-shifting model. In the time-segment diagram (right part of the figure), the x-axis corresponds to time and the y-axis corresponds to segments. Each stream is labeled with the request it was initiated for, and merge times are shown by the dashed vertical lines. For example, stream B merges to stream A at time 9. The live root stream (starting at time 0) runs forever, and all other streams are terminated when they are no longer needed by any client. For example, stream D terminates at time 19 when request E merges with stream C. On the other hand, stream C does not terminate at time 28 when request E leaves it to merge with the root stream R. This is because requests F and G need later segments from stream C.

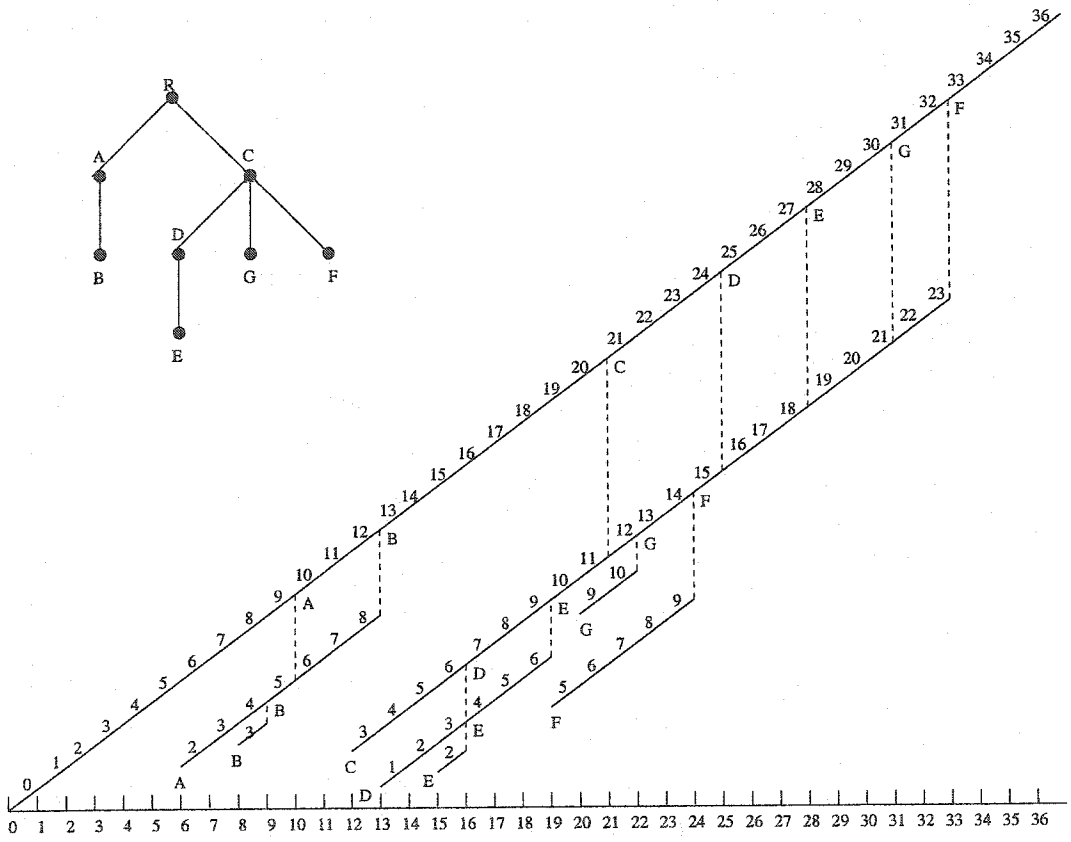

Figure 3.1: Merge tree and corresponding time-segment diagram for the seven requests A, B, C, D, E, F, G in the live broadcast with time-shifting model.

We condense the time-segment diagram into a *merge tree* whose nodes are labeled by the streams.

Definition 2 *A merge tree represents the streams used to satisfy a finite set of client requests, and has the following properties.*

- *The root of the tree is the live stream R.*
- *If stream X merges with stream Y, then Y is the parent of X.*
- *The children of a stream Y are ordered by their termination time, which is by definition their merge time with stream Y.*

Figure 3.1 shows a merge tree and its corresponding time-segment diagram. The root stream is R, streams A and C merge directly to R with stream A merging before stream C. Stream B is the only stream that merges to stream A. Streams D, G, and F merge to stream C (in the order given). Stream E merges to stream D. Note that the merge tree does not show stream lengths, however, it implicitly contains stream length information as will be shown in Lemma 4.

Receiving Procedures: Clients receive and buffer data from at most two streams. The particular stream(s) that a client listens to at any given time is determined by its location in the merge tree. Let $\mathcal{P} = \langle X = X_k, X_{k-1}, \ldots, X_0 = R \rangle$ be the path from client X to the root in the merge tree. This ordered sequence of streams $\mathcal{P}$ is called the *receive program* for X, and can be found from the merge tree representation by following the path from X to the root. For example, the receive program for client E in Figure 3.1 is $\langle E, D, C, R \rangle$. Next, we define how client X follows its receive program to ensure it can play the broadcast without any interruptions.

Definition 3 *Stream Merging Rules: Let $\mathcal{P} = \langle X = X_k, X_{k-1}, \ldots, X_0 = R \rangle$ be the receive program for client X. At stage i, $0 \leq i \leq k-1$, from time $[2t(X_k) - t(X_{k-i})] + [f(X_{k-i}) - f(X_k)]$ to time $[2t(X_k) - t(X_{k-i-1})] + [f(X_{k-i-1}) - f(X_k)]$, the client receives parts*

1. $f(X_k) + 2[t(X_k) - f(X_k)] - 2[t(X_{k-i}) - f(X_{k-i})], \ldots, f(X_k) + 2[t(X_k) - f(X_k)] - [t(X_{k-i}) - f(X_{k-i})] - [t(X_{k-i-1}) - f(X_{k-i-1})] - 1$ *from stream* X_{k-i}, *and parts*

2. $f(X_k)+2[t(X_k)-f(X_k)]-[t(X_{k-i})-f(X_{k-i})]-[t(X_{k-i-1})-f(X_{k-i-1})], \ldots, f(X_k)+2[t(X_k) - f(X_k)] - 2[t(X_{k-i-1}) - f(X_{k-i-1})] - 1$ *from stream* X_{k-i-1}.

At stage k, the client only receives data from the live root stream.

Consider the example shown in Figure 3.1. We show how the client $E = (15, 2)$ uses the stream merging rules to follow its receive program $\mathcal{P} = \langle E, D, C, R \rangle$. In this case we have $k = 3$ with $E = X_3 = (15, 2)$, $D = X_2 = (13, 1)$, $C = X_1 = (12, 3)$, $R = X_0 = (0, 0)$. From time 15 to time 16, the client receives part 2 from stream $E = X_3$ and part 3 from stream $D = X_2$. From time 16 to time 19, the client receives parts $4, \ldots, 6$ from stream $D = X_2$ and parts $7, \ldots, 9$ from stream $C = X_1$. From time 19 to time 28, the client receives parts $10, \ldots, 18$ from stream $C = X_1$ and parts $19 \ldots 27$ from stream $R = X_0$. After that, the client only receives data from the live root stream R. The above discussion can be verified by looking at the time-segment diagram shown in figure 3.1.

Stream Lengths: The stream merging rules explain how a client follows its receive program. We still need to show how to compute the minimum stream lengths needed to guarantee that all clients receive the necessary data.

Let $P(X)$ be the parent of stream X. Let $Z(X)$ be the last request to merge with stream X. If X appears in the receive program of a request, then $P(X)$ is the next stream in this receive program. In the merge tree representation, request $Z(X)$ is the right most leaf in the subtree rooted at X. The root stream represents the live broadcast, and runs forever. The next lemma shows how to compute the length of all non-root streams.

Lemma 4 *The length of a non-root stream X is*

$$\ell(X) = [2t(Z(X)) - t(X) - t(P(X))] + [f(P(X)) - f(Z(X))] .$$

Proof: Since $Z(X)$ is the last request to merge with X, we know that the length of X is dictated by the needs of $Z(X)$. Let $\mathcal{P} = \langle Z(X) = X_k, X_{k-1}, \ldots, X_0 = R \rangle$ be the receive

program for $Z(X)$ that contains X. That is, $X = X_i$ and $P(X) = X_{i-1}$ for some $i > 0$. By Definition 3, at stage $k - i$, the client $Z(X)$ receives data from stream $X = X_i$ until time $[2t(X_k) - t(X_{k-i-1})] + [f(X_{k-i-1}) - f(X_k)] = [2t(Z(X)) - t(P(X))] + [f(P(X)) - f(Z(X))]$. Since $Z(X)$ is the last client requiring stream X, stream X can be terminated at this time. Since X begins at time $t(X)$ and ends at time $[2t(Z(X)) - t(P(X))] + [f(P(X)) - f(Z(X))]$, its length is $[2t(Z(X)) - t(X) - t(P(X))] + [f(P(X)) - f(Z(X))]$. □

As an example, consider stream C in Figure 3.1. It follows that $F = Z(C)$ and $R = P(C)$ (the root stream). Therefore, the above lemma implies that $\ell(C) = [2t(F) - t(C) - t(R)] + [f(R) - f(F)]$. Since $t(F) = 19$, $t(C) = 12$, $t(R) = 0$, $f(R) = 0$, and $f(F) = 5$, we get that $\ell(C) = (2 \times 19 - 12 - 0) + (0 - 5) = 21$. Indeed, in the diagram of Figure 3.1, the length of Stream C is 21 since it starts at time 12 and terminates at time 33.

The Optimization Goal: The natural optimization objective is to minimize the total bandwidth needed to support all the requests. In other words, the goal is to minimize the sum of the lengths of all streams except the root stream (which runs forever). Formally, Let $\mathcal{X} = \{X_1, \ldots, X_n\}$ be a set of n requests. Let $\mathcal{Y}(\mathcal{X}) = \{Y_1, \ldots, Y_m\}$ be a set of m streams that satisfies all the requests in $\mathcal{X}$. Then the cost of the solution $\mathcal{Y}(\mathcal{X})$ is

$$C(\mathcal{Y}) = \sum_{i=1}^{m} \ell(Y_i) \ .$$

Note that minimizing the total bandwidth is essentially the same as minimizing the average bandwidth needed to satisfy the requests. This is because the average bandwidth required can be defined as the total bandwidth required divided by the length of the interval starting at time 0 and ending at the time of the last merge.

3.2 Optimal Offline Algorithm

It is not clear how to find the optimal merge tree with a polynomial time algorithm for the time-shift model. A dynamic programming solution was derived for the special media-on-demand model when all clients start with segment 0 [7]. A key property of all merge trees in the media-on-demand model is that a stream started at time t_j can only merge to a stream started at an earlier time $t_i < t_j$. This defined a *total ordering* over the client

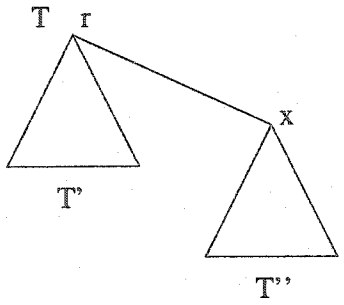

Figure 3.2: The recursive structure of a merge tree T with root r. The last arrival to merge directly to r is x.

requests. A key property of optimal merge trees in the media-on-demand model was the *preorder traversal property*, which stated that a preorder traversal of the merge tree yields the arrival times in order. This allowed a recursive decomposition of any merge tree in a natural way as shown in Figure 3.2. The recursive decomposition along with the total ordering allowed a complete search over all possible merge trees in the form of a dynamic programming solution. We will show that artificial arrival times define a total ordering over the client requests in the live broadcast with time-shifting model, and that optimal merge trees do not have the preorder traversal property with respect to this total ordering. Thus, we need to use another technique to derive a polynomial time optimal algorithm (if one exists).

Lemma 5 *A stream X can merge with stream Y only if $a(Y) \leq a(X)$.*

Proof: For the purpose of contradiction, assume that $a(Y) > a(X)$. This means that at any given time t, stream Y is broadcasting an earlier segment than stream X, which implies that stream X will never receive any segment from stream Y that it does not already have. Thus, stream X will never merge with stream Y, and we have a contradiction. □

Using Lemma 5, we can define a total ordering over the client requests using artificial arrival times (breaking ties using actual arrival times). In other words, X comes before Y if $a(X) < a(Y)$, or $a(X) = a(Y)$ and $t(X) < t(Y)$. Does a preorder traversal of an optimal merge tree yield the artificial arrival times in order? Unfortunately, our next example shows that this is not always the case.

Consider the following sequence of client requests (ordered by artificial arrival time):

$$c_1 = (5, 0)$$
$$c_2 = (16, 10)$$
$$c_3 = (7, 0)$$

Figure 3.3(a) shows the optimal merge pattern. Note that this tree is not preorder with respect to the three arrivals c_1, c_2, and c_3. Parts (b), (c), and (d) show all the possible preorder merge trees. The tree shown in part (b) is not completed because no matter how you complete the tree its cost is greater than the optimal tree shown in part (a). The final three preorder trees are created from the tree in part (b) by making c_3 a child of the root, c_3 a child of c_1, or c_3 a child of c_2.

Being unable to show that an optimal merge tree has the preorder traversal property, we cannot simply extend the dynamic programming solution used for the media-on-demand model [7]. To further complicate the situation, an optimal algorithm might even need to start a stream in the absence of a client request. As an example, the two possible solutions for the arrivals $X = (4, 1)$ and $Y = (5, 3)$ are shown in Figure 3.4. Note that the optimal solution (on the left) requires stream Y to start one unit before request Y actually arrives. These issues make the derivation of an optimal algorithm in the live broadcast with time-shifting model considerably harder than it was in the media-on-demand model.

3.3 Online Algorithms

It may seem trivial to apply the stream merging algorithms described in Chapter 2 to the live broadcast with time-shifting model. However, this is not the case due to the total ordering imposed by artificial arrival times as shown in Lemma 5. This means that an earlier arriving client can actually merge to one that arrives in the future, which was definitely not the case in the media-on-demand model. Another difference is that there can be two distinct clients who arrive at the same time but request different starting segments. As an example, consider the ERMT algorithm, which was described in Section 2.2.3. Recall that the ERMT algorithm has each group of clients attempt to merge with the closest stream that it can successfully merge with. Figure 3.5 shows two possibilities when the

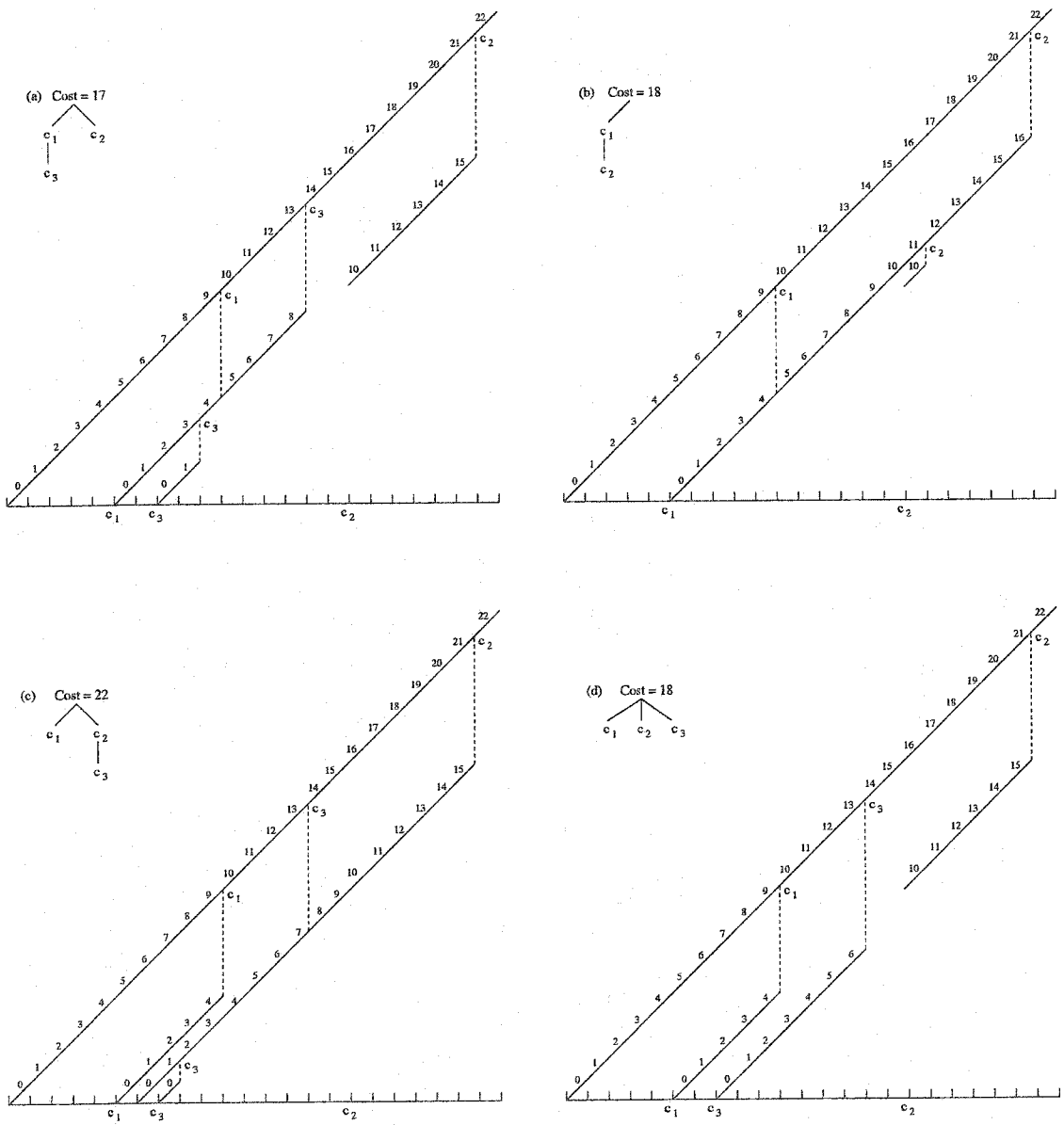

Figure 3.3: Example showing that the optimal merge tree is not always preorder with respect to artificial arrival times. Part (a) shows an optimal merge tree for the three requests (ordered by artificial arrival time) $c_1 = (5, 0)$, $c_2 = (16, 10)$, $c_3 = (7, 0)$. Parts (b)-(d) show the possible preorder merge trees (or partial merge trees) for these three arrivals. Part (b) shows a partial merge tree because no matter how you complete the tree, its cost is at least 18, and thus cannot be an optimal merge tree for these arrivals.

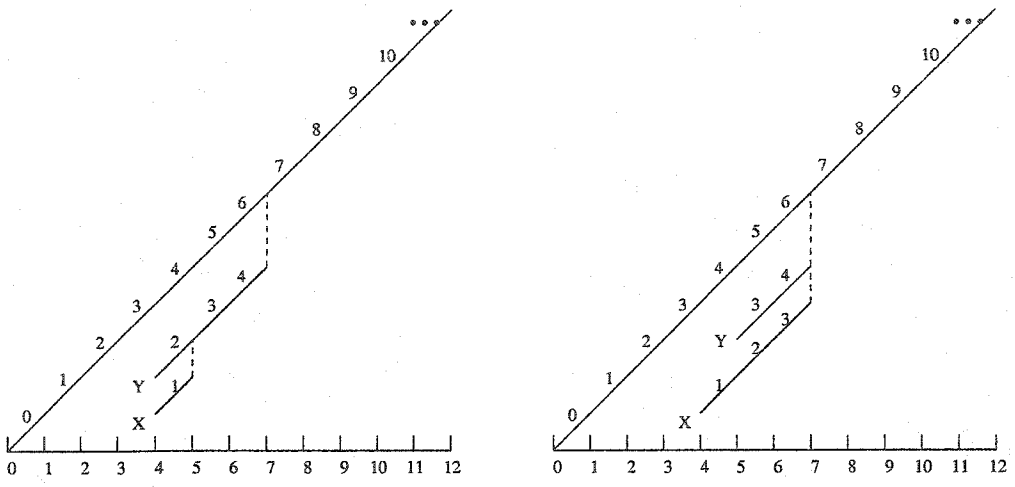

Figure 3.4: The two possible solutions for the arrivals $X = (4, 1)$ and $Y = (5, 3)$. Note that for the optimal solution (on the left), stream Y is started one unit before request Y actually arrives.

client requests $X = (7, 0)$ and $Y = (7, 3)$ both arrive at time 7. The figure does not show complete merge patterns, but instead shows only the next merge from the point of view of each client. This example shows that even if we only consider the next merge for each stream (as the ERMT algorithm does), we still have ambiguity due to the fact that we can have more than one distinct arrival at the same time. Which one should the algorithm use? As we can see, subtle differences from the media-on-demand model mean that we cannot simply run the stream merging algorithms in the live broadcast with time-shifting model without first making a few changes to the algorithms.

In this section, we adapt one of the algorithms that we studied in Chapter 2. In particular, we adapt the dyadic algorithm since it achieved a nice trade-off between performance and complexity. This is a good first step since we showed that all of the recently proposed stream merging algorithms had very good performance in the media-on-demand model. We leave the modification of the other stream merging algorithms for future work.

The Time-Shift Dyadic Algorithm: We begin with an example that illustrates the need to modify the dyadic algorithm. Recall from Definition 1 that each client request

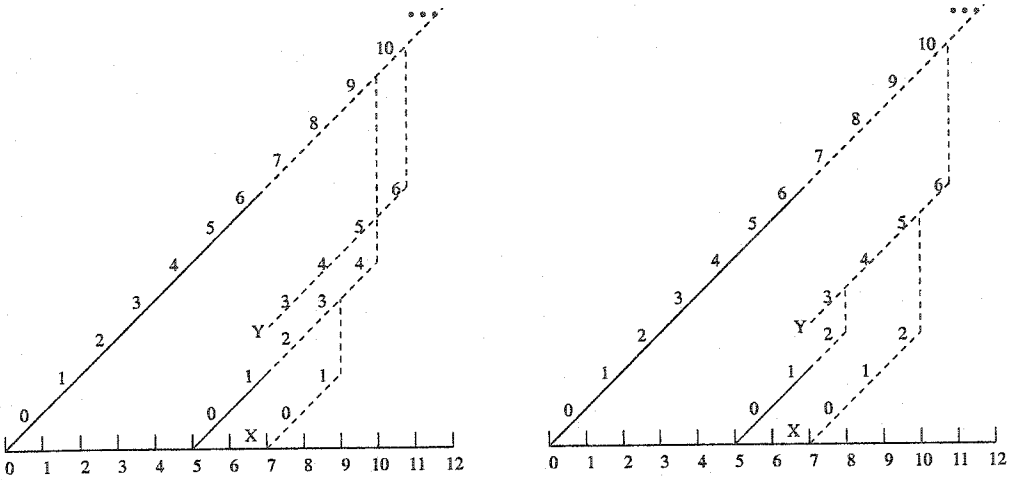

Figure 3.5: Two possibilities for the ERMT algorithm at time 7, with the arrival of both $X = (7, 0)$ and $Y = (7, 3)$. The dashed lines show the next merge from the point of view of each client in the absence of any other events. As a result, the figures do not show complete merge patterns. However, they do show that even if we only consider the next merge for each stream, we still have ambiguity due to the fact that we can have more than one distinct arrival at the same time.

X is represented by the pair $(t(X), f(X))$. Consider the three client requests $c_1 = (6, 0)$, $c_2 = (9, 0)$, and $c_3 = (10, 3)$. The merge patterns defined by the original dyadic algorithm with $\alpha = 2$ and $\beta L = 16$ are shown in Figure 3.6. Note that c_3 falls into the dyadic interval for c_2, but by Lemma 5 the merge is impossible because $a(c_3) < a(c_2)$. In addition, c_3 could have merged with c_1, but they are not in the same dyadic interval, so the merge is not allowed by the dyadic algorithm. Allowing c_3 to merge to c_1 is more efficient in this case, as can be seen in Figure 3.7. Although the savings is small in this example (only 1 unit), we can see how the dyadic intervals do not work in the same way for the time-shift model.

The problem in the above example is that an earlier arrival can merge to a later arriving one, which was not the case for the media-on-demand model. Lemma 5 showed that possible merge patterns in the time-shift model are determined by artificial arrival times instead of actual arrival times. In other words, a client X can merge with a client Y if $a(Y) \leq a(X)$. We modify the dyadic algorithm to operate using artificial arrival times. Instead of using a stack data structure to represent the dyadic intervals, we now use a list because the artificial arrival times of successive client requests are not always monotonically nondecreasing. This

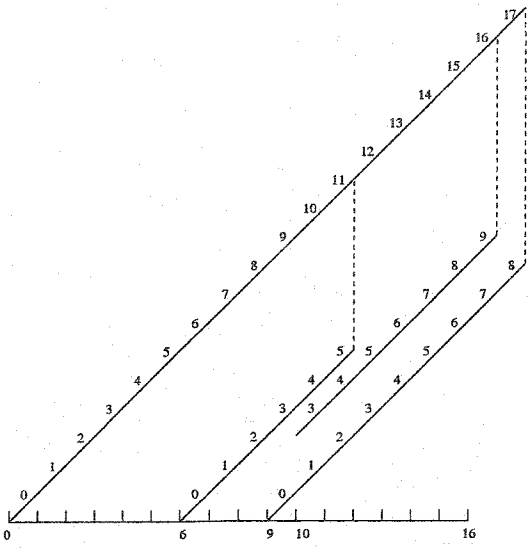

Figure 3.6: Illustration of the merge patterns for the original dyadic algorithm with $\alpha = 2$ and $\beta L = 16$. The merge cost is 22 units.

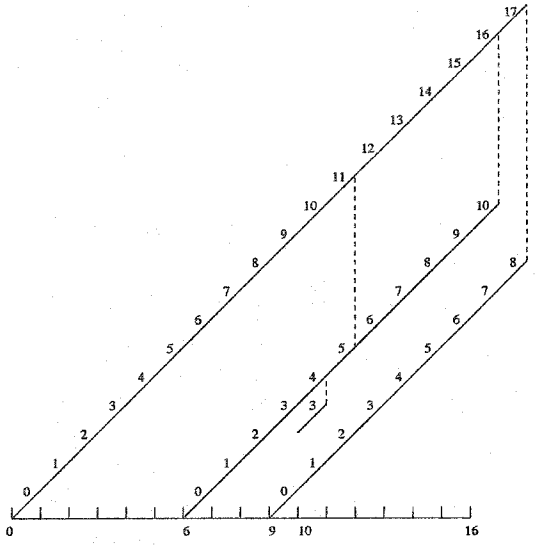

Figure 3.7: Illustration of the merge patterns for the dyadic algorithm assuming the intervals use artificial arrival times with $\alpha = 2$ and $\beta L = 16$. The merge cost is 21 units.

means that the only way we can be sure that an interval is no longer needed is if the root stream for that interval has terminated. The online version of the time-shift dyadic algorithm can be formulated as follows:

1. At the time t of a new request (with artificial arrival time a), search back from the front of the list to find the interval $[t_a, t_r)$ such that $t_a \leq a < t_r$ and the stream represented by the interval has not yet terminated. Along the way, remove from the list all pairs representing streams that have already terminated, leaving those whose streams are still active.

2. There are now two possibilities:

   (a) There is no such interval $[t_a, t_r)$ meeting the above criteria. In this case, the new arrival merges directly to the live root stream. The interval $[a, \infty)$ is placed at the back of the list.

(b) Let s be the root stream of the interval $[t_a, t_r)$. Build the pair $[a, r)$, where

$$r = t_a + (t_r - t_a) \max\{\alpha^{-k+1} : \alpha^{-k}(t_r - t_a) < a - t_a\}.$$

The new arrival becomes a child of stream s, and its interval $[a, r)$ is inserted into the list immediately in front of $[t_a, t_r)$.

Notice that the above algorithm uses ∞ as the time after which no arrivals are allowed to merge to certain streams. This is because we do not have an explicit stream length in the live broadcast with time-shifting model. We can implement dyadic interval partitioning by assuming a "unit" size (the smallest possible interval). Then we can find which interval a falls in by finding the index $i \geq 0$ such that $\alpha^i \leq a < \alpha^{i+1}$.

The time-shift dyadic algorithm imposes a structure that is useful in the time-shift model. For example, recall that we showed in Figures 3.6 and 3.7 how the original dyadic algorithm misses out on merge opportunities. The time-shift dyadic algorithm would generate the more efficient merge pattern shown in Figure 3.7.

3.4 Simulation Results

The merge-once algorithm is the simplest stream merging solution. Each client listens to the live root stream and its own dedicated channel that broadcasts the $t - w$ segments required for the client to merge with the live stream. After this one merge, the client listens to the live root stream for the remainder of the broadcast. Recall that the merge-once stream merging solution provides a big savings in server bandwidth compared to the unicast solution, as was shown in Figure 1.3. Because we do not have an optimal algorithm, the merge-once algorithm provides us with a good starting point to study the performance of our time-shift dyadic algorithm. We also compare our time-shift dyadic algorithm to a modified version of the greedy stream merging algorithms of Bar-Noy and Ladner [6]. These algorithms did not have very good competitive performance for the media-on-demand model, but did perform better than the simple merge-once algorithm and are relatively straightforward to implement. Consider the path $\langle R = X_0, X_1, \ldots, X_n \rangle$ in a merge tree from the root to the rightmost leaf in the merge tree. The nodes along this path are used as potential parents

for the next arrival. We use the two greedy rules which were introduced by Bar-Noy and Ladner [6] to select the parent.

1. *Best fit rule.* The best fit rule selects the parent X_i such that the increment in merge cost is minimized.

2. *Nearest fit rule.* The nearest fit rule selects the parent X_i that is the furthest away from the root.

We generate client arrival times using a Poisson arrival process, and generate first requested segments to be at the start of an hour using a uniform random distribution (the offset w is chosen such that $t - w$ is at the start of an hour). This simulates the situation where people want to start a program at the beginning. Our results show the time-shift dyadic algorithm with $\alpha = 2$. Bandwidth is measured as the number of concurrent streams that the server uses to satisfy the client requests.

Figure 3.8 shows the average bandwidth usage, and Figure 3.9 shows the maximum bandwidth usage over 24 hour experiments with different client request intensities. We assume an unlimited amount of time-shifting (clients can request to start receiving the broadcast at the beginning of our experiment). The x-axis plots the mean inter-arrival rate over our 24 hour experiments, and the y-axis plots either the average or maximum number of concurrent channels on a log scale. We can see that the merge-once algorithm performs very poorly compared to the other stream merging algorithms, and that the time-shift dyadic algorithm significantly outperforms the other algorithms. This is consistent with our results in Chapter 2 for the media-on-demand model, and is what we would expect. It is also interesting to note that there is a pretty significant difference between average and maximum bandwidth usage for all of the algorithms. This is important for real servers that would have to deny (or delay) service to clients when they hit their maximum number of channels.

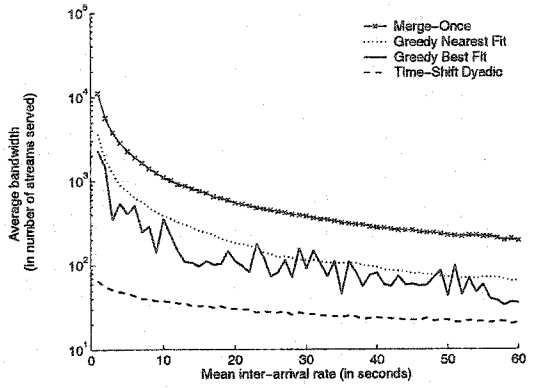

Figure 3.8: Performance of the merge-once, greedy, and time-shift dyadic algorithms for the live broadcast with time-shifting model. Comparison of average bandwidth usage (in number of concurrent streams).

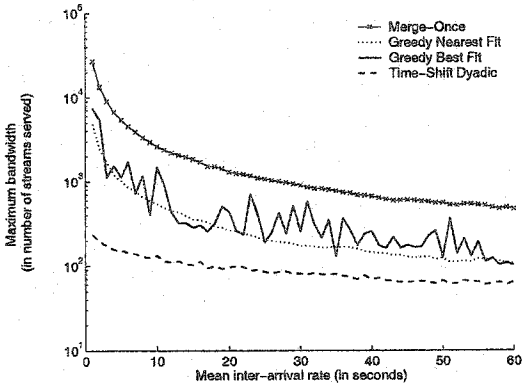

Figure 3.9: Performance of the merge-once, greedy, and time-shift dyadic algorithms for the live broadcast with time-shifting model. Comparison of maximum bandwidth usage (in number of concurrent streams).

3.5 Conclusions and Future Work

We have presented the live broadcast with time-shifting model. We identified subtle differences between this model and the media-on-demand model, and showed that these differences make it challenging to find an optimal stream merging solution for the live broadcast with time-shifting model. Next, we presented online stream merging algorithms for this model. In particular, we presented modifications of the merge-once, greedy best and nearest fit [6], and dyadic [17] algorithms. All of these exhibit a huge improvement over the unicast solution that dedicates a stream for each client request. The time-shift dyadic algorithm has the best performance, and outperforms its closest competitor by a large margin.

We have just begun to explore the area of efficient stream merging algorithms for the live broadcast with time-shifting model. There are many remaining open problems in this area, and we list a few of the more interesting ones.

1. Derive a polynomial time optimal algorithm (or show that the problem is NP-hard). An optimal algorithm is important when trying to determine how well our algorithms perform, and how big the room for improvement is.

2. Experiment with other client arrival and segment request distributions. It would be extremely helpful to gather real data to see what types of patterns occur in practice.

3. Modify other stream merging algorithms to operate in the time-shift model. We presented a modification of the dyadic algorithm, and could try to do the same for the other stream merging algorithms that were studied in Chapter 2.

4. Perform an in-depth study similar to what we did in Chapter 2 for the media-on-demand model.

5. Design stream merging algorithms that minimize other metrics. We focused on minimizing the total server bandwidth usage. Another important metric to minimize is the maximum server bandwidth usage. As seen in Figures 3.8 and 3.9, there is a fairly significant difference between the average and maximum bandwidth usage for our algorithms. The merge tree model is useful for the total bandwidth metric, but a different model is needed for the maximum bandwidth metric (this is also true for the media-on-demand model).

6. Extend our study to other stream merging models. We focused on the receive-two model where clients can simultaneously receive data from at most two streams. Bar-Noy and Ladner also studied the receive-all model, and the case where clients have bounded buffers [7]. We could try to extend our analysis to these models.

Chapter 4

KEY DISTRIBUTION FOR SECURE GROUP COMMUNICATIONS

We study the problem of multicast key distribution for group security. Secure group communication systems typically rely on a group key, which is a secret shared among the members of the group. This key is used to provide access to group data by encrypting all group communications. Because groups can be large and highly dynamic, it becomes necessary to change the group key in a scalable and secure fashion when members join and leave the group. We present a series of algorithms for solving this problem based on key trees. The algorithms attempt to minimize the worst-case communication cost of updating the group and auxiliary keys by maintaining balanced key tree structures. We focus on the trade-off between the communication cost due to the structure of the tree and that due to the overhead of restructuring the tree to maintain its balanced structure. The algorithms are analyzed for worst-case tree structure bounds and evaluated empirically via simulations.

The remainder of this chapter is organized as follows: in Section 4.1, we provide background material and define the model used in our work. We present our three new online algorithms in Section 4.2. We study the scalability of our algorithms both analytically and empirically by providing the worst-case analysis of their tree structures in Section 4.3, and by using simulations in Section 4.4. Finally, we conclude and provide directions for future work in Section 4.5.

4.1 Model and Preliminaries

As mentioned earlier, multicast key distribution refers to the problem of managing and distributing the set of keys that are used to provide access to group data. We consider solutions that handle arbitrary re-key sequences. Previous work showed that communication cost can be reduced if assumptions are made on the statistics of the re-key sequences [43, 9, 55]. We study the general case when we do not know anything about re-key patterns.

Following previous work [12, 49], we abstract away secret key cryptographic details by assuming that any member holding a key k can decrypt a message that was encrypted with k, but no coalition of members not holding k will be able to decrypt the message or gain any information (even if they are computationally unbounded). Under this assumption, the security of the group depends only on the secure management and distribution of the group and auxiliary keys.

Our model is essentially the same as the one that was introduced by Canetti *et al.* [12] and enhanced by Snoeyink *et al.* [49]. We assume that authentication and individual key exchange are provided, and focus on the secure management and distribution of the group and auxiliary keys to ensure that only valid members have access to the group data. Define the *key server* to be the entity responsible for this job. The key server must handle two re-key operations: ADD(u) to add member u to the group, and DELETE(u) to remove member u from the group. There are two types of security that a key distribution algorithm can provide: *backward security*, meaning that a new member is not able to decrypt past group communications, and *forward security*, meaning that a departing member is not able to decrypt future group communications. We define a key distribution algorithm to be secure if for any set of adversaries, after any sequence of update operations, the adversaries cannot distinguish the group key k_M from a random key. This definition provides both backward and forward security and is met if all keys are pseudo-random, all keys that member u receives as a result of an ADD(u) operation are new, and all keys that member u held before a DELETE(u) operation are removed.

A *key tree* is a data structure representing the set of keys held by each group member. Each node is labeled with a key. If i is a node in a key tree, then the label of i is a group key for the subgroup G_i consisting of all leaves in the subtree rooted at i. The root label is the group key for the entire multicast group. The label of a leaf is the individual key for that one member, a group of one. Generally, a new key k is distributed to subgroup G_i using the key k_i (the label of node i) using the encrypted message $E(k, k_i)$, where the key k is encrypted with the key k_i.

Define *communication cost* as the number of encrypted messages required by the key server to update the keys due to group membership change. This is the metric we are

interested in minimizing. We assume that there is one key per message. In a real system, multiple keys would likely be combined into one message. These systems also have to worry about packet loss, and try to pack keys into messages to minimize communication overhead. We do not consider these issues, so we use the normalized metric of one key per message to measure communication cost. In addition, this matches the model used by Snoeyink *et al.* [49].

We borrow definitions on key tree properties from Snoeyink *et al.* [49]. The *ancestor weight* w_i of a node i is the sum of the degrees of all nodes on the path from node i to the root. Alternatively, w_i can be computed using the following recursive definition. The ancestor weight of the root r is $w_r = 0$. For $i \neq r$, define p to be the parent of i, and define $deg(p)$ to be the degree for the node p. Then $w_i = w_p + deg(p)$. Note that the ancestor weight of a node i represents the number of edges that must disappear from the tree when i is removed. This is because all links to these compromised keys can no longer be used. Most of these edges need to be replaced, representing the distribution of a changed key. Therefore, ancestor weight is a good estimate of the communication cost required to remove a member. The *tree weight* $w(T)$ is the maximum ancestor weight of any node in tree T. Note that $w(T)$ is achieved by some leaf in the tree. The *optimal tree weight* $w(n)$ is the minimum value of $w(T)$ for n members over all possible T, where the number of members equals the number of leaves in the tree. Define d_u to be the depth of member u in the tree, that is, the length of the path, in edges, from the node u to the root.

Asymmetry of Re-key Operations: It is interesting to note that there is an asymmetry between the two re-key operations. In particular, the cost of an ADD(u) operation is proportional to u's depth d_u, while the cost of a DELETE(u) operation is proportional to u's ancestor weight w_u. We illustrate this fact using the following example on the key trees shown in Figure 4.1. In the descriptions that follow, if k represents a key that must be changed, then we use k' to mean the new key used to replace k. We consider the two re-key operations below:

1. ADD(u_9), shown in the top part of Figure 4.1. Assume that node k_3 is chosen to be the parent for u_9. Then we need to change the $d_{u_9} = 2$ keys k_M and k_3. We can

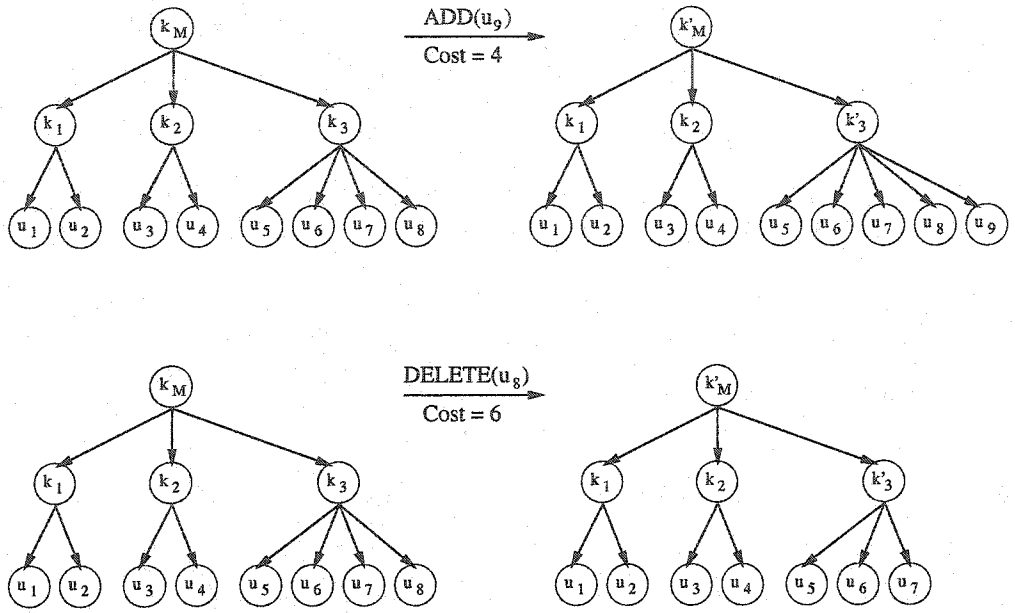

Figure 4.1: An illustration of the asymmetry between re-key operations. The top part shows how the key tree is modified for an ADD operation, while the bottom part shows how the key tree is modified for a DELETE operation.

change each key with 2 encrypted messages because the old keys act as secrets that u_9 does not know. In this example, we have a total cost of $2 \cdot d_{u_9} = 4$ encrypted messages: $E(k'_3, k_3)$, $E(k'_3, u_9)$, $E(k'_M, k_M)$, $E(k'_M, u_9)$.

2. DELETE(u_8), shown in the bottom part of Figure 4.1. Removing u_8 forces us to change the two keys k_M and k_3. Unlike the ADD(u) situation, this time the old keys cannot act as secrets that u_8 does not know. The tree is updated from bottom to top, requiring the $w_{u_8} - 1 = 6$ encrypted messages: $E(k'_3, u_5)$, $E(k'_3, u_6)$, $E(k'_3, u_7)$, $E(k'_M, k_1)$, $E(k'_M, k_2)$, $E(k'_M, k'_3)$.

In the example of ADD(u_9), the two messages $E(k'_3, u_9)$ and $E(k'_M, u_9)$ are sent with the same key u_9. As we mentioned earlier, in a real system these would likely be combined into one message encrypted with u_9. However, we measure normalized cost, and treat these as two encrypted messages.

Security Requirements: Before moving on, we briefly describe the security requirements that guide the design of our algorithms.

1. *Provide complete security against member collusion.* Previous work provided weaker security, saying that their system is resistant for up to k colluding members [25]. They focused on establishing a secret with a chosen subset of possible receivers for broadcast TV. In that setting, it might be more cost-effective to reduce communication cost at the expense of slightly reduced security.

2. *Maintain backward and forward security on every group membership change.* Periodic re-keying [34, 47] was introduced to lower communication cost requirements, but does not provide backward and forward security during the interval between re-key times.

4.2 Online Algorithm Design

The optimal multicast key trees (those with minimum worst-case lower bounds) described by Snoeyink *et al.* [49] and Poovendran and Baras [43] were static, no descriptions were given on how to dynamically maintain the optimal trees. Most of the previous work with online algorithms did not attempt to maintain balanced tree structures, which could lead to bad worst-case performance. We investigate the design of online multicast key tree algorithms that attempt to minimize worst-case communication cost. There are two components that contribute to the communication cost for online algorithms:

1. *Tree structure.* The ancestor weight w_i for a given node i is the number of edges that must disappear for a DELETE(u) operation. Although this does not always measure the exact communication cost, it is usually close because a new edge often replaces an old one (representing the distribution of a changed key). The tree weight $w(T)$ for a given tree T is calculated as the maximum w_i over all leaves i in tree T. Therefore, when considering worst-case cost, a good tree structure is one with low tree weight.

2. *Restructuring costs.* The process of dynamically maintaining a desired tree structure may require us to move nodes to different locations in the tree. This may force us to

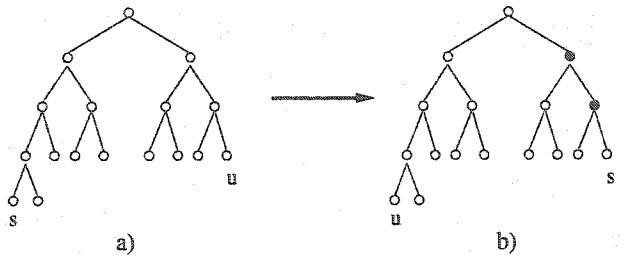

Figure 4.2: Illustration of switching to maintain balanced for a DELETE(u) operation. The tree is considered to be balanced if the distance from the root to any two leaf nodes differs by at most 1. Removing u from the tree shown in part a would cause the tree to become unbalanced. Part b shows a switch that will allow the tree to remain balanced after u is removed. The darkened nodes in part b show the extra keys that must be changed due to the switch.

change some keys as a result of restructuring, and can cost us additional encrypted messages.

When designing our online algorithms to minimize worst-case cost, we need to trade-off between tree structures with low tree weight, and the restructuring costs needed to maintain those tree structures. The degree-k key trees of Wong *et al.* [53] perform no restructuring. This minimizes restructuring cost, but can increase tree structure cost if the tree becomes highly unbalanced. Moyer *et al.* [37] addressed this problem by maintaining balanced binary trees (the distance from the root to any two leaf nodes differs by at most 1). Maintaining balance for ADD(u) operations are easy since the algorithm can control the placement of the new member. However, DELETE(u) operations are harder. Moyer *et al.* maintained their balanced trees using a novel switching algorithm. The main idea is if the removal of u from the tree would cause it to become unbalanced, the location of u is switched with another member so that the removal of u from the new location allows the tree to remain balanced. An example of switching is shown in Figure 4.2. Assume u is the member being removed from the tree. If we simply remove u from the tree structure shown in part a, the tree would become unbalanced. Part b shows the result of switching members u and s so that the tree will remain balanced after the removal of u. Note that additional keys need to be replaced as a result of the switch. This is shown by the darkened nodes in the figure, and is the result of s obtaining new keys.

We say that the switching algorithm uses *global restructuring* because we need a global view of the tree structure in order to find a node to switch with. Note that the switching algorithm can be used to maintain any type of tree structure. For example, it can be used to maintain an optimal tree structure (one with minimum worst-case lower bounds) at all times by maintaining the optimal tree structures described by Snoeyink *et al.* [49]. This algorithm minimizes the tree structure costs. However, the cost of restructuring can be very high because large portions of the tree may need to be modified due to switching. This is illustrated with a comparison of the degree-3 algorithm (no restructuring), the balanced binary tree algorithm of Moyer *et al.*, and the optimal tree weight with switching algorithm over a sequence of 177,147 random DELETE(u) operations from an initial tree of that size. We choose the initial tree to have $3^{11} = 177,147$ members because both the degree-3 and optimal tree weight with switching algorithms arrive at the same tree structure (using their respective algorithms for handling ADD(u) operations) when the number of members $n = 3^i$. For all of our graphs, each point represents either the average or maximum value over 1,772 consecutive operations.

The restructuring costs for the switching algorithms are quite high, as seen in Figures 4.3 (average restructuring costs) and 4.4 (maximum restructuring costs). These high restructuring costs eliminate any advantage of using good tree structures. This can be seen by comparing the tree structure costs to the communication costs in Figures 4.5 and 4.6 for average values, and Figures 4.7 and 4.8 for maximum values. Note that the tree structure and communication cost graphs show the same range on the y-axis. The switching algorithms have good tree structure, but perform poorly due to the high amounts of restructuring.

We have seen that switching algorithms can be inefficient due to high restructuring costs. In the remainder of this section we describe three new online algorithms for maintaining multicast key distribution trees using *local restructuring*, meaning that there are rules that can be applied while considering only a small portion of the tree. The degree-k tree algorithm minimizes restructuring costs, while the optimal tree with switching algorithm minimizes tree structure costs. Our algorithms achieve different points in the trade-off between tree structure and restructuring cost. An evaluation of their performance is presented in Sections 4.3 and 4.4.

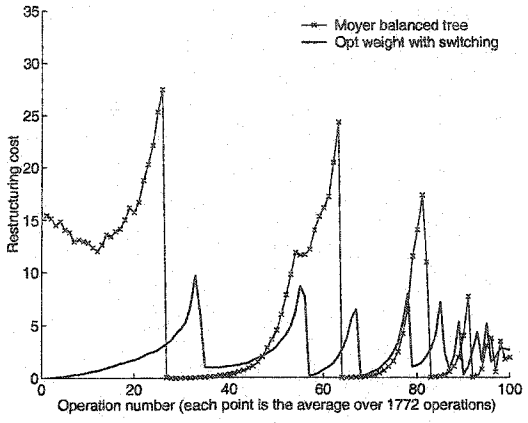

Figure 4.3: Restructuring costs for the balanced binary tree algorithm of Moyer *et al.* and the optimal tree weight with switching algorithm. Experiment consists of 177, 147 random deletions from an initial tree of the same size. Each point is the average value over 1,772 consecutive operations.

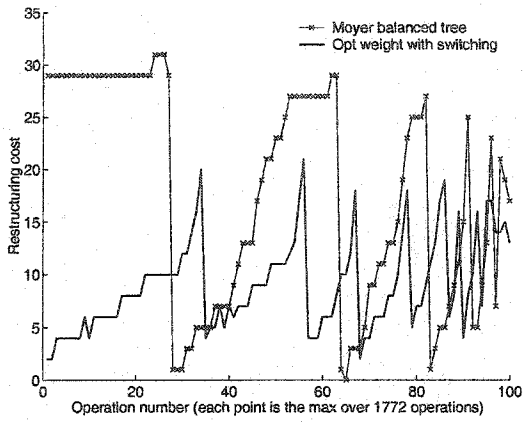

Figure 4.4: Restructuring costs for the balanced binary tree algorithm of Moyer *et al.* and the optimal tree weight with switching algorithm. Experiment consists of 177, 147 random deletions from an initial tree of the same size. Each point is the maximum value over 1,772 consecutive operations.

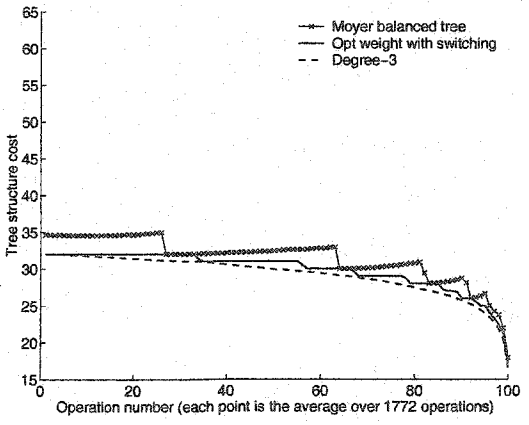

Figure 4.5: Tree structure cost measured as the ancestor weight of deleted members for the balanced binary tree algorithm of Moyer *et al.*, the optimal tree weight with switching algorithm, and the degree-3 algorithm. Experiment consists of 177, 147 random deletions from an initial tree of the same size. Each point is the average value over 1,772 consecutive operations.

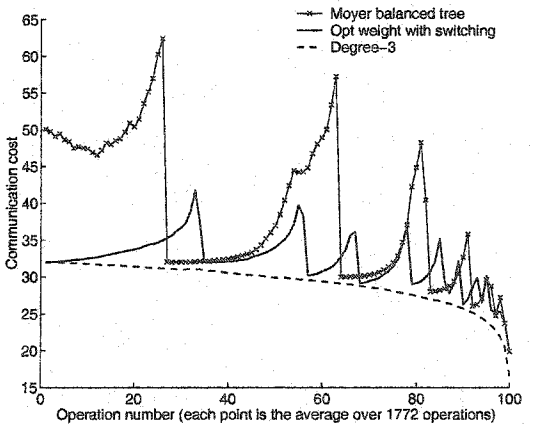

Figure 4.6: Communication cost measured as the number of encrypted messages for the balanced binary tree algorithm of Moyer *et al.*, the optimal tree weight with switching algorithm, and the degree-3 algorithm. Experiment consists of 177, 147 random deletions from an initial tree of the same size. Each point is the average value over 1,772 consecutive operations.

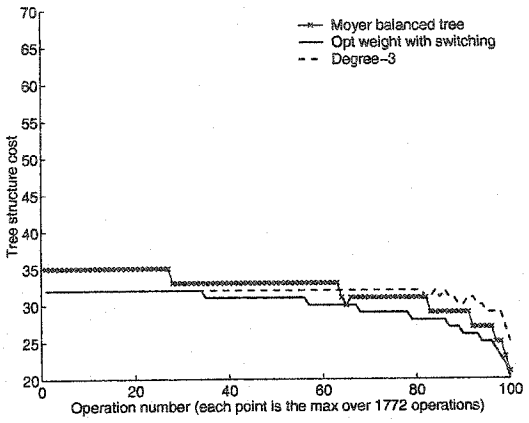

Figure 4.7: Tree structure cost measured as the ancestor weight of deleted members for the balanced binary tree algorithm of Moyer *et al.*, the optimal tree weight with switching algorithm, and the degree-3 algorithm. Experiment consist of 177,147 random deletions from an initial tree of the same size. Each point is the maximum over 1,772 consecutive operations.

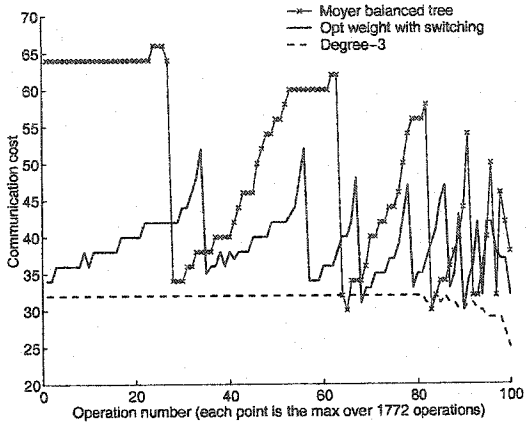

Figure 4.8: Communication cost measured as the number of encrypted messages for the balanced binary tree algorithm of Moyer *et al.*, the optimal tree weight with switching algorithm, and the degree-3 algorithm. Experiment consists of 177,147 random deletions from an initial tree of the same size. Each point is the maximum over 1,772 consecutive operations.

4.2.1 B-Tree Algorithm

B-trees were introduced by Bayer and McCreight [8]. A B-tree of order t is one where all internal nodes except the root have $[\lceil t/2 \rceil, t]$ children (where $t \geq 3$), and all leaves are at the same depth in the tree. The root can have $[2,t]$ children. In our application, members are always added at an insertion point that minimizes the ancestor weight for the new member. Such an insertion point can be found by storing the possible insertion points in a priority queue. Other than this modification, we use the same insertion/deletion rules as the original B-trees. These rules are shown below for the general case when there are $n \geq 2$ members. The special cases when $n = 0, 1$ are easy to handle, and are omitted.

1. *ADD(u) operation.* Find the insertion point p that will minimize the ancestor weight for the new member. Add u as a child of p using the following algorithm.

   (a) If p has $x < t$ children, add u as a child of p.

(b) If p has t children, create a new node p'. Add u as a child p', and move $\lfloor t/2 \rfloor$ children of p to be children of p'. There are now two possibilities:

i. p has no parent. Create a new root r. Make p and p' the two children of r.

ii. p has a parent g. Call this function recursively to add p' as a child of g.

2. *DELETE(u) operation.* Delete u using the following algorithm.

(a) If p has $x > \lceil t/2 \rceil$ children, remove u.

(b) If p has $\lceil t/2 \rceil$ children, then there are two possibilities:

i. p is the root. Remove u. If p has only one child c remaining, remove p, leaving c as the new root of the tree.

ii. p is not the root. Remove u (it is no longer a child of p). Call p' the sibling of p with the most children. If p' has $x \geq \lceil t/2 \rceil + 1$ children, move a child of p' to be a child of p. Otherwise, move the $\lceil t/2 \rceil - 1$ remaining children of p to be children of p', and call this function recursively to delete p.

4.2.2 Height-Balanced 2-k Tree Algorithm

B-trees require that all leaf nodes be at the same depth in the tree. Because of this strict requirement, we may incur high restructuring costs to maintain this tree structure. We design an algorithm with a relaxed definition of a "balanced" node which we expect will lead to a different point in the trade-off between tree structure and restructuring cost.

Define a node to be *height-balanced* if the subtrees of the node differ in height by at most 1. A *height-balanced 2-k tree* is one where all internal nodes have degree $2 \leq d \leq k$, and all nodes in the tree are height-balanced. Height-balanced 2-k trees are a generalization of AVL trees (which are binary, or height-balanced 2-2 trees). Define $h(T)$ to be the height of tree T. We describe an algorithm to merge two height-balanced 2-k trees T_1 and T_2 into a height-balanced 2-k tree T with height $\max(h(T_1), h(T_2)) \leq h(T) \leq 1 + \max(h(T_1), h(T_2))$. This is a generalization of the algorithm for merging two AVL trees that was presented by Rodeh *et al.* [45].

$Merge(T_1, T_2)$. Without loss of generality, assume that $h(T_1) \geq h(T_2)$. Define C_{min} to be the child of T_1 with minimum height, and k to be the maximum allowed degree for any node in the tree. The algorithm maintains the invariant that each node is height-balanced, and that the resulting tree $T = Merge(T_1, T_2)$ has height $\max(h(T_1), h(T_2)) \leq h(T) \leq 1 + \max(h(T_1), h(T_2))$. There are 3 cases to consider (checked in the order they appear), and the first one that applies is used.

1. $h(T_1) > h(T_2)$, T_1 has degree $d < k$, and $h(T_2)$ is the same as at least 1 child of T_1. Add T_2 as a child of T_1, and return T_1.

   Because $h(T_2)$ is the same as a child of T_1, adding T_2 as a child of T_1 maintains the height-balance of T_1 and does not increase its height.

2. $h(T_1) = h(T_2)$ or $h(T_1) = h(T_2) + 1$. Create a new node T to be the parent for T_1 and T_2, and return T.

   The invariant holds since T is height-balanced, and has height $h(T) = 1 + h(T_1)$.

3. $h(T_1) > h(T_2) + 1$. Recall that C_{min} is a child of T_1 with minimum height. Let T be the result of replacing C_{min} in T_1 with $T_{new} = Merge(C_{min}, T_2)$, and return T.

   We know by the invariant that T_{new} is height-balanced, and

   $$\max(h(C_{min}), h(T_2)) \leq h(T_{new}) \leq 1 + \max(h(C_{min}), h(T_2)).$$

   Because C_{min} is a child of T_1, we know that

   $$\begin{aligned} h(T_1) - 2 &\leq h(C_{min}) \leq h(T_1) - 1 \\ h(T_1) - 1 &\leq 1 + h(C_{min}) \leq h(T_1). \end{aligned}$$

   Together with the case hypothesis $1 + h(T_2) < h(T_1)$, this implies that $h(T_2) \leq h(C_{min})$. This means that the invariant on T_{new} reduces to

   $$h(C_{min}) \leq h(T_{new}) \leq 1 + h(C_{min}) \leq h(T_1).$$

The invariant implies that T_{new} is balanced with its siblings. The invariant also implies that T_1 has maximum height among the children of T, and therefore, $h(T_1) \leq h(T) \leq 1 + h(T_1)$.

We can use the algorithm for merging height-balanced 2-k trees to handle re-key operations.

1. *ADD(u) operation.* Define T to be the existing height-balanced 2-k tree. Run the merge algorithm on tree T and a tree consisting of a single node representing the joining member u.

2. *DELETE(u) operation.* A DELETE operation requires that all keys on the path from u to the root be changed. We delete all nodes representing these keys, which corresponds to splitting the current tree T into $O(\log n)$ height-balanced 2-k subtrees. We insert all of these subtrees into a heap sorted by height and run the following algorithm:

   While there are two or more subtrees on the heap
   Pop the two shortest subtrees T_1 and T_2 off the heap and
   merge them together to produce the tree $T = Merge(T_1, T_2)$.

   While T has $d < k$ children, and the next subtree T_i on
   the heap has height that differs from T's children by at most 1
   Pop T_i from the heap and make it a child of T.

   Insert T back into the heap.
   The last remaining subtree is the new height-balanced tree.

   This algorithm is similar to the one given by Rodeh *et al.* [45] for AVL trees, however, we try to produce shorter trees by increasing the degree of our nodes (up to the maximum degree k).

4.2.3 *Weight-Balanced Tree Algorithm*

The B-tree and height-balanced 2-k algorithms attempt to balance the height of the tree. Because tree weight is a good estimate of tree structure cost, it is interesting to try to balanced the weight of the tree instead. This idea forms the basis for our most novel key tree algorithm. Define the *node weight* of a node i as the tree weight of the sub-tree rooted at i. A node is *weight-balanced* if the node weight of its children differ by at most 1. Define a weight-balanced tree as one where all nodes are weight-balanced.

We designed weight-balanced algorithms using local restructuring rules to maintain the balanced of the tree. Because these rules are enumerations of all possible unbalanced structures, and these unbalanced structures depend on the allowed node degrees, we were unable to come up with a general weight-balanced algorithm. However, we were able to design a weight-balanced 2-3 algorithm (all internal nodes have degree 2 or 3) and a weight-balanced 2-3-4 algorithm (all internal nodes have degree 2, 3, or 4).

We show the weight-balanced 2-3 rules for insertion in Figure 4.9, deletion in Figure 4.10, and local restructuring rules to restore weight-balance in Figures 4.11 and 4.12. Note that insertion rule b in Figure 4.9 is not needed because rules a and c are sufficient to cover all possibilities. However, rule b is an optimization we used to help keep the height of the tree from growing too fast. The weight-balanced 2-3-4 rules for insertion are shown in Figure 4.13, deletion in Figure 4.14, and local restructuring rules to restore weight-balance in Figures 4.15 and 4.16. Note that insertion rule c in Figure 4.13 shows three possibilities. The rules on the left and middle are optimizations used to help keep the height of the tree from growing too fast. The rule on the right shows the default case when the optimizations cannot be applied.

For each rule, the left part shows the subtree before applying the rule, the number above the arrow shows the change in the subtree root's node weight, the number below the arrow shows the communication cost for applying the rule, and the right part shows the subtree after applying the rule. Each node is labeled with its node weight. Nodes that do not show their node weight are not constrained, meaning that their node weight can be any value that allows it to maintain weight-balance with its siblings.

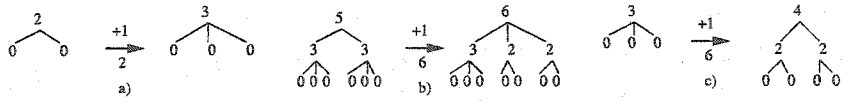

Figure 4.9: Illustration of the rules for adding a new member to a weight-balanced 2-3 tree.

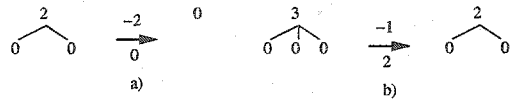

Figure 4.10: Illustration of the rules for deleting a member from a weight-balanced 2-3 tree.

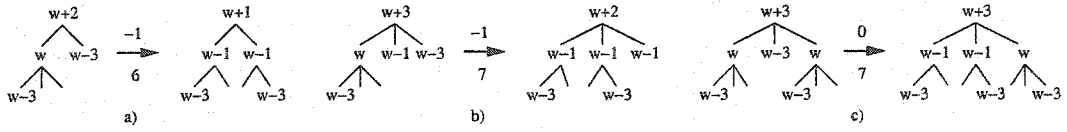

Figure 4.11: Illustration of the rules for handling a weight-imbalance caused by a delete in a weight-balanced 2-3 tree, where the maximum difference in node weight is 3. Note that these situations can only occur immediately after delete rule a shown in Figure 4.10.

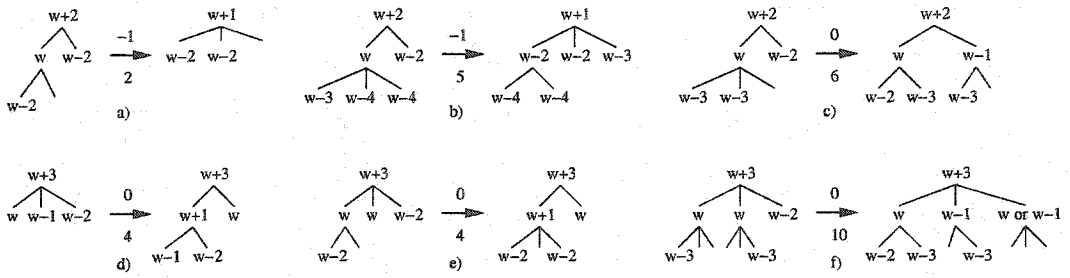

Figure 4.12: Illustration of the rules for handling a weight-imbalance caused by a delete in a weight-balanced 2-3 tree, where the maximum difference in node weight is 2.

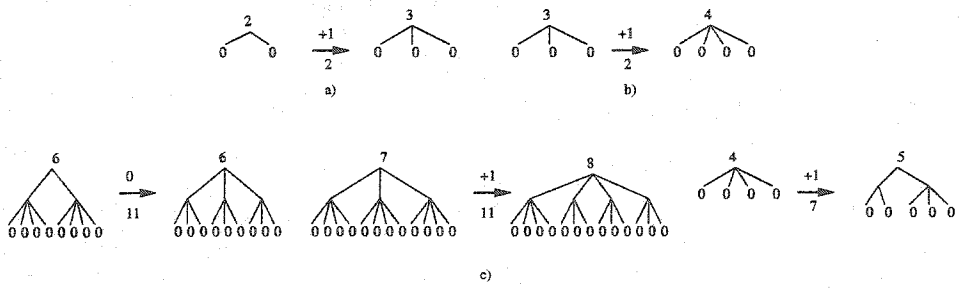

Figure 4.13: Illustration of the rules for adding a new member to a weight-balanced 2-3-4 tree.

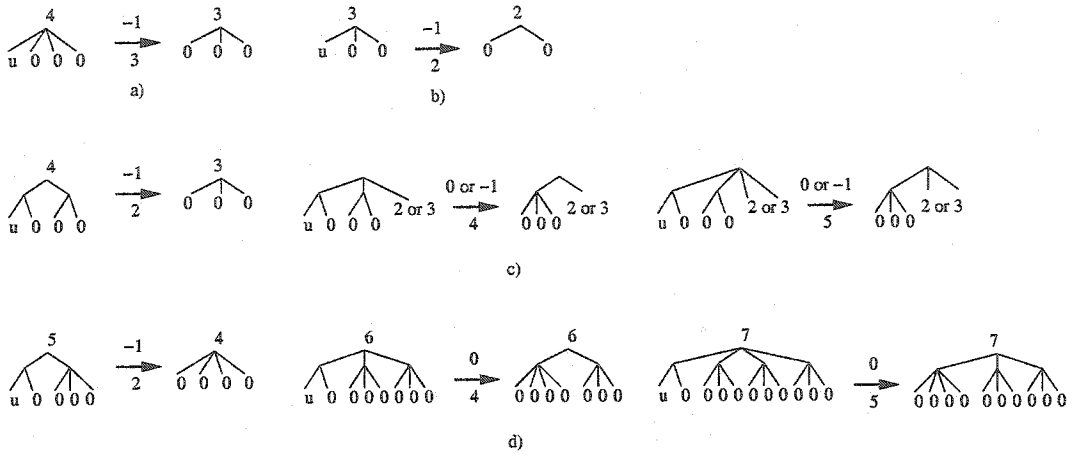

Figure 4.14: Illustration of the rules for deleting a member from a weight-balanced 2-3-4 tree. The member u is the one being deleted.

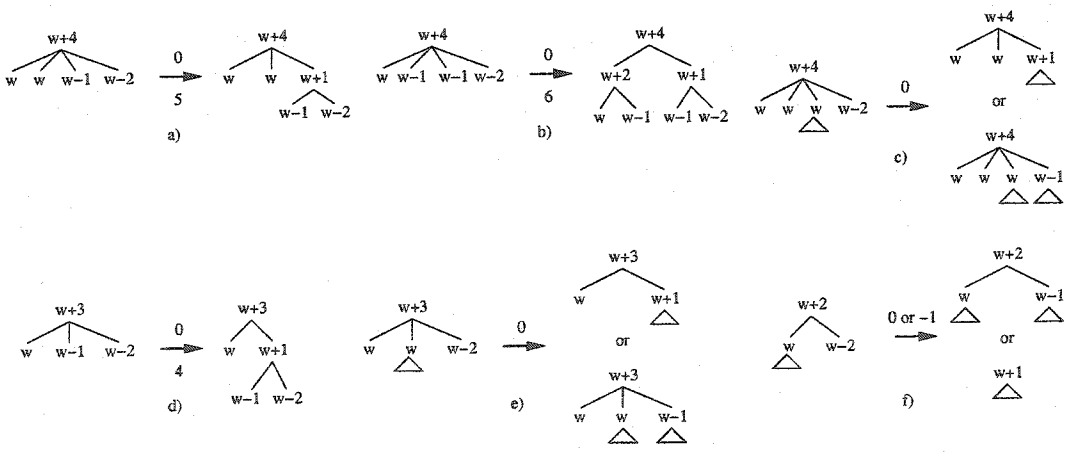

Figure 4.15: Illustration of the rules for handling a weight-imbalance caused by a delete in a weight-balanced 2-3-4 tree. Some rules show subtrees represented by triangles (and leave the cost blank). For these rules, the appropriate case is selected from the set of rules shown in Figure 4.16. The cost can be computed by adding the cost to change the subtree root key (its degree) to the cost of the rule that was used from Figure 4.16.

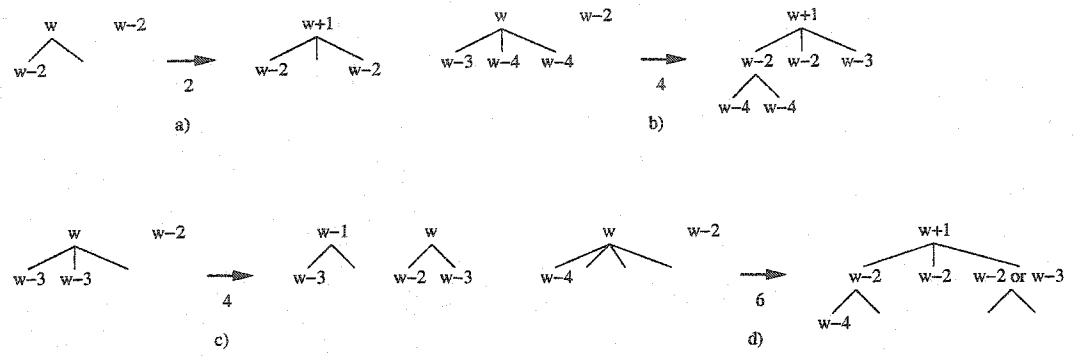

Figure 4.16: These rules are used for handling the weight-imbalances in weight-balanced 2-3-4 trees that are represented by triangles in Figure 4.15. The triangles have been expanded, and a rule appears for each possibility.

One thing to note is that there are no local restructuring rules to restore weight-balance after an insertion (only after a deletion). The following lemma shows that no such local restructuring rules are necessary because the tree resulting from an insertion must still be weight-balanced.

Lemma 6 *If T is a weight-balanced 2-3 or 2-3-4 tree and the insertion point p for a new member u is chosen to minimize the ancestor weight w_u of the new member, then applying the appropriate rule to add u as a child of p from Figure 4.9 (2-3 tree) or Figure 4.13 (2-3-4 tree) results in a tree T' that is still weight-balanced.*

Proof: Define $n(i)$ to be the node weight of a node i, and $s(i)$ to be a sibling of node i. Consider the path from the root to the insertion point p. Recall that the node weight of node i is defined as the tree weight of the subtree rooted at i, and that a node is weight-balanced if the node weight of its children differ by at most 1. For all $1 \leq i \leq p$ on the path from the root to the insertion point, it must be that $n(i) = n(s(i))$ or $n(i) = n(s(i)) - 1$. The fact that $n(i) \leq n(s(i))$ is true by definition of node p being an insertion point for the new member u that minimizes the ancestor weight w_u (otherwise i cannot be on this minimizing path). The fact that the node weights differ by at most 1 is true from the fact that the tree is weight-balanced.

Adding u to the tree using the insertion point p can increase the node weight of p by at

most 1 (see Figures 4.9 and 4.13). If p's node weight stays the same, then the tree remains weight-balanced. We only need to worry about the case when p's node weight increases by 1. In this situation, because the node weight of p was equal to or 1 less than its siblings, it must now be equal to or 1 greater than its siblings. This means p is still weight-balanced with its siblings. However, increasing p's node weight can cause the node weight of its parent g to increase by 1. But since g is on the path to the root, it has the same relationship with its siblings that p had with its siblings, and we can apply the same argument to g. In fact, we can recursively apply the same argument all the way up to the root, and therefore, the entire tree remains weight-balanced. □

We can use the rules for insertion, deletion, and restoring weight-balance to handle re-key operations.

1. *ADD(u) operation.* Similar to the B-tree ADD(u) operation, find an insertion point p that will minimize the ancestor weight for the new member. Add u as a child of p by applying the appropriate insertion rule from the set of rules shown in Figure 4.9 for weight-balanced 2-3 trees or Figure 4.13 for weight-balanced 2-3-4 trees. By Lemma 6, the resulting tree is still weight-balanced.

2. *DELETE(u) operation.* Delete u using the appropriate deletion rule from the set of rules shown in Figure 4.10 for weight-balanced 2-3 trees or Figure 4.14 for weight-balanced 2-3-4 trees. If applying the rule has caused a weight-imbalance higher in the tree, recursively apply the appropriate rule from the list of rules shown in Figures 4.11 and 4.12 for weight-balanced 2-3 trees or Figure 4.15 for weight-balanced 2-3-4 trees until the tree is weight-balanced.

4.3 Worst-case Tree Weight Analysis

Our online algorithms were designed with the goal of minimizing worst-case communication cost. Recall that communication cost for online algorithms is composed of two parts: the tree structure cost, and the restructuring cost to maintain a given tree structure. We would like to analyze worst-case communication cost, but this is difficult because we do not know

how to properly bound the restructuring cost for our algorithms. Instead, we settle for analyzing the worst-case tree weight for our algorithms. If restructuring costs are small, then this will be a good estimate of the communication costs. We return to this topic in Section 4.4 when we study our algorithms empirically via simulation. Throughout this section, we use the notation $g(n) \sim f(n)$ to mean $\lim_{n\to\infty} g(n)/f(n) = 1$.

We have general algorithms for B-trees and height-balanced trees (maximum degrees are settable parameters), and have both weight-balanced 2-3 and 2-3-4 algorithms. In this section, we provide our analysis assuming that our algorithms use maximum degree 4. It was shown that 4 is the best degree for random re-keying [53] (equally likely ADD(u) and DELETE(u) operations), and since we assume that we have no knowledge of the relative frequency of the re-key operations, using degree 4 seems like a reasonable choice.

4.3.1 Worst-case B-Tree Analysis

Define $w_B(n)$ to be the maximum weight of any B-tree of order 4 with n members. The analysis of the bounds on $w_B(n)$ is divided into three steps. First, we derive a relationship between tree weight w and the minimum number of leaves n to achieve that weight. We then show that $w = w_B(n)$. Using these two results we prove that $w_B(n) \sim 4\log_2 n$.

To derive a relationship between tree weight w and the minimum number of leaves n in a B-tree of order 4 to achieve that weight, we first consider the problem with an extra parameter k representing the height of the tree. Define $M(k, w)$ to be the minimum number of leaves in a B-tree of order 4 with height k and weight w. We first define a recurrence for $M(k, w)$, then derive a closed-form equation for $M(k, w)$ using the recurrence. Finally, using the equation for $M(k, w)$ we derive an equation for the minimum number of leaves in a B-tree of order 4 with weight w (independent of height k).

Theorem 7

$$M(1, w) = \begin{cases} w & \text{if } 2 \le w \le 4 \\ \infty & \text{otherwise} \end{cases} \tag{4.1}$$

$$M(k,w) = \begin{cases} \min \begin{pmatrix} M(k-1,w-2)+2^{k-1}, \\ M(k-1,w-3)+2\cdot 2^{k-1}, \\ M(k-1,w-4)+3\cdot 2^{k-1} \end{pmatrix} & \text{if } 2k \le w \le 4k \\ \infty & \text{otherwise} \end{cases} \tag{4.2}$$

Proof: The proof is by induction on k. The base case is for $k = 1$. The only possibilities for a B-tree of order 4 with height $k = 1$ is to have 2, 3, or 4 leaves corresponding to the root having degree 2, 3, or 4. Therefore, Equation (4.1) holds.

For $k > 1$, assume that Equation (4.1) or (4.2) holds for $k - 1$. For a minimum leaf B-tree of order 4 with height k, the root can have 2, 3, or 4 children, and each child should have height $k - 1$. One of these children is defined by $M(k-1, w-i)$ and is assumed to have maximum weight among its siblings. The other children should have the minimum number of leaves for a B-tree with height $k-1$, which is 2^{k-1} (a complete binary tree). This implies that the weight of each of these children is $2(k-1)$. By the induction hypothesis, $2(k-1) \le w-i$, which means that $M(k-1, w-i)$ has maximum weight among its siblings. Thus, in order to complete the proof we need to show two things: (1) for $2k \le w \le 4k$, $M(k,w)$ is finite, and (2) for $w < 2k$ or $w > 4k$, $M(k,w)$ is infinite.

Step 1: For $2k \le w \le 4k$, $M(k,w)$ is finite.

For $2k \le w \le 4k$, we have the following ranges for $w - i$ for each $M(k-1, w-i)$ appearing in the recurrence:

$2(k-1) \le w-2 \le 4(k-1)+2$ for $M(k-1, w-2)$.

$2(k-1)-1 \le w-3 \le 4(k-1)+1$ for $M(k-1, w-3)$.

$2(k-1)-2 \le w-4 \le 4(k-1)$ for $M(k-1, w-4)$.

By the induction hypothesis we know that $M(k-1, w-i)$ is finite over the range $2(k-1) \le w-i \le 4(k-1)$. We can see that for $2k \le w \le 4k$, at least one of the $M(k-1, w-i)$ are finite, thus $M(k,w)$ is finite.

Step 2: For $w < 2k$ or $w > 4k$, $M(k,w)$ is infinite.

For $w < 2k$ or $w > 4k$, we have the following ranges for $w - i$ for each $M(k-1, w-i)$ appearing in the recurrence:

$w-2 < 2(k-1)$ or $w-2 > 4(k-1)+2$ for $M(k-1, w-2)$.

$w-3 < 2(k-1)-1$ or $w-3 > 4(k-1)+1$ for $M(k-1, w-3)$.

$w-4 < 2(k-1)-2$ or $w-4 > 4(k-1)$ for $M(k-1, w-4)$.

By the induction hypothesis, we know that $M(k-1, w-i)$ is infinite if $w-i < 2(k-1)$ or $w-i > 4(k-1)$. Therefore, $M(k, w) = \infty$ if $w < 2k$ or $w > 4k$. □

Lemma 8 *Let*

$$\begin{aligned} M_2 &= 2^k + 2^{\lfloor \frac{w-2k}{2} \rfloor} + 2^{\lfloor \frac{w-2k+1}{2} \rfloor} - 2 \\ M_3 &= 2^k + 2^{k-1} + 2^{\lfloor \frac{w-2k-1}{2} \rfloor} + 2^{\lfloor \frac{w-2k}{2} \rfloor} - 2 \\ M_4 &= 2^{k+1} + 2^{\lfloor \frac{w-2k-2}{2} \rfloor} + 2^{\lfloor \frac{w-2k-1}{2} \rfloor} - 2 \end{aligned}$$

Then for $2k \leq w \leq 4k$

$$M_2 \leq M_3 \leq M_4$$

In addition, for $w = 4k-1$ *or* $w = 4k$, $M_2 = M_3$. *For* $w = 4k$, $M_3 = M_4$.

Proof: The proof is separated into two parts. We first prove the relationship between M_2 and M_3, then prove the relationship between M_3 and M_4. The proof of each relationship is separated into two cases: w even, and w odd.

For $2k \leq w \leq 4k$, $M_2 \leq M_3$. In addition, for $w = 4k-1$ or $w = 4k$, $M_2 = M_3$.

Case 1: w is even. $w \leq 4k$ implies that $\frac{w-2k-2}{2} \leq k-1$. Hence

$$\begin{aligned} M_2 &= 2^k + 2^{\lfloor \frac{w-2k}{2} \rfloor} + 2^{\lfloor \frac{w-2k+1}{2} \rfloor} - 2 \\ &= 2^k + 2^{\frac{w-2k}{2}} + 2^{\frac{w-2k}{2}} - 2 \\ &= 2^k + 2^{\frac{w-2k-2}{2}} + 2^{\frac{w-2k-2}{2}} + 2^{\frac{w-2k}{2}} - 2 \\ &= 2^k + 2^{\frac{w-2k-2}{2}} + 2^{\lfloor \frac{w-2k-1}{2} \rfloor} + 2^{\lfloor \frac{w-2k}{2} \rfloor} - 2 \\ &\leq 2^k + 2^{k-1} + 2^{\lfloor \frac{w-2k-1}{2} \rfloor} + 2^{\lfloor \frac{w-2k}{2} \rfloor} - 2 \\ &= M_3 \end{aligned}$$

Case 2: w is odd. $w \leq 4k-1$ implies that $\frac{w-2k-1}{2} \leq k-1$. Hence

$$\begin{aligned} M_2 &= 2^k + 2^{\lfloor \frac{w-2k}{2} \rfloor} + 2^{\lfloor \frac{w-2k+1}{2} \rfloor} - 2 \\ &= 2^k + 2^{\frac{w-2k-1}{2}} + 2^{\frac{w-2k+1}{2}} - 2 \\ &= 2^k + 2^{\frac{w-2k-1}{2}} + 2^{\frac{w-2k-1}{2}} + 2^{\frac{w-2k-1}{2}} - 2 \\ &= 2^k + 2^{\frac{w-2k-1}{2}} + 2^{\lfloor \frac{w-2k-1}{2} \rfloor} + 2^{\lfloor \frac{w-2k}{2} \rfloor} - 2 \\ &\leq 2^k + 2^{k-1} + 2^{\lfloor \frac{w-2k-1}{2} \rfloor} + 2^{\lfloor \frac{w-2k}{2} \rfloor} - 2 \\ &= M_3 \end{aligned}$$

In addition, by changing the starting inequalities to equalities, we see that when $w = 4k-1$ or $w = 4k$, $M_2 = M_3$.

For $2k \leq w \leq 4k$, $M_3 \leq M_4$. In addition, for $w = 4k$, $M_3 = M_4$.

Case 1: w is even. $w \leq 4k$ implies that $\frac{w-2k-2}{2} \leq k-1$. Hence

$$\begin{aligned} M_3 &= 2^k + 2^{k-1} + 2^{\lfloor \frac{w-2k-1}{2} \rfloor} + 2^{\lfloor \frac{w-2k}{2} \rfloor} - 2 \\ &= 2^k + 2^{k-1} + 2^{\frac{w-2k-2}{2}} + 2^{\frac{w-2k}{2}} - 2 \\ &= 2^k + 2^{k-1} + 2^{\frac{w-2k-2}{2}} + 2^{\frac{w-2k-2}{2}} + 2^{\frac{w-2k-2}{2}} - 2 \\ &= 2^k + 2^{k-1} + 2^{\frac{w-2k-2}{2}} + 2^{\lfloor \frac{w-2k-2}{2} \rfloor} + 2^{\lfloor \frac{w-2k-1}{2} \rfloor} - 2 \\ &\leq 2^k + 2^{k-1} + 2^{k-1} + 2^{\lfloor \frac{w-2k-2}{2} \rfloor} + 2^{\lfloor \frac{w-2k-1}{2} \rfloor} - 2 \\ &= M_4 \end{aligned}$$

Case 2: w is odd. $w < 4k+1$ implies that $\frac{w-2k-3}{2} < k-1$. Hence

$$\begin{aligned} M_3 &= 2^k + 2^{k-1} + 2^{\lfloor \frac{w-2k-1}{2} \rfloor} + 2^{\lfloor \frac{w-2k}{2} \rfloor} - 2 \\ &= 2^k + 2^{k-1} + 2^{\frac{w-2k-1}{2}} + 2^{\frac{w-2k-1}{2}} - 2 \\ &= 2^k + 2^{k-1} + 2^{\frac{w-2k-3}{2}} + 2^{\frac{w-2k-3}{2}} + 2^{\frac{w-2k-1}{2}} - 2 \\ &= 2^k + 2^{k-1} + 2^{\frac{w-2k-3}{2}} + 2^{\lfloor \frac{w-2k-2}{2} \rfloor} + 2^{\lfloor \frac{w-2k-1}{2} \rfloor} - 2 \\ &< 2^k + 2^{k-1} + 2^{k-1} + 2^{\lfloor \frac{w-2k-2}{2} \rfloor} + 2^{\lfloor \frac{w-2k-1}{2} \rfloor} - 2 \\ &= M_4 \end{aligned}$$

In addition, by changing the starting inequality $w \leq 4k$ to an equality, we see that when $w = 4k$, $M_3 = M_4$. □

Table 4.1 shows some values of $M(k, w)$ for different values of k and w. This table led to the derivation of the closed-form equation for $M(k, w)$, which is presented in the following Theorem.

Table 4.1: Values of $M(k, w)$ for B-trees of order 4. Empty entries represent the cases where $M(k, w) = \infty$.

w \ k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1															
2	2														
3	3														
4	4	4													
5		5													
6		6	8												
7		8	9												
8		10	10	16											
9			12	17											
10			14	18	32										
11			18	20	33										
12			22	22	34	64									
13				26	36	65									
14				30	38	66	128								
15				38	42	68	129								
16				46	46	70	130	256							
17					54	74	132	257							
18					62	78	134	258	512						
19					78	86	138	260	513						
20					94	94	142	262	514	1024					
21						110	150	266	516	1025					
22						126	158	270	518	1026	2048				
23						158	174	278	522	1028	2049				
24						190	190	286	526	1030	2050	4096			
25							222	302	534	1034	2052	4097			

Theorem 9

$$M(k,w) = \begin{cases} 2^k + 2^{\lfloor \frac{w-2k}{2} \rfloor} + 2^{\lfloor \frac{w-2k+1}{2} \rfloor} - 2 & \text{if } 2k \leq w \leq 4k \\ \infty & \text{otherwise} \end{cases} \tag{4.3}$$

Proof: The proof is by induction on k. The base case is for $k = 1$. The only possibilities for a B-tree of order 4 with height $k = 1$ is to have 2, 3, or 4 leaves corresponding to the root having degree 2, 3, or 4. It is straightforward to verify that the equation holds for the base case.

For $k > 1$, assume that Equation (4.3) holds for $k-1$. For $w < 2k$ or $w > 4k$, we showed in the proof of Theorem 7 that $M(k,w) = \infty$. We just need to show that Equation (4.3) is true for $2k \leq w \leq 4k$. By the induction hypothesis, we have the following equation for $M(k-1, w-i)$:

$$M(k-1,w-i) = \begin{cases} 2^{k-1} + 2^{\lfloor \frac{w-2k+2-i}{2} \rfloor} + 2^{\lfloor \frac{w-2k+3-i}{2} \rfloor} - 2 & \text{if } 2(k-1) \leq w-i \leq 4(k-1) \\ \infty & \text{otherwise} \end{cases} \tag{4.4}$$

Using Lemma 8 and Equation (4.4), we derive an equation for each possibility in the recurrence for $M(k,w)$ given in Theorem 7.

$$M(k-1,w-2) + 2^{k-1} = \begin{cases} M_2 & \text{if } 2(k-1) \leq w-2 \leq 4(k-1) \\ \infty & \text{otherwise} \end{cases}$$

$$M(k-1,w-3) + 2 \cdot 2^{k-1} = \begin{cases} M_3 & \text{if } 2(k-1) \leq w-3 \leq 4(k-1) \\ \infty & \text{otherwise} \end{cases}$$

$$M(k-1,w-4) + 3 \cdot 2^{k-1} = \begin{cases} M_4 & \text{if } 2(k-1) \leq w-4 \leq 4(k-1) \\ \infty & \text{otherwise} \end{cases}$$

Note that $M(k-1,w-2)$ is finite for $2(k-1) \leq w-2 \leq 4(k-1)$, and thus is finite for $2k \leq w \leq 4k-2$. Hence by Lemma 8, $M(k,w) = M_2$ for $2k \leq w \leq 4k-2$. For $w = 4k-1$, both $M(k-1,w-3)$ and $M(k-1,w-4)$ are finite. Hence by Lemma 8, $M(k,w) = M_3 = M_2$ for $w = 4k-1$. Finally, for $w = 4k$, only $M(k-1,w-4)$ is finite. Hence by Lemma 8, $M(k,w) = M_4 = M_2$ for $w = 4k$. Therefore, $M(k,w) = M_2$ for $2k \leq w \leq 4k$. □

Lemma 10 *For fixed w and $\frac{w}{4} \leq k \leq \frac{w}{2}$, the function $M(k,w)$ is non-decreasing.*

Proof: $\frac{w}{4} \leq k$ and $k+1 \leq \frac{w}{2}$ implies that both $2k \leq w \leq 4k$ and $2(k+1) \leq w \leq 4(k+1)$. By Theorem 9, we know that $M(k,w)$ and $M(k+1,w)$ are finite over these ranges. Therefore

$$\begin{aligned} M(k,w) &= 2^k + 2^{\lfloor \frac{w-2k}{2} \rfloor} + 2^{\lfloor \frac{w-2k+1}{2} \rfloor} - 2 \\ M(k+1,w) &= 2^{k+1} + 2^{\lfloor \frac{w-2(k+1)}{2} \rfloor} + 2^{\lfloor \frac{w-2(k+1)+1}{2} \rfloor} - 2 \end{aligned}$$

We compare $M(k,w)$ to $M(k+1,w)$ for $\frac{w}{4} \leq k$, $k+1 \leq \frac{w}{2}$. The comparison is split into two cases: w even, and w odd.

Case 1: w is even. $w \leq 4k$ implies that $\frac{w-2k}{2} \leq k$. Hence

$$\begin{aligned} M(k,w) &= 2^k + 2^{\lfloor \frac{w-2k}{2} \rfloor} + 2^{\lfloor \frac{w-2k+1}{2} \rfloor} - 2 \\ &= 2^k + 2^{\frac{w-2k}{2}} + 2^{\frac{w-2k}{2}} - 2 \\ &= 2^k + 2^{\frac{w-2k}{2}} + 2^{\frac{w-2k-2}{2}} + 2^{\frac{w-2k-2}{2}} - 2 \\ &= 2^k + 2^{\frac{w-2k}{2}} + 2^{\lfloor \frac{w-2(k+1)}{2} \rfloor} + 2^{\lfloor \frac{w-2(k+1)+1}{2} \rfloor} - 2 \\ &\leq 2^k + 2^k + 2^{\lfloor \frac{w-2(k+1)}{2} \rfloor} + 2^{\lfloor \frac{w-2(k+1)+1}{2} \rfloor} - 2 \\ &= M(k+1,w) \end{aligned}$$

Case 2: w is odd. $w \leq 4k-1$ implies that $\frac{w-2k+1}{2} \leq k$. Hence

$$\begin{aligned} M(k,w) &= 2^k + 2^{\lfloor \frac{w-2k}{2} \rfloor} + 2^{\lfloor \frac{w-2k+1}{2} \rfloor} - 2 \\ &= 2^k + 2^{\frac{w-2k-1}{2}} + 2^{\frac{w-2k+1}{2}} - 2 \\ &= 2^k + 2^{\frac{w-2k+1}{2}} + 2^{\frac{w-2k-3}{2}} + 2^{\frac{w-2k-3}{2}} - 2 \\ &< 2^k + 2^{\frac{w-2k+1}{2}} + 2^{\lfloor \frac{w-2k-2}{2} \rfloor} + 2^{\frac{w-2k-1}{2}} - 2 \\ &= 2^k + 2^{\frac{w-2k+1}{2}} + 2^{\lfloor \frac{w-2(k+1)}{2} \rfloor} + 2^{\lfloor \frac{w-2(k+1)+1}{2} \rfloor} - 2 \\ &\leq 2^k + 2^k + 2^{\lfloor \frac{w-2(k+1)}{2} \rfloor} + 2^{\lfloor \frac{w-2(k+1)+1}{2} \rfloor} - 2 \\ &= M(k+1,w) \end{aligned}$$

□

Theorem 11 *Define $N(w)$ to be the minimum number of leaves in a B-tree of order 4 with weight w. For $w \geq 2$*

1. $w = 4k - 3 \Rightarrow N(w) = 2^k + 2^{k-1} + 2^{k-2} - 2.$

2. $w = 4k - 2 \Rightarrow N(w) = 2^{k+1} - 2.$

3. $w = 4k - 1 \Rightarrow N(w) = 2^{k+1} + 2^{k-1} - 2.$

4. $w = 4k \Rightarrow N(w) = 2^{k+1} + 2^k - 2.$

Proof: For $w \geq 2$, the form of the implications in the statement of this Theorem implies that $2k \leq w \leq 4k$. Thus by Theorem 9, $M(k, w)$ is finite. To find the minimum number of leaves $N(w)$ in a B-tree of order 4 with weight w, we know by Lemma 10 that we should choose the height of the tree to be the smallest k such that $M(k, w)$ is finite. In other words, $k = \lceil \frac{w}{4} \rceil$. We use this value of k along with the equation for $M(k, w)$ given by Theorem 9 to prove each of the implications.

Implication 1: $w = 4k - 3 \Rightarrow N(w) = 2^k + 2^{k-1} + 2^{k-2} - 2.$

$$\begin{aligned} M(k,w) &= 2^k + 2^{\lfloor \frac{2k-3}{2} \rfloor} + 2^{\lfloor \frac{2k-2}{2} \rfloor} - 2 \\ &= 2^k + 2^{\lfloor k - \frac{3}{2} \rfloor} + 2^{\lfloor k-1 \rfloor} - 2 \\ &= 2^k + 2^{k-2} + 2^{k-1} - 2 \end{aligned}$$

Implication 2: $w = 4k - 2 \Rightarrow N(w) = 2^{k+1} - 2.$

$$\begin{aligned} M(k,w) &= 2^k + 2^{\lfloor \frac{2k-2}{2} \rfloor} + 2^{\lfloor \frac{2k-1}{2} \rfloor} - 2 \\ &= 2^k + 2^{\lfloor k-1 \rfloor} + 2^{\lfloor k - \frac{1}{2} \rfloor} - 2 \\ &= 2^k + 2^{k-1} + 2^{k-1} - 2 \\ &= 2^{k+1} - 2 \end{aligned}$$

Implication 3: $w = 4k - 1 \Rightarrow N(w) = 2^{k+1} + 2^{k-1} - 2.$

$$\begin{aligned} M(k,w) &= 2^k + 2^{\lfloor \frac{2k-1}{2} \rfloor} + 2^{\lfloor \frac{2k}{2} \rfloor} - 2 \\ &= 2^k + 2^{\lfloor k - \frac{1}{2} \rfloor} + 2^{\lfloor k \rfloor} - 2 \\ &= 2^k + 2^{k-1} + 2^k - 2 \\ &= 2^{k+1} + 2^{k-1} - 2 \end{aligned}$$

Implication 4: $w = 4k \Rightarrow N(w) = 2^{k+1} + 2^k - 2$.

$$\begin{aligned} M(k,w) &= 2^k + 2^{\lfloor \frac{2k}{2} \rfloor} + 2^{\lfloor \frac{2k+1}{2} \rfloor} - 2 \\ &= 2^k + 2^{\lfloor k \rfloor} + 2^{\lfloor k + \frac{1}{2} \rfloor} - 2 \\ &= 2^k + 2^k + 2^k - 2 \\ &= 2^{k+1} + 2^k - 2 \end{aligned}$$

□

Theorem 11 gives us a relationship between tree weight w and the minimum number of leaves $N(w)$ to achieve that weight. Next, we show that that $w = w_B(n)$, and that $w_B(n)$ is non-decreasing. This allows us to use Theorem 11 to derive a relationship between n and $w_B(n)$.

Lemma 12 *For all $N(w) \leq n < N(w+1)$, there exists a B-tree of order 4 with n leaves and weight $w' \geq w$.*

Proof: Let T be a B-tree of order 4 with $N(w)$ leaves and weight w. Define u to be a leaf in T with ancestor weight equal to w. In other words, u has the largest ancestor weight in the tree. Define p to be the parent of u. There are two different scenarios to consider when adding a new leaf into a B-tree of order 4 as a child of p: (1) p has degree less than 4, (2) p has degree 4. Consider the algorithm for inserting a new member into a B-tree given in Section 4.2.1. For the first case, it is easy to see that the ancestor weight of u cannot decrease. For the second case, a new node p' is created to be the parent of the new node and one child of p, and the algorithm is recursively applied to add p'. Assuming that u remains as a child of p, we can see that the ancestor weight or u has just decreased by 1. However, as a result of creating p', at least one additional branch must be added higher up in the tree, so the ancestor weight of u does not decrease. This same argument can be applied at every level of the recursion, and thus the ancestor weight of u either remains the same or increases as a result of adding the new leaf. Because u was assumed to have the largest ancestor weight in the tree, adding a new member to a B-tree in the manner just described does not decrease the tree weight, and this fact proves the lemma. □

Theorem 13 *The relationship between the number of leaves $N(w)$ and the worst tree weight $w_B(n)$ in a B-tree of order 4 is given by*

$$\begin{aligned}
2^k + 2^{k-1} + 2^{k-2} - 2 \;\leq\; n \;<\; 2^{k+1} - 2, &\qquad w_B(n) = 4k-3 \\
2^{k+1} - 2 \;\leq\; n \;<\; 2^{k+1} + 2^{k-1} - 2, &\qquad w_B(n) = 4k-2 \\
2^{k+1} + 2^{k-1} - 2 \;\leq\; n \;<\; 2^{k+1} + 2^{k} - 2, &\qquad w_B(n) = 4k-1 \\
2^{k+1} + 2^{k} - 2 \;\leq\; n \;<\; 2^{k+1} + 2^{k} + 2^{k-1} - 2, &\qquad w_B(n) = 4k
\end{aligned}$$

Proof: We know that $w_B(n)$ is the maximum tree weight for any B-tree of order 4 with n leaves. Given w, by Theorem 11 $N(w)$ is the fewest number of leaves in a B-tree of order 4 with weight w. Also, by Theorem 11, $N(w+1) > N(w)$ is the fewest number of leaves in a B-tree of order 4 with weight $w+1$. We show that $w_B(n) = w$ for $N(w) \leq n < N(w+1)$, which proves the Theorem. There are two inequalities to show.

(i) $w_B(n) \geq w$. This is true by Lemma 12.

(ii) $w_B(n) \leq w$. For the purpose of contradiction, suppose that $w' = w_B(n) > w$. This means that $w' \geq w+1$, and the smallest n such that there is a B-tree of order 4 with weight $w' = w_B(n)$ is $n \geq N(w+1)$. But since we are considering the range $N(w) \leq n < N(w+1)$, we have a contradiction. □

Using the relationship between n and $w_B(n)$ given by Theorem 13, we now prove that $w_B(n) \sim 4\log_2 n$.

Theorem 14 $w_B(n) \sim 4\log_2 n$.

Proof: We show that $4\log_2 n - c_1 \leq w_B(n) \leq 4\log_2 n - c_2$, where c_1 and c_2 are constants. This implies that $\lim_{n\to\infty} w_B(n)/4\log_2 n = 1$. We show that the bounds hold for each of the ranges given in Theorem 13, which covers all values of $w_B(n) \geq 2$.

1. $2^k + 2^{k-1} + 2^{k-2} - 2 \leq n < 2^{k+1} - 2$, $w_B(n) = 4k-3$.

For all $k \geq 1$, choosing $c_1 \geq 4\log_2(2 - 3/2^k) + 3$ allows us to prove the following.

$$\begin{aligned} 4k - 3 &= w_B(n) \\ 4k + 4\log_2(2 - 3/2^k) - c_1 &\leq \\ 4\log_2 2^k + 4\log_2(2 - 3/2^k) - c_1 &= \\ 4\log_2(2^k(2 - 3/2^k)) - c_1 &= \\ 4\log_2(2^{k+1} - 3) - c_1 &= \\ 4\log_2 n - c_1 &\leq \end{aligned}$$

For all $k \geq 1$, choosing $c_2 \leq 4\log_2(1 + 1/2 + 1/2^2 - 2/2^k) + 3$ allows us to prove the following.

$$\begin{aligned} w_B(n) &= 4k - 3 \\ &\leq 4k + 4\log_2(1 + 1/2 + 1/2^2 - 2/2^k) - c_2 \\ &= 4\log_2 2^k + 4\log_2(1 + 1/2 + 1/2^2 - 2/2^k) - c_2 \\ &= 4\log_2(2^k(1 + 1/2 + 1/2^2 - 2/2^k)) - c_2 \\ &= 4\log_2(2^k + 2^{k-1} + 2^{k-2} - 2) - c_2 \\ &\leq 4\log_2 n - c_2 \end{aligned}$$

2. $2^{k+1} - 2 \leq n < 2^{k+1} + 2^{k-1} - 2$, $w_B(n) = 4k - 2$.

For all $k \geq 1$, choosing $c_1 \geq 4\log_2(2 + 1/2 - 3/2^k) + 2$ allows us to prove the following.

$$\begin{aligned} 4k - 2 &= w_B(n) \\ 4k + 4\log_2(2 + 1/2 - 3/2^k) - c_1 &\leq \\ 4\log_2 2^k + 4\log_2(2 + 1/2 - 3/2^k) - c_1 &= \\ 4\log_2(2^k(2 + 1/2 - 3/2^k)) - c_1 &= \\ 4\log_2(2^{k+1} + 2^{k-1} - 3) - c_1 &= \\ 4\log_2 n - c_1 &\leq \end{aligned}$$

For all $k \geq 1$, choosing $c_2 \leq 4\log_2(2 - 2/2^k) + 2$ allows us to prove the following.

$$\begin{aligned} w_B(n) &= 4k - 2 \\ &\leq 4k + 4\log_2(2 - 2/2^k) - c_2 \\ &= 4\log_2 2^k + 4\log_2(2 - 2/2^k) - c_2 \\ &= 4\log_2(2^k(2 - 2/2^k)) - c_2 \\ &= 4\log_2(2^{k+1} - 2) - c_2 \\ &\leq 4\log_2 n - c_2 \end{aligned}$$

3. $2^{k+1} + 2^{k-1} - 2 \leq n < 2^{k+1} + 2^k - 2$, $w_B(n) = 4k - 1$.

For all $k \geq 1$, choosing $c_1 \geq 4\log_2(3 - 3/2^k) + 1$ allows us to prove the following.

$$\begin{aligned} 4k - 1 &= w_B(n) \\ 4k + 4\log_2(3 - 3/2^k) - c_1 &\leq \\ 4\log_2 2^k + 4\log_2(3 - 3/2^k) - c_1 &= \\ 4\log_2(2^k(2 + 1 - 3/2^k)) - c_1 &= \\ 4\log_2(2^{k+1} + 2^k - 3) - c_1 &\leq \\ 4\log_2 n - c_1 &= \end{aligned}$$

For all $k \geq 1$, choosing $c_2 \leq 4\log_2(2 + 1/2 - 2/2^k) + 1$ allows us to prove the following.

$$\begin{aligned} w_B(n) &= 4k - 1 \\ &\leq 4k + 4\log_2(2 + 1/2 - 2/2^k) - c_2 \\ &= 4\log_2 2^k + 4\log_2(2 + 1/2 - 2/2^k) - c_2 \\ &= 4\log_2(2^k(2 + 1/2 - 2/2^k)) - c_2 \\ &= 4\log_2(2^{k+1} + 2^{k-1} - 2) - c_2 \\ &\leq 4\log_2 n - c_2 \end{aligned}$$

4. $2^{k+1} + 2^k - 2 \leq n < 2^{k+1} + 2^k + 2^{k-1} - 2$, $w_B(n) = 4k$.

For all $k \geq 1$, choosing $c_1 \geq 4\log_2(3 + 1/2 - 3/2^k)$ allows us to prove the following.

$$\begin{aligned} 4k &= w_B(n) \\ 4k + 4\log_2(3 + 1/2 - 3/2^k) - c_1 &\leq \\ 4\log_2 2^k + 4\log_2(3 + 1/2 - 3/2^k) - c_1 &= \\ 4\log_2(2^k(2 + 1 + 1/2 - 3/2^k)) - c_1 &= \\ 4\log_2(2^{k+1} + 2^k + 2^{k-1} - 3) - c_1 &= \\ 4\log_2 n - c_1 &\leq \end{aligned}$$

For all $k \geq 1$, choosing $c_2 \leq 4\log_2(3 - 2/2^k)$ allows us to prove the following.

$$\begin{aligned} w_B(n) &= 4k \\ &\leq 4k + 4\log_2(3 - 2/2^k) - c_2 \\ &= 4\log_2 2^k + 4\log_2(3 - 2/2^k) - c_2 \\ &= 4\log_2(2^k(2 + 1 - 2/2^k)) - c_2 \\ &= 4\log_2(2^{k+1} + 2^k - 2) - c_2 \\ &\leq 4\log_2 n - c_2 \end{aligned}$$

□

4.3.2 Worst-case Height-Balanced 2-k Tree Analysis

Define $w_H(n)$ to be the maximum weight of any height-balanced 2-4 tree with n members. The analysis of the bounds on $w_H(n)$ is divided into three steps. First, we derive a relationship between tree weight w and the minimum number of leaves n to achieve that weight. We then show that $w = w_H(n)$. Using these two results we prove that $w_H(n) \sim 4\log_\phi n$, where $\phi = (1 + \sqrt{5})/2 \approx 1.61803$ (ϕ is the golden ratio).

To derive a relationship between tree weight w and the minimum number of leaves n in a height-balanced 2-4 tree to achieve that weight, we first consider the problem with an extra parameter k representing the height of the tree. For $k \geq 1$, define $M(k, w)$ to be the minimum number of leaves in a height-balanced 2-4 tree with height k and weight w. For $k \leq 0$, define $M(k, w) = \infty$ because a tree with negative height makes no sense, and a tree with height 0 is a single node (which has no weight). We first define a recurrence

for $M(k,w)$, then derive a closed-form equation for $M(k,w)$ using the recurrence. Finally, using the equation for $M(k,w)$ we derive an equation for the minimum number of leaves in a height-balanced 2-4 tree with weight w (independent of height k).

Before moving on, we describe how to construct minimum leaf height-balanced 2-4 trees of a given height h.

Lemma 15 *Define $L(h)$ to be the minimum number of leaves in a height-balanced tree of height h. Then $L(h) = F_{h+2}$, where F_{h+2} is the $h+2$nd Fibonacci number.*

Proof: The root of the tree must be height-balanced, and we should use the minimum number of leaves in each sub-tree. This implies that $L(h) = L(h-1) + L(h-2)$, and that each internal node has degree 2. It can also be verified by hand constructing the trees that $L(1) = 2$ and $L(2) = 3$. This equation for $L(h)$ is the same as the Fibonacci sequence [18] with different initial values. In particular, $L(h) = F_{h+2}$. □

Theorem 16

$$M(1,w) = \begin{cases} w & \text{if } 2 \le w \le 4 \\ \infty & \text{otherwise} \end{cases} \tag{4.5}$$

$$M(k,w) = \begin{cases} \min \begin{pmatrix} M_{k-1,w-2}, \\ M_{k-1,w-3}, \\ M_{k-1,w-4}, \\ M_{k-2,w-2}, \\ M_{k-2,w-3}, \\ M_{k-2,w-4} \end{pmatrix} & \text{if } 2k \le w \le 4k \\ \infty & \text{otherwise} \end{cases} \tag{4.6}$$

where

$$M_{k-1,w-2} = \begin{cases} M(k-1,w-2) + F_k & \text{if } 2(k-2) \le w-2 \\ \infty & \text{otherwise} \end{cases}$$

$$M_{k-1,w-3} = \begin{cases} M(k-1,w-3) + 2F_k & \text{if } 2(k-2) \le w-3 \\ \infty & \text{otherwise} \end{cases}$$

$$M_{k-1,w-4} = \begin{cases} M(k-1,w-4)+3F_k & \text{if } 2(k-2) \leq w-4 \\ \infty & \text{otherwise} \end{cases}$$

$$M_{k-2,w-2} = \begin{cases} M(k-2,w-2)+F_{k+1} & \text{if } 2(k-1) \leq w-2 \\ \infty & \text{otherwise} \end{cases}$$

$$M_{k-2,w-3} = \begin{cases} M(k-2,w-3)+F_{k+1}+F_k & \text{if } 2(k-1) \leq w-3 \\ \infty & \text{otherwise} \end{cases}$$

$$M_{k-2,w-4} = \begin{cases} M(k-2,w-4)+F_{k+1}+2F_k & \text{if } 2(k-1) \leq w-4 \\ \infty & \text{otherwise} \end{cases}$$

Proof: The proof is by induction on k. The base case is for $k = 1$. The only possibilities for a height-balanced 2-4 tree is to have 2, 3, or 4 leaves corresponding to the root having degree 2, 3, or 4. Therefore, Equation (4.5) holds.

For $k > 1$, assume that Equation (4.5) or (4.6) holds for $k-1$. For a minimum leaf height-balanced 2-4 tree with height k, the root can have 2, 3, or 4 children. One of these children should have height $k-1$, and the others should have height $k-2$. By choosing the child $M(k-j,w-i)$ we are imposing constraints on the height and weight of the other children. For $M(k-1,w-i)$, the other children have height $k-2$. For $M(k-2,w-i)$, the height of one other child is $k-1$ and the height of the remaining children is $k-2$. Also note that by choosing $M(k-j,w-i)$, the weight of the other children are constrained to be less than or equal to $w-i$, otherwise the weight of the tree $M(k,w)$ will be greater than w. Since we know by Lemma 15 that the minimum number of leaves in a height-balanced 2-4 tree with height h is F_{h+2}, and is a tree where all internal nodes have degree 2, we see that the definition of the $M_{k-j,w-i}$ checks this constraint, assigning $M_{k-j,w-i} = \infty$ if it is not met. In order to complete the proof we need to show two things: (1) for $2k \leq w \leq 4k$, $M(k,w)$ is finite, and (2) for $w < 2k$ or $w > 4k$, $M(k,w)$ is infinite.

Step 1: For $2k \leq w \leq 4k$, $M(k,w)$ is finite.

For $2k \leq w \leq 4k$, we have the following ranges for $w-i$ for each $M_{k-1,w-i}$ appearing in the recurrence:

$2(k-1) \leq w-2 \leq 4(k-1)+2$ for $M(k-1, w-2)$.

$2(k-1)-1 \leq w-3 \leq 4(k-1)+1$ for $M(k-1, w-3)$.

$2(k-1)-2 \leq w-4 \leq 4(k-1)$ for $M(k-1, w-4)$.

By the induction hypothesis and the definition of $M_{k-1,w-i}$ we know that $M_{k-1,w-i}$ is finite over the range $2(k-1) \leq w-i \leq 4(k-1)$. We can see that for $2k \leq w \leq 4k$, at least one of the $M_{k-1,w-i}$ are finite, thus $M(k,w)$ is finite.

Step 2: For $w < 2k$ or $w > 4k$, $M(k,w)$ is infinite.

For $w < 2k$ or $w > 4k$, we have the following ranges for $w-i$ for each $M_{k-j,w-i}$ appearing in the recurrence:

$w-2 < 2(k-1)$ or $w-2 > 4(k-1)+2$ for $M(k-1, w-2)$.

$w-3 < 2(k-1)-1$ or $w-3 > 4(k-1)+1$ for $M(k-1, w-3)$.

$w-4 < 2(k-1)-2$ or $w-4 > 4(k-1)$ for $M(k-1, w-4)$.

$w-2 < 2(k-2)+2$ or $w-2 > 4(k-2)+6$ for $M(k-2, w-2)$.

$w-3 < 2(k-2)+1$ or $w-3 > 4(k-2)+5$ for $M(k-2, w-3)$.

$w-4 < 2(k-2)$ or $w-4 > 4(k-2)+4$ for $M(k-2, w-4)$.

By the induction hypothesis, we know that $M_{k-1,w-i}$ is infinite if $w-i < 2(k-1)$ or $w-i > 4(k-1)$. Also by the induction hypothesis, $M_{k-2,w-i}$ is infinite if $w-i < 2(k-2)$ or $w-i > 4(k-2)$. By the definition of the $M_{k-j,w-i}$ we also know that the $M_{k-2,w-i}$ are infinite when $2(k-1) > w-i$. Therefore, $M(k,w) = \infty$ if $w < 2k$ or $w > 4k$. □

Lemma 17 *Let*

$$
\begin{aligned}
M_{12} &= F_{k+2} + F_{\lfloor \frac{w-2k+4}{2} \rfloor} + F_{\lfloor \frac{w-2k+5}{2} \rfloor} - 2 \\
M_{13} &= F_{k+2} + F_k + F_{\lfloor \frac{w-2k+3}{2} \rfloor} + F_{\lfloor \frac{w-2k+4}{2} \rfloor} - 2 \\
M_{14} &= F_{k+2} + 2F_k + F_{\lfloor \frac{w-2k+2}{2} \rfloor} + F_{\lfloor \frac{w-2k+3}{2} \rfloor} - 2 \\
M_{22} &= F_{k+2} + F_{\lfloor \frac{w-2k+6}{2} \rfloor} + F_{\lfloor \frac{w-2k+7}{2} \rfloor} - 2 \\
M_{23} &= F_{k+2} + F_k + F_{\lfloor \frac{w-2k+5}{2} \rfloor} + F_{\lfloor \frac{w-2k+6}{2} \rfloor} - 2 \\
M_{24} &= F_{k+2} + 2F_k + F_{\lfloor \frac{w-2k+4}{2} \rfloor} + F_{\lfloor \frac{w-2k+5}{2} \rfloor} - 2
\end{aligned}
$$

Then for $2k \leq w \leq 4k$

$$M_{12} \leq M_{13} \leq M_{14}$$

$$\begin{aligned} M_{12} &< M_{22} \\ M_{12} &< M_{23} \\ M_{12} &< M_{24} \end{aligned}$$

In addition, for $w = 4k-1$ *or* $w = 4k$, $M_{12} = M_{13}$. *For* $w = 4k$, $M_{13} = M_{14}$.

Proof: The proof of each inequality is separated into two cases: w even, and w odd.

For $2k \leq w \leq 4k$, $M_{12} \leq M_{13}$. In addition, for $w = 4k-1$ or $w = 4k$, $M_{12} = M_{13}$.

Case 1: w is even. $w \leq 4k$ implies that $\frac{w-2k}{2} \leq k$. Hence

$$\begin{aligned} M_{12} &= F_{k+2} + F_{\lfloor \frac{w-2k+4}{2} \rfloor} + F_{\lfloor \frac{w-2k+5}{2} \rfloor} - 2 \\ &= F_{k+2} + F_{\frac{w-2k+4}{2}} + F_{\frac{w-2k+4}{2}} - 2 \\ &= F_{k+2} + F_{\frac{w-2k}{2}} + F_{\frac{w-2k+2}{2}} + F_{\frac{w-2k+4}{2}} - 2 \\ &= F_{k+2} + F_{\frac{w-2k}{2}} + F_{\lfloor \frac{w-2k+3}{2} \rfloor} + F_{\lfloor \frac{w-2k+4}{2} \rfloor} - 2 \\ &\leq F_{k+2} + F_k + F_{\lfloor \frac{w-2k+3}{2} \rfloor} + F_{\lfloor \frac{w-2k+4}{2} \rfloor} - 2 \\ &= M_{13} \end{aligned}$$

Case 2: w is odd. $w \leq 4k-1$ implies that $\frac{w-2k+1}{2} \leq k$. Hence

$$\begin{aligned} M_{12} &= F_{k+2} + F_{\lfloor \frac{w-2k+4}{2} \rfloor} + F_{\lfloor \frac{w-2k+5}{2} \rfloor} - 2 \\ &= F_{k+2} + F_{\frac{w-2k+3}{2}} + F_{\frac{w-2k+5}{2}} - 2 \\ &= F_{k+2} + F_{\frac{w-2k+1}{2}} + F_{\frac{w-2k+3}{2}} + F_{\frac{w-2k+3}{2}} - 2 \\ &= F_{k+2} + F_{\frac{w-2k+1}{2}} + F_{\lfloor \frac{w-2k+3}{2} \rfloor} + F_{\lfloor \frac{w-2k+4}{2} \rfloor} - 2 \\ &\leq F_{k+2} + F_k + F_{\lfloor \frac{w-2k+3}{2} \rfloor} + F_{\lfloor \frac{w-2k+4}{2} \rfloor} - 2 \\ &= M_{13} \end{aligned}$$

In addition, by changing the starting inequalities to equalities, we see that when $w = 4k-1$ or $w = 4k$, $M_{12} = M_{13}$.

For $2k \leq w \leq 4k$, $M_{13} \leq M_{14}$. In addition, for $w = 4k$, $M_{13} = M_{14}$.

Case 1: w is even. $w \leq 4k$ implies that $\frac{w-2k}{2} \leq k$. Hence

$$\begin{aligned} M_{13} &= F_{k+2} + F_k + F_{\lfloor \frac{w-2k+3}{2} \rfloor} + F_{\lfloor \frac{w-2k+4}{2} \rfloor} - 2 \\ &= F_{k+2} + F_k + F_{\frac{w-2k+2}{2}} + F_{\frac{w-2k+4}{2}} - 2 \\ &= F_{k+2} + F_k + F_{\frac{w-2k}{2}} + F_{\frac{w-2k+2}{2}} + F_{\frac{w-2k+2}{2}} - 2 \\ &= F_{k+2} + F_k + F_{\frac{w-2k}{2}} + F_{\lfloor \frac{w-2k+2}{2} \rfloor} + F_{\lfloor \frac{w-2k+3}{2} \rfloor} - 2 \\ &\leq F_{k+2} + 2F_k + F_{\lfloor \frac{w-2k+2}{2} \rfloor} + F_{\lfloor \frac{w-2k+3}{2} \rfloor} - 2 \\ &= M_{14} \end{aligned}$$

Case 2: w is odd. $w < 4k + 1$ implies that $\frac{w-2k-1}{2} < k$. Hence

$$\begin{aligned} M_{13} &= F_{k+2} + F_k + F_{\lfloor \frac{w-2k+3}{2} \rfloor} + F_{\lfloor \frac{w-2k+4}{2} \rfloor} - 2 \\ &= F_{k+2} + F_k + F_{\frac{w-2k+3}{2}} + F_{\frac{w-2k+3}{2}} - 2 \\ &= F_{k+2} + F_k + F_{\frac{w-2k-1}{2}} + F_{\frac{w-2k+1}{2}} + F_{\frac{w-2k+3}{2}} - 2 \\ &= F_{k+2} + F_k + F_{\frac{w-2k-1}{2}} + F_{\lfloor \frac{w-2k+2}{2} \rfloor} + F_{\lfloor \frac{w-2k+3}{2} \rfloor} - 2 \\ &< F_{k+2} + 2F_k + F_{\lfloor \frac{w-2k+2}{2} \rfloor} + F_{\lfloor \frac{w-2k+3}{2} \rfloor} - 2 \\ &= M_{14} \end{aligned}$$

In addition, by changing the starting inequality $w \leq 4k$ to an equality, we see that when $w = 4k$, $M_{13} = M_{14}$.

For $2k \leq w \leq 4k$, $M_{12} < M_{22}$.

$$\begin{aligned} M_{12} &= F_{k+2} + F_{\lfloor \frac{w-2k+4}{2} \rfloor} + F_{\lfloor \frac{w-2k+5}{2} \rfloor} - 2 \\ &< F_{k+2} + F_{\lfloor \frac{w-2k+6}{2} \rfloor} + F_{\lfloor \frac{w-2k+7}{2} \rfloor} - 2 \\ &= M_{22} \end{aligned}$$

For $2k \leq w \leq 4k$, $M_{12} < M_{23}$.

$$\begin{aligned} M_{12} &= F_{k+2} + F_{\lfloor \frac{w-2k+4}{2} \rfloor} + F_{\lfloor \frac{w-2k+5}{2} \rfloor} - 2 \\ &< F_{k+2} + F_k + F_{\lfloor \frac{w-2k+5}{2} \rfloor} + F_{\lfloor \frac{w-2k+6}{2} \rfloor} - 2 \\ &= M_{23} \end{aligned}$$

For $2k \leq w \leq 4k$, $M_{12} < M_{24}$.

$$\begin{aligned} M_{12} &= F_{k+2} + F_{\lfloor \frac{w-2k+4}{2} \rfloor} + F_{\lfloor \frac{w-2k+5}{2} \rfloor} - 2 \\ &< F_{k+2} + 2F_k + F_{\lfloor \frac{w-2k+4}{2} \rfloor} + F_{\lfloor \frac{w-2k+5}{2} \rfloor} - 2 \\ &= M_{23} \end{aligned}$$

□

Table 4.2 shows some values of $M(k, w)$ for different values of k and w. This table led to the derivation of the closed-form equation for $M(k, w)$, which is presented in the following Theorem.

Theorem 18

$$M(k,w) = \begin{cases} F_{k+2} + F_{\lfloor \frac{w-2k+4}{2} \rfloor} + F_{\lfloor \frac{w-2k+5}{2} \rfloor} - 2 & \text{if } 2k \leq w \leq 4k \\ \infty & \text{otherwise} \end{cases} \tag{4.7}$$

Proof: The proof is by induction on k. The base case is for $k = 1$. The only possibilities for a height-balanced 2-4 tree with height $k = 1$ is to have 2, 3, or 4 leaves corresponding to the root having degree 2, 3, or 4. It is straightforward to verify that the equation holds for the base case.

For $k > 1$, assume that Equation (4.7) holds for $k-1$. For $w < 2k$ or $w > 4k$, we showed in the proof of Theorem 16 that $M(k, w) = \infty$. We just need to show that Equation (4.7) is true for $2k \leq w \leq 4k$. By the induction hypothesis, we have the following equation for $M(k-1, w-i)$:

$$M(k-1,w-i) = \begin{cases} F_{k+1} + F_{\lfloor \frac{w-2k+6-i}{2} \rfloor} + F_{\lfloor \frac{w-2k+7-i}{2} \rfloor} - 2 & \text{if } 2(k-1) \leq w-i \leq 4(k-1) \\ \infty & \text{otherwise} \end{cases} \tag{4.8}$$

Using Lemma 17 and Equation (4.8), we derive an equation for each possibility in the recurrence for $M(k, w)$ given in Theorem 16.

$$M_{k-1,w-2} = \begin{cases} M_{12} & \text{if } 2(k-1) \leq w-2 \leq 4(k-1) \\ \infty & \text{otherwise} \end{cases}$$

Table 4.2: Values of $M(k, w)$ for height-balanced 2-4 trees. Empty entries represent the cases where $M(k, w) = \infty$.

w \ k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1															
2	2														
3	3														
4	4	3													
5		4													
6		5	5												
7		6	6												
8		7	7	8											
9			8	9											
10			9	10	13										
11			11	11	14										
12			13	12	15	21									
13				14	16	22									
14				16	17	23	34								
15				19	19	24	35								
16				22	21	25	36	55							
17					24	27	37	56							
18					27	29	38	57	89						
19					32	32	40	58	90						
20					37	35	42	59	91	144					
21						40	45	61	92	145					
22						45	48	63	93	146	233				
23						53	53	66	95	147	234				
24						61	58	69	97	148	235	377			
25							66	74	100	150	236	378			

$$M_{k-1,w-3} = \begin{cases} M_{13} & \text{if } 2(k-1) \leq w-3 \leq 4(k-1) \\ \infty & \text{otherwise} \end{cases}$$

$$M_{k-1,w-4} = \begin{cases} M_{14} & \text{if } 2(k-1) \leq w-4 \leq 4(k-1) \\ \infty & \text{otherwise} \end{cases}$$

Note that when finite, $M_{k-2,w-i} = M_{2i}$. However by Lemma 17, none of these can be the minimum value. Note that $M(k-1, w-2)$ is finite for $2(k-1) \leq w-2 \leq 4(k-1)$, and thus is finite for $2k \leq w \leq 4k-2$. Hence by Lemma 17, $M(k,w) = M_2$ for $2k \leq w \leq 4k-2$. For $w = 4k-1$, both $M(k-1, w-3)$ and $M(k-1, w-4)$ are finite. Hence by Lemma 17, $M(k,w) = M_3 = M_2$ for $w = 4k-1$. Finally, for $w = 4k$, only $M(k-1, w-4)$ is finite. Hence by Lemma 17, $M(k,w) = M_4 = M_2$ for $w = 4k$. Therefore, $M(k,w) = M_2$ for $2k \leq w \leq 4k$. □

Lemma 19 *For fixed w and $\frac{w}{4} < k \leq \frac{w}{2}$, the function $M(k,w)$ is non-decreasing. For $k = \frac{w}{4}$, $M(k+1,w) \leq M(k,w)$.*

Proof: $\frac{w}{4} \leq k$ and $k+1 \leq \frac{w}{2}$ implies that both $2k \leq w \leq 4k$ and $2(k+1) \leq w \leq 4(k+1)$. By Theorem 18, we know that $M(k,w)$ and $M(k+1,w)$ are finite over these ranges. Therefore

$$\begin{aligned} M(k,w) &= F_{k+2} + F_{\lfloor \frac{w-2k+4}{2} \rfloor} + F_{\lfloor \frac{w-2k+5}{2} \rfloor} - 2 \\ M(k+1,w) &= F_{k+3} + F_{\lfloor \frac{w-2(k+1)+4}{2} \rfloor} + F_{\lfloor \frac{w-2(k+1)+5}{2} \rfloor} - 2 \end{aligned}$$

First, we compare $M(k,w)$ to $M(k+1,w)$ for $\frac{w}{4} < k$, $k+1 \leq \frac{w}{2}$. The comparison is split into two cases: w even, and w odd.

Case 1: w is even. $w \leq 4k-2$ implies that $\frac{w-2k+4}{2} \leq k+1$. Hence

$$\begin{aligned}
M(k,w) &= F_{k+2} + F_{\lfloor \frac{w-2k+4}{2} \rfloor} + F_{\lfloor \frac{w-2k+5}{2} \rfloor} - 2 \\
&= F_{k+2} + F_{\frac{w-2k+4}{2}} + F_{\frac{w-2k+4}{2}} - 2 \\
&< F_{k+2} + F_{\frac{w-2k+4}{2}} + F_{\frac{w-2k+2}{2}} + F_{\frac{w-2k+2}{2}} - 2 \\
&= F_{k+2} + F_{\frac{w-2k+4}{2}} + F_{\lfloor \frac{w-2k+2}{2} \rfloor} + F_{\lfloor \frac{w-2k+3}{2} \rfloor} - 2 \\
&\leq F_{k+3} + F_{\lfloor \frac{w-2(k+1)+4}{2} \rfloor} + F_{\lfloor \frac{w-2(k+1)+5}{2} \rfloor} - 2 \\
&= M(k+1,w)
\end{aligned}$$

Case 2: w is odd. $w \leq 4k-1$ implies that $\frac{w-2k+3}{2} \leq k+1$. Hence

$$\begin{aligned}
M(k,w) &= F_{k+2} + F_{\lfloor \frac{w-2k+4}{2} \rfloor} + F_{\lfloor \frac{w-2k+5}{2} \rfloor} - 2 \\
&= F_{k+2} + F_{\frac{w-2k+3}{2}} + F_{\frac{w-2k+5}{2}} - 2 \\
&= F_{k+2} + F_{\frac{w-2k+3}{2}} + F_{\frac{w-2k+1}{2}} + F_{\frac{w-2k+3}{2}} - 2 \\
&= F_{k+2} + F_{\frac{w-2k+3}{2}} + F_{\lfloor \frac{w-2k+2}{2} \rfloor} + F_{\lfloor \frac{w-2k+3}{2} \rfloor} - 2 \\
&\leq F_{k+3} + F_{\lfloor \frac{w-2(k+1)+4}{2} \rfloor} + F_{\lfloor \frac{w-2(k+1)+5}{2} \rfloor} - 2 \\
&= M(k+1,w)
\end{aligned}$$

Next, we show that for $k = \frac{w}{4}$, $M(k+1,w) \leq M(k,w)$. Note that for $k \geq 1$, $F_{k+1} \leq 2F_k$ (they are equal for $k=2$). By adding $2F_{k+1}$ to both sides of the inequality we get $3F_{k+1} \leq 2F_{k+2}$. Hence

$$\begin{aligned}
M(k+1,w) &= F_{k+3} + F_{\lfloor \frac{w-2(k+1)+4}{2} \rfloor} + F_{\lfloor \frac{w-2(k+1)+5}{2} \rfloor} - 2 \\
&= F_{k+3} + F_{\lfloor \frac{2k+2}{2} \rfloor} + F_{\lfloor \frac{2k+3}{2} \rfloor} - 2 \\
&= F_{k+3} + F_{\lfloor k+1 \rfloor} + F_{\lfloor k+\frac{3}{2} \rfloor} - 2 \\
&= F_{k+2} + 3F_{k+1} - 2 \\
&\leq F_{k+2} + 2F_{k+2} - 2 \\
&= F_{k+2} + 2F_{\frac{w-2k+4}{2}} - 2 \\
&= F_{k+2} + F_{\lfloor \frac{w-2k+4}{2} \rfloor} + F_{\lfloor \frac{w-2k+5}{2} \rfloor} - 2 \\
&= M(k,w)
\end{aligned}$$

□

Theorem 20 *Define $N(w)$ to be the minimum number of leaves in a height-balanced 2-4 tree with weight w. For $w \geq 2$*

1. $w = 4k - 3 \Rightarrow N(w) = 2F_{k+2} - 2.$

2. $w = 4k - 2 \Rightarrow N(w) = F_{k+3} + F_{k+1} - 2.$

3. $w = 4k - 1 \Rightarrow N(w) = F_{k+4} - 2.$

4. $w = 4k \Rightarrow N(w) = F_{k+3} + 2F_{k+1} - 2.$

Proof: For $w \geq 2$, the form of the implications in the statement of this Theorem implies that $2k \leq w \leq 4k$. Thus by Theorem 18, $M(k, w)$ is finite. To find the minimum number of leaves $N(w)$ in a height-balanced 2-4 tree with weight w, we know by Lemma 19 that for $\frac{w}{4} < k \leq \frac{w}{2}$ we should choose the height of the tree to be the smallest k such that $M(k, w)$ is finite. In other words, $k = \lceil \frac{w}{4} \rceil$. Also by Lemma 19, we know that when $k = \frac{w}{4}$ we should choose the height of the tree to be $k + 1$. We use the appropriate height along with the equation for $M(k, w)$ given by Theorem 18 to prove each of the implications.

Implication 1: $w = 4k - 3 \Rightarrow N(w) = 2F_{k+2} - 2.$

$$\begin{aligned} M(k,w) &= F_{k+2} + F_{\lfloor \frac{2k+1}{2} \rfloor} + F_{\lfloor \frac{2k+2}{2} \rfloor} - 2 \\ &= F_{k+2} + F_{\lfloor k+\frac{1}{2} \rfloor} + F_{\lfloor k+1 \rfloor} - 2 \\ &= F_{k+2} + F_k + F_{k+1} - 2 \\ &= 2F_{k+2} - 2 \end{aligned}$$

Implication 2: $w = 4k - 2 \Rightarrow N(w) = F_{k+3} + F_{k+1} - 2.$

$$\begin{aligned} M(k,w) &= F_{k+2} + F_{\lfloor \frac{2k+2}{2} \rfloor} + F_{\lfloor \frac{2k+3}{2} \rfloor} - 2 \\ &= F_{k+2} + F_{\lfloor k+1 \rfloor} + F_{\lfloor k+\frac{3}{2} \rfloor} - 2 \\ &= F_{k+2} + F_{k+1} + F_{k+1} - 2 \\ &= F_{k+3} + F_{k+1} - 2 \end{aligned}$$

Implication 3: $w = 4k - 1 \Rightarrow N(w) = F_{k+4} - 2.$

$$\begin{aligned} M(k,w) &= F_{k+2} + F_{\lfloor \frac{2k+3}{2} \rfloor} + F_{\lfloor \frac{2k+4}{2} \rfloor} - 2 \\ &= F_{k+2} + F_{\lfloor k+\frac{3}{2} \rfloor} + F_{\lfloor k+2 \rfloor} - 2 \\ &= F_{k+2} + F_{k+1} + F_{k+2} - 2 \\ &= F_{k+4} - 2 \end{aligned}$$

Implication 4: $w = 4k \Rightarrow N(w) = F_{k+3} + 2F_{k+1} - 2$.

$$\begin{aligned} M(k+1,w) &= F_{k+3} + F_{\lfloor \frac{2k+2}{2} \rfloor} + F_{\lfloor \frac{2k+3}{2} \rfloor} - 2 \\ &= F_{k+3} + F_{\lfloor k+1 \rfloor} + F_{\lfloor k+\frac{3}{2} \rfloor} - 2 \\ &= F_{k+3} + F_{k+1} + F_{k+1} - 2 \\ &= F_{k+3} + 2F_{k+1} - 2 \end{aligned}$$

□

Theorem 20 gives us a relationship between tree weight w and the minimum number of leaves $N(w)$ to achieve that weight. Next, we show that $w = w_H(n)$, and that $w_H(n)$ is non-decreasing. This allows us to use Theorem 20 to derive a relationship between n and $w_H(n)$.

Lemma 21 *For all $N(w) \leq n < N(w+1)$, there exists a height-balanced 2-4 tree with n leaves and weight $w' \geq w$.*

Proof: Let T be a height-balanced 2-4 tree with $N(w)$ leaves and weight w. Define $h(T_i)$ to be the height of the subtree rooted at node i, and $s(i)$ to be a sibling of node i. Consider the shortest path in T. For all $1 \leq i \leq \ell$ on that path, either $h(T_i) = h(T_{s(i)})$ or $h(T_i) + 1 = h(T_{s(i)})$. The fact that $h(T_i) \leq h(T_{s(i)})$ is true by definition of i being on the shortest path in T. The fact that the subtree heights differ by at most 1 is true from the fact that the tree is height-balanced.

Consider what happens when we add a leaf node u to the shortest path in T. Recall that ℓ is the leaf node with the shortest path to the root. We create a new node p and add u and ℓ as children of p. We place p into the old spot that ℓ occupied. Before the addition of the new node, $h(\ell)$ was equal to or 1 less than its siblings. The height of p (now in ℓ's

spot) must now be equal to or 1 greater than its siblings (since $h(p) = h(\ell) + 1$). This means p is still height-balanced with its siblings. However, increasing p's height can cause the height of its parent g to increase by 1. But since g is on the path to the root, it has the same relationship with its siblings that p had with its siblings, and we can apply the same argument to g. In fact, we can recursively apply the same argument all the way up to the root, and therefore, the entire tree remains height-balanced. The weight of ℓ has increased by 2, and the weight of all other leaf nodes have remained the same. Therefore, adding a new member to a height-balanced tree in the fashion just described cannot decrease the tree weight, and this fact proves the lemma. □

Theorem 22 *The relationship between the number of leaves $N(w)$ and the worst tree weight $w_H(n)$ in a height-balanced 2-4 tree is given by*

$$\begin{aligned}
2F_{k+2} - 2 \;&\leq\; n \;<\; F_{k+3} + F_{k+1} - 2, && w_H(n) = 4k - 3 \\
F_{k+3} + F_{k+1} - 2 \;&\leq\; n \;<\; F_{k+4} - 2, && w_H(n) = 4k - 2 \\
F_{k+4} - 2 \;&\leq\; n \;<\; F_{k+3} + 2F_{k+1} - 2, && w_H(n) = 4k - 1 \\
F_{k+3} + 2F_{k+1} - 2 \;&\leq\; n \;<\; 2F_{k+3} - 2, && w_H(n) = 4k
\end{aligned}$$

Proof: We know that $w_H(n)$ is the maximum tree weight for any height-balanced 2-4 tree with n leaves. Given w, by Theorem 20 $N(w)$ is the fewest number of leaves in a height-balanced 2-4 tree with weight w. Also, by Theorem 20, $N(w+1) > N(w)$ is the fewest number of leaves in a height-balanced 2-4 tree weight $w+1$. We show that $w_H(n) = w$ for $N(w) \leq n < N(w+1)$, which proves the Theorem. There are two inequalities to show.

(i) $w_H(n) \geq w$. This is true by Lemma 21.

(ii) $w_H(n) \leq w$. For the purpose of contradiction, suppose that $w' = w_H(n) > w$. This means that $w' \geq w + 1$, and the smallest n such that there is a height-balanced 2-4 with weight $w' = w_H(n)$ is $n \geq N(w+1)$. But since we are considering the range $N(w) \leq n < N(w+1)$, we have a contradiction. □

Using the relationship between n and $w_H(n)$ given by Theorem 22, we now prove that $w_H(n) \sim 4 \log_\phi n$, where $\phi = (1 + \sqrt{5})/2 \approx 1.61803$.

Theorem 23 $w_H(n) \sim 4\log_\phi n$.

Proof: We show that $4\log_\phi n - c_1 \leq w_H(n) \leq 4\log_\phi n - c_2$, where c_1 and c_2 are constants. This implies that $\lim_{n\to\infty} w_H(n)/4\log_\phi n = 1$. We show that the bounds hold for each of the ranges given in Theorem 22, which covers all values of $w_H(n) \geq 2$. The kth Fibonacci number F_k can be computed as $F_k = \phi^k/\sqrt{5}$ rounded to the nearest integer [18].

1. $2F_{k+2} - 2 \leq n < F_{k+3} + F_{k+1} - 2$, $w_H(n) = 4k - 3$.

   For all $k \geq 1$, choosing $c_1 \geq 4\log_\phi((\frac{\phi^3}{\sqrt{5}} + \frac{1}{\phi^k}) + (\frac{\phi}{\sqrt{5}} + \frac{1}{\phi^k}) - \frac{3}{\phi^k})) + 3$ allows us to prove the following.

$$\begin{aligned}
4k - 3 &= w_H(n) \\
4k + 4\log_\phi((\tfrac{\phi^3}{\sqrt{5}} + \tfrac{1}{\phi^k}) + (\tfrac{\phi}{\sqrt{5}} + \tfrac{1}{\phi^k}) - \tfrac{3}{\phi^k})) - c_1 &\leq \\
4\log_\phi \phi^k + 4\log_\phi((\tfrac{\phi^3}{\sqrt{5}} + \tfrac{1}{\phi^k}) + (\tfrac{\phi}{\sqrt{5}} + \tfrac{1}{\phi^k}) - \tfrac{3}{\phi^k})) - c_1 &= \\
4\log_\phi(\phi^k((\tfrac{\phi^3}{\sqrt{5}} + \tfrac{1}{\phi^k}) + (\tfrac{\phi}{\sqrt{5}} + \tfrac{1}{\phi^k}) - \tfrac{3}{\phi^k})) - c_1 &= \\
4\log_\phi((\tfrac{\phi^{k+3}}{\sqrt{5}} + 1) + (\tfrac{\phi^{k+1}}{\sqrt{5}} + 1) - 3) - c_1 &= \\
4\log_\phi(F_{k+3} + F_{k+1} - 3) - c_1 &\leq \\
4\log_\phi n - c_1 &\leq
\end{aligned}$$

   For all $k \geq 1$, choosing $c_2 \leq 4\log_\phi((2(\frac{\phi^2}{\sqrt{5}} - \frac{1}{\phi^k}) - \frac{2}{\phi^k})) + 3$ allows us to prove the following.

$$\begin{aligned}
w_H(n) &= 4k - 3 \\
&\leq 4k + 4\log_\phi((2(\tfrac{\phi^2}{\sqrt{5}} - \tfrac{1}{\phi^k}) - \tfrac{2}{\phi^k})) - c_2 \\
&= 4\log_\phi \phi^k + 4\log_\phi((2(\tfrac{\phi^2}{\sqrt{5}} - \tfrac{1}{\phi^k}) - \tfrac{2}{\phi^k})) - c_2 \\
&= 4\log_\phi(\phi^k(2(\tfrac{\phi^2}{\sqrt{5}} - \tfrac{1}{\phi^k}) - \tfrac{2}{\phi^k})) - c_2 \\
&= 4\log_\phi(2(\tfrac{\phi^{k+2}}{\sqrt{5}} - 1) - 2) - c_2 \\
&\leq 4\log_\phi(2F_{k+2} - 2) - c_2 \\
&\leq 4\log_\phi n - c_2
\end{aligned}$$

2. $F_{k+3} + F_{k+1} - 2 \leq n < F_{k+4} - 2$, $w_H(n) = 4k - 2$.

For all $k \geq 1$, choosing $c_1 \geq 4\log_\phi((\frac{\phi^4}{\sqrt{5}} + \frac{1}{\phi^k}) - \frac{3}{\phi^k}) + 2$ allows us to prove the following.

$$\begin{aligned}
4k-2 &= w_H(n) \\
4k + 4\log_\phi((\tfrac{\phi^4}{\sqrt{5}} + \tfrac{1}{\phi^k}) - \tfrac{3}{\phi^k}) - c_1 &\leq \\
4\log_\phi \phi^k + 4\log_\phi((\tfrac{\phi^4}{\sqrt{5}} + \tfrac{1}{\phi^k}) - \tfrac{3}{\phi^k}) - c_1 &= \\
4\log_\phi(\phi^k((\tfrac{\phi^4}{\sqrt{5}} + \tfrac{1}{\phi^k}) - \tfrac{3}{\phi^k})) - c_1 &= \\
4\log_\phi((\tfrac{\phi^{k+4}}{\sqrt{5}} + 1) - 3) - c_1 &= \\
4\log_\phi(F_{k+4} - 3) - c_1 &\leq \\
4\log_\phi n - c_1 &\leq
\end{aligned}$$

For all $k \geq 1$, choosing $c_2 \leq 4\log_\phi((\frac{\phi^3}{\sqrt{5}} - \frac{1}{\phi^k}) + (\frac{\phi}{\sqrt{5}} - \frac{1}{\phi^k}) - \frac{2}{\phi^k}) + 2$ allows us to prove the following.

$$\begin{aligned}
w_H(n) &= 4k-2 \\
&\leq 4k + 4\log_\phi((\tfrac{\phi^3}{\sqrt{5}} - \tfrac{1}{\phi^k}) + (\tfrac{\phi}{\sqrt{5}} - \tfrac{1}{\phi^k}) - \tfrac{2}{\phi^k}) - c_2 \\
&= 4\log_\phi \phi^k + 4\log_\phi((\tfrac{\phi^3}{\sqrt{5}} - \tfrac{1}{\phi^k}) + (\tfrac{\phi}{\sqrt{5}} - \tfrac{1}{\phi^k}) - \tfrac{2}{\phi^k}) - c_2 \\
&= 4\log_\phi(\phi^k((\tfrac{\phi^3}{\sqrt{5}} - \tfrac{1}{\phi^k}) + (\tfrac{\phi}{\sqrt{5}} - \tfrac{1}{\phi^k}) - \tfrac{2}{\phi^k})) - c_2 \\
&= 4\log_\phi((\tfrac{\phi^{k+3}}{\sqrt{5}} - 1) + (\tfrac{\phi^{k+1}}{\sqrt{5}} - 1) - 2) - c_2 \\
&\leq 4\log_\phi(F_{k+3} + F_{k+1} - 2) - c_2 \\
&\leq 4\log_\phi n - c_2
\end{aligned}$$

3. $F_{k+4} - 2 \leq n < F_{k+3} + 2F_{k+1} - 2$, $w_H(n) = 4k-1$.

For all $k \geq 1$, choosing $c_1 \geq 4\log_\phi((\frac{\phi^3}{\sqrt{5}} + \frac{1}{\phi^k}) + 2(\frac{\phi}{\sqrt{5}} + \frac{1}{\phi^k}) - \frac{3}{\phi^k}) + 1$ allows us to prove the following.

$$\begin{aligned}
4k-1 &= w_H(n) \\
4k + 4\log_\phi((\tfrac{\phi^3}{\sqrt{5}} + \tfrac{1}{\phi^k}) + 2(\tfrac{\phi}{\sqrt{5}} + \tfrac{1}{\phi^k}) - \tfrac{3}{\phi^k}) - c_1 &\leq \\
4\log_\phi \phi^k + 4\log_\phi((\tfrac{\phi^3}{\sqrt{5}} + \tfrac{1}{\phi^k}) + 2(\tfrac{\phi}{\sqrt{5}} + \tfrac{1}{\phi^k}) - \tfrac{3}{\phi^k}) - c_1 &= \\
4\log_\phi(\phi^k((\tfrac{\phi^3}{\sqrt{5}} + \tfrac{1}{\phi^k}) + 2(\tfrac{\phi}{\sqrt{5}} + \tfrac{1}{\phi^k}) - \tfrac{3}{\phi^k})) - c_1 &= \\
4\log_\phi((\tfrac{\phi^{k+3}}{\sqrt{5}} + 1) + 2(\tfrac{\phi^{k+1}}{\sqrt{5}} + 1) - 3) - c_1 &= \\
4\log_\phi(F_{k+3} + 2F_{k+1} - 3) - c_1 &\leq \\
4\log_\phi n - c_1 &\leq
\end{aligned}$$

For all $k \geq 1$, choosing $c_2 \leq 4\log_\phi((\frac{\phi^4}{\sqrt{5}} - \frac{1}{\phi^k}) - \frac{2}{\phi^k}) + 1$ allows us to prove the following.

$$\begin{aligned}
w_H(n) &= 4k - 1 \\
&\leq 4k + 4\log_\phi((\tfrac{\phi^4}{\sqrt{5}} - \tfrac{1}{\phi^k}) - \tfrac{2}{\phi^k}) - c_2 \\
&= 4\log_\phi \phi^k + 4\log_\phi((\tfrac{\phi^4}{\sqrt{5}} - \tfrac{1}{\phi^k}) - \tfrac{2}{\phi^k}) - c_2 \\
&= 4\log_\phi(\phi^k((\tfrac{\phi^4}{\sqrt{5}} - \tfrac{1}{\phi^k}) - \tfrac{2}{\phi^k})) - c_2 \\
&= 4\log_\phi((\tfrac{\phi^{k+4}}{\sqrt{5}} - 1) - 2) - c_2 \\
&\leq 4\log_\phi(F_{k+4} - 2) - c_2 \\
&\leq 4\log_\phi n - c_2
\end{aligned}$$

4. $F_{k+3} + 2F_{k+1} - 2 \leq n < 2F_{k+3} - 2$, $w_H(n) = 4k$.

For all $k \geq 1$, choosing $c_1 \geq 4\log_\phi(2(\frac{\phi^3}{\sqrt{5}} + \frac{1}{\phi^k}) - \frac{3}{\phi^k})$ allows us to prove the following.

$$\begin{aligned}
4k &= w_H(n) \\
4k + 4\log_\phi(2(\tfrac{\phi^3}{\sqrt{5}} + \tfrac{1}{\phi^k}) - \tfrac{3}{\phi^k}) - c_1 &\leq \\
4\log_\phi \phi^k + 4\log_\phi(2(\tfrac{\phi^3}{\sqrt{5}} + \tfrac{1}{\phi^k}) - \tfrac{3}{\phi^k}) - c_1 &= \\
4\log_\phi(\phi^k(2(\tfrac{\phi^3}{\sqrt{5}} + \tfrac{1}{\phi^k}) - \tfrac{3}{\phi^k})) - c_1 &= \\
4\log_\phi(2(\tfrac{\phi^{k+3}}{\sqrt{5}} + 1) - 3) - c_1 &= \\
4\log_\phi(2F_{k+3} - 3) - c_1 &\leq \\
4\log_\phi n - c_1 &\leq
\end{aligned}$$

For all $k \geq 1$, choosing $c_2 \leq 4\log_\phi((\frac{\phi^3}{\sqrt{5}} - \frac{1}{\phi^k}) + 2(\frac{\phi}{\sqrt{5}} - \frac{1}{\phi^k}) - \frac{2}{\phi^k})$ allows us to prove the following.

$$\begin{aligned}
w_H(n) &= 4k \\
&\leq 4k + 4\log_\phi((\tfrac{\phi^3}{\sqrt{5}} - \tfrac{1}{\phi^k}) + 2(\tfrac{\phi}{\sqrt{5}} - \tfrac{1}{\phi^k}) - \tfrac{2}{\phi^k}) - c_2 \\
&= 4\log_\phi \phi^k + 4\log_\phi((\tfrac{\phi^3}{\sqrt{5}} - \tfrac{1}{\phi^k}) + 2(\tfrac{\phi}{\sqrt{5}} - \tfrac{1}{\phi^k}) - \tfrac{2}{\phi^k}) - c_2 \\
&= 4\log_\phi(\phi^k((\tfrac{\phi^3}{\sqrt{5}} - \tfrac{1}{\phi^k}) + 2(\tfrac{\phi}{\sqrt{5}} - \tfrac{1}{\phi^k}) - \tfrac{2}{\phi^k})) - c_2 \\
&= 4\log_\phi((\tfrac{\phi^{k+3}}{\sqrt{5}} - 1) + 2(\tfrac{\phi^{k+1}}{\sqrt{5}} - 1) - 2) - c_2 \\
&\leq 4\log_\phi(F_{k+3} + 2F_{k+1} - 2) - c_2 \\
&\leq 4\log_\phi n - c_2
\end{aligned}$$

□

4.3.3 Worst-case Weight-Balanced Tree Analysis

Recall that we designed both weight-balanced 2-3 and weight-balanced 2-3-4 algorithms. Our analysis shows that they both have the same worst-case tree weight bounds. Define $w_W(n)$ to be the maximum weight of any weight-balanced 2-3 or weight-balanced 2-3-4 tree with n members. The analysis of the bounds on $w_W(n)$ is divided into three steps. First, we describe the structure of minimum leaf weight-balanced 2-3 or 2-3-4 trees that achieve weight w. We then show that $w = w_W(n)$. Using these two results we prove that $w_W(n) \sim \log_b n$, where b is the real solution to the equation $b^3 = b+1$. Solving the equation gives us $b \approx 1.32472$.

Theorem 24 *Define $N(w)$ to be the minimum number of leaves in a weight-balanced 2-3 or 2-3-4 tree with weight $= w$. Then $N(2) = 2$, $N(3) = 3$, $N(4) = 4$, and $N(w)$ satisfies the equation: $N(w) = N(w-2) + N(w-3)$, for $w \geq 5$.*

Proof: We can hand-construct the trees for $w = 2, 3, 4$ to verify that $N(2) = 2$, $N(3) = 3$, and $N(4) = 4$ for both weight-balanced 2-3 and weight-balanced 2-3-4 trees. For $w \geq 5$, the minimum number of leaves in a weight-balanced 2-3 or 2-3-4 tree with weight w must have one of the forms shown in Figure 4.17. This is true because of the restriction on the degree of the root, the fact that the root must be weight-balanced, and because we should use the minimum number of leaves in each sub-tree. However, notice that the structure must be the one on the left (the one where the root has degree 2) because Figure 4.18 shows that the structures with degrees 3 and 4 can be reduced to the one with degree 2. Also, note that these trees do not have the minimum number of leaves because the sub-trees do not have minimal size. Therefore, $N(2) = 2$, $N(3) = 3$, $N(4) = 4$, and $N(w) = N(w-2) + N(w-3)$, for $w \geq 5$. □

Theorem 25 *$a_L b^w \leq N(w) \leq a_U b^w$, where $a_L = 1.13965$, $a_U = 1.299$, and $b \approx 1.32472$ is the real solution to the equation $b^3 = b + 1$.*

Proof: The proof is by induction on w. The base case is for $w = 2, 3, 4$. By Theorem 24, we know that $N(2) = 2$, $N(3) = 3$, and $N(4) = 4$ respectively. It is easy to verify that the inequalities hold for the base cases.

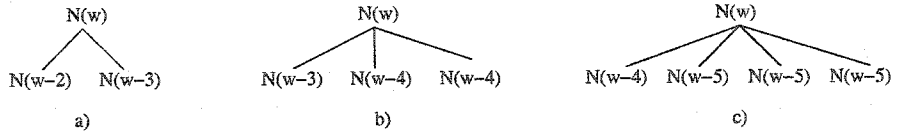

Figure 4.17: Consider possible weight-balanced tree structures with weight w and the minimum number of leaves to achieve that weight. Parts a and b show the possibilities for weight-balanced 2-3 trees, and parts a, b, and c show the possibilities for weight-balanced 2-3-4 trees.

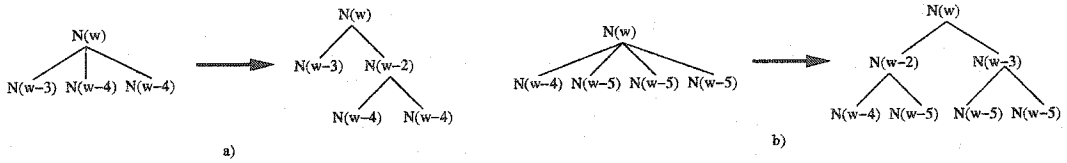

Figure 4.18: Part a shows that the tree with degree 3 cannot have the minimum number of leaves because it can be reduced to a tree with degree 2 that does not have the minimum number of leaves. Part b shows that the tree with degree 4 cannot be one with minimum leaves either.

For $w \geq 5$, we use the equation $N(w) = N(w-2) + N(w-3)$, which was derived in Theorem 24. Starting with the induction hypothesis we can derive the following:

$$\begin{array}{rcccl} a_L b^{w-2} + a_L b^{w-3} & \leq & N(w-2) + N(w-3) & \leq & a_U b^{w-2} + a_U b^{w-3} \\ a_L b^{w-3}(b+1) & \leq & N(w) & \leq & a_U b^{w-3}(b+1) \\ a_L b^{w} & \leq & N(w) & \leq & a_U b^{w} \end{array}$$

□

Theorem 24 gives us a relationship between tree weight w and the minimum number of leaves $N(w)$ to achieve that weight. We show that $w = w_W(n)$ and $w_W(n)$ is non-decreasing, allowing us to use Theorem 24 to derive a relationship between n and $w_W(n)$.

Lemma 26 *For all $N(w) \leq n < N(w+1)$ there exists a weight-balanced tree with n leaves and weight $w' \geq w$.*

Proof: Note that none of the weight-balanced insertion rules can decrease the weight of a node (see Figures 4.9 and 4.13 in section 4.2.3). So simply applying the appropriate insertion rule to a weight-balanced tree with weight w ensures that the resulting tree has weight $w' \geq w$. □

Theorem 27 *$w_W(n) = w$ for $N(w) \leq n < N(w+1)$.*

Proof: We know that $w_W(n)$ is the maximum tree weight for any weight-balanced tree with n leaves. Given w, by Theorem 24 $N(w)$ is the fewest number of leaves in a weight-balanced tree with weight w. Also, by Theorem 24, $N(w+1) > N(w)$ is the fewest number of leaves in a weight-balanced tree with weight $w+1$. We show that $w_W(n) = w$ for $N(w) \leq n < N(w+1)$, which proves the Theorem.

$w_W(n) \geq w$. This is true by Lemma 26.

$w_W(n) \leq w$. For the purpose of contradiction, suppose that $w' = w_W(n) > w$. This means that $w' \geq w+1$, and the smallest n such that there is a weight-balanced tree with weight $w' = w_W(n)$ is $n \geq N(w+1)$. But since we are considering the range $N(w) \leq n < N(w+1)$, we have a contradiction. □

Using the relationship between n and $w_W(n)$ given by Theorem 27, we now prove that $w_W(n) \sim \log_b n$, where $b \approx 1.32472$ is the real solution to the equation $b^3 = b+1$.

Theorem 28 *$w_W(n) \sim \log_b n$.*

Proof: We show that $\log_b n - c_1 \leq w_W(n) \leq \log_b n - c_2$, where c_1 and c_2 are constants, and $b \approx 1.32472$ is the real solution to the equation $b^3 = b+1$. This implies that $\lim_{n\to\infty} w_W(n)/\log_b n = 1$. We show that the bounds hold for the range given in Theorem 27, which covers all values of $w_W(n) \geq 2$.

Recall from Theorem 25 that $a_L = 1.13965$, $a_U = 1.299$, $b \approx 1.32472$. Choosing $c_1 \geq \log_b(a_U b - \frac{1}{b^w})$ and $c_2 \leq \log_b(a_L)$ allows us to prove the theorem:

$$
\begin{array}{rcccl}
w & = & w_W(n) & = & w \\
w + \log_b(a_U b - \frac{1}{b^w}) - c_1 & \leq & & \leq & w + \log_b(a_L) - c_2 \\
\log_b b^w + \log_b(a_U b - \frac{1}{b^w}) - c_1 & = & & = & \log_b b^w + \log_b(a_L) - c_2 \\
\log_b(a_U b^{w+1} - 1) - c_1 & = & & = & \log_b(a_L b^w) - c_2 \\
\log_b(N(w+1) - 1) - c_1 & \leq & & \leq & \log_b(N(w)) - c_2 \\
\log_b n - c_1 & \leq & & \leq & \log_b n - c_2
\end{array}
$$

□

4.3.4 *Discussion of the Worst-case Tree Weight Analysis*

We derived worst-case tree weight bounds for each of our new algorithms. The best possible tree weight $w(n)$ for n members is $w(n) \sim 3 \log_3 n$ [49]. Consider the ratio of our worst-case tree weight bounds to that of $w(n)$.

$$\frac{w_W(n)}{w(n)} \approx 1.30229 < \frac{w_B(n)}{w(n)} \approx 2.11328 < \frac{w_H(n)}{w(n)} \approx 3.04403$$

The relative ordering of our algorithms with respect to tree weight is what we would expect. The weight-balanced algorithm was designed with tree weight in mind, while the height-balanced algorithm had the most relaxed definition of a balanced node. But we also expect the order of performance to be reversed when considering restructuring costs, since more restrictive tree structures would seem to require more restructuring to maintain. However, since we cannot actually analyze the restructuring cost of our algorithms, we turn to simulations to study this issue in more detail.

4.4 *Simulation Results*

We use simulations to get a better understanding of the trade-off between the tree structure and restructuring costs of our algorithms. The major difference between our new online algorithms and previous online algorithms is that we maintain tree structures that we believe will lead to good performance. Moyer *et al.* [37] maintained balanced binary tree structures. However, they used switching which can lead to poor performance due to high restructuring costs as seen in Section 4.2. Rodeh *et al.* [45] also maintained balanced binary (AVL) trees. However, we do not compare our algorithms against their work since our height-balanced 2-k trees are a generalization of their AVL trees. As a baseline for comparison, we use the degree-k key trees of Wong *et al.* [53]. They do not perform any restructuring. However, for ADD(u) operations they attempt to place u in a location to keep the tree as full and balanced as possible. Note that the degree-k tree, B-tree of order t, and height-balanced 2-k tree algorithms can all vary the maximum degree of internal nodes, and that we have two choices for our weight-balanced tree algorithm (2-3 or 2-3-4 tree). For each experiment, we use the same maximum node degree (either 3 or 4) for all of our algorithms to give a

fair comparison. Another thing to note is that some of our experiments use non-intuitive initial tree sizes k^i, where k is the maximum degree for our algorithms. The reason is that all of our algorithms reach the same tree structure (complete k-ary tree) at k^i members when repeatedly applying their respective ADD(u) algorithms. Using an initial tree size of k^i means that our algorithms all have the same starting point, which leads to a fair comparison.

We present the results of experiments using three different re-key sequence types: all ADD(u) operations, all DELETE(u) operations, and weighted re-key operations. These experiments are meant to give us insight into the relative performance of the algorithms, and to better understand their trade-offs between tree structure and restructuring cost. In particular, we focus on the following questions:

1. *Is it important to maintain tree structures?* Our algorithms maintain different tree structures at the cost of additional restructuring. Will this lead to better performance than simply letting the trees become unbalanced?

2. *Which of our algorithms achieves the best trade-off between tree structure and restructuring cost?*

4.4.1 All ADD(u) Operations

All of the algorithms have control over where to insert new members, and choose a location to try to maintain balance. In this situation, the additional restructuring required by our algorithms may hurt their performance relative to the degree-k algorithm (which performs no restructuring). For this experiment, our algorithms use maximum degree 4 because the best tree for a sequence of all ADD(u) operations is a star (and 4 is the highest maximum degree for our weight-balanced algorithm).

We present the results of an experiment of $4^9 = 262,144$ ADD(u) operations to an initially empty tree. Our graphs show the average or maximum values over $2,622$ consecutive operations. We compare tree structure to communication costs for average values in Figures 4.19 and 4.20, and for maximum values in Figures 4.21 and 4.22. Note that the

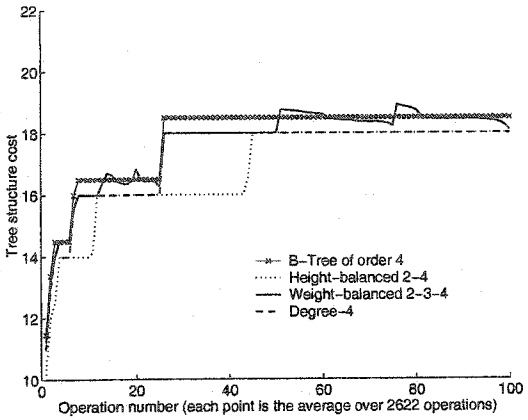

Figure 4.19: Tree structure cost measured as two times the depth of new members for the B-tree of order 4, height-balanced 2-4, weight-balanced 2-3-4, and degree-4 algorithms. Experiment consists of 262, 144 insertions into an initially empty tree. Each point is the average value over 2, 622 consecutive operations.

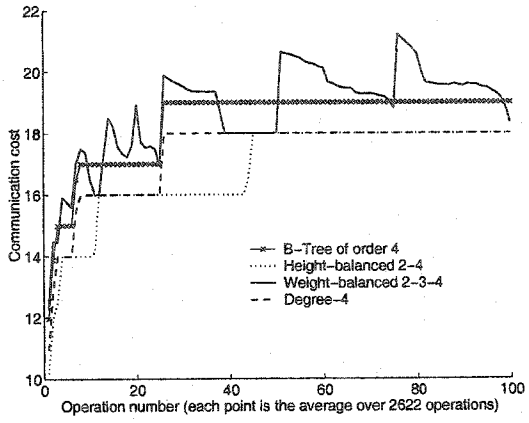

Figure 4.20: Communication cost measured as the number of encrypted messages for the B-tree of order 4, height-balanced 2-4, weight-balanced 2-3-4, and degree-4 algorithms. Experiment consists of 262, 144 insertions into an initially empty tree. Each point is the average value over 2, 622 consecutive operations.

tree structure and communication cost figures show the same range on the y-axis, allowing an easy comparison between tree structure and communication costs. Also note that the maximum value figures have a larger y-axis range than the average value figures. All of the algorithms have very similar tree structure costs for both average and maximum values. However, when considering communication costs, we see that the restructuring required by our B-tree and weight-balanced algorithms cause them to perform worse that the degree-k and height-balanced algorithms (which perform no restructuring). The height-balanced algorithm is able to avoid restructuring because it is always possible to choose an insertion point that will maintain the height-balance of the tree. However, the B-tree and weight-balanced algorithms cannot do this because of their stricter definition of "balance".

Because this experiment considered only ADD(u) operations and all of the algorithms can control where to insert u into their key tree, we expected the restructuring required to maintain our tree structures to cause them to perform worse than the simple degree-k key trees. What is not so intuitive however, is the performance of the B-tree algorithm. It has

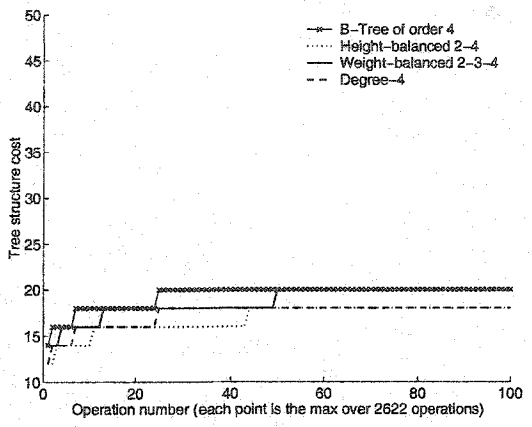

Figure 4.21: Tree structure cost measured as two times the depth of new members for the B-tree of order 4, height-balanced 2-4, weight-balanced 2-3-4, and degree-4 algorithms. Experiment consists of 262, 144 insertions into an initially empty tree. Each point is the maximum value over 2, 622 consecutive operations.

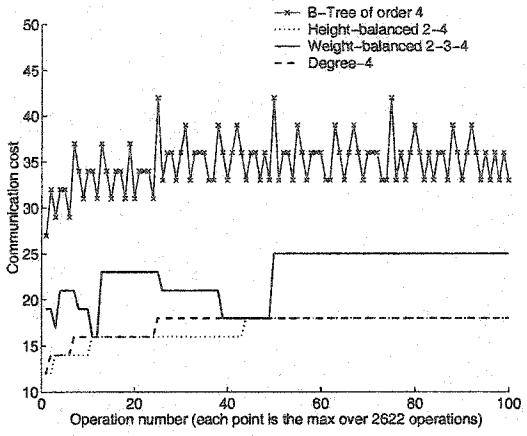

Figure 4.22: Communication cost measured as the number of encrypted messages for the B-tree of order 4, height-balanced 2-4, weight-balanced 2-3-4, and degree-4 algorithms. Experiment consists of 262, 144 insertions into an initially empty tree. Each point is the maximum value over 2, 622 consecutive operations.

decent performance when considering average values, but poor performance when considering maximum values. We see that the performance of the B-tree algorithm is dictated by its restructuring costs which have low average values (Figure 4.23), but high maximum values (Figure 4.24). The reason for the high variance between average and maximum values is understood by considering the B-tree algorithm for handling ADD(u) operations. Most of the time, restructuring costs are low. However we can see that the ADD(u) algorithm in Section 4.2.1 sometimes requires high restructuring costs, in particular, when we insert a new member into a full tree. This causes us to split the children of each full node on the path to the root, which costs us t messages for each such node (where t is the maximum degree for internal nodes).

4.4.2 All DELETE(u) Operations

For this sequence type, the algorithms have no control over which member is being removed from the group. This can give us insight into whether or not maintaining a balanced tree

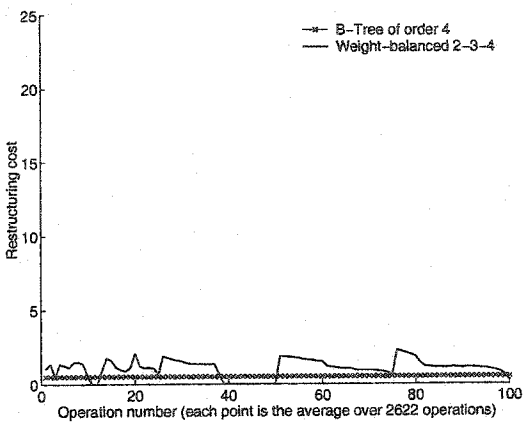

Figure 4.23: Restructuring costs for the B-tree of order 4 and weight-balanced 2-3-4 algorithms. Experiment consists of 262,144 insertions into an initially empty tree. Each point is the average value over 2,622 consecutive operations.

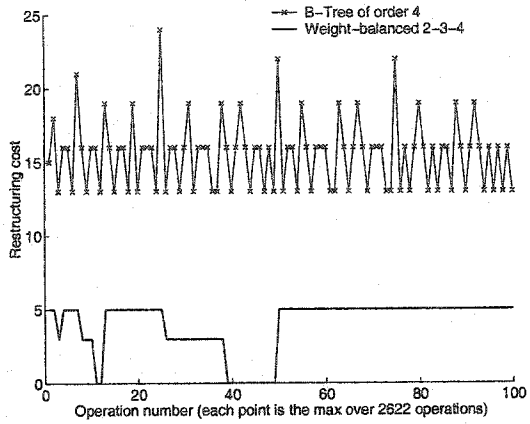

Figure 4.24: Restructuring costs for the B-tree of order 4 and weight-balanced 2-3-4 algorithms. Experiment consists of 262,144 insertions into an initially empty tree. Each point is the maximum value over 2,622 consecutive operations.

structure can aid performance. Our algorithms use maximum degree 3 because it can be shown that a single deletion from a balanced d-ary tree is minimized for $d = 3$ [49].

We present the results of a sequence of all DELETE(u) operations from an initial tree with $3^{11} = 177,147$ members, where the next member to remove is chosen uniformly at random. Our graphs show the average or maximum values over 1,772 consecutive DELETE(u) operations. We compare tree structure to communication costs for average values in Figures 4.25 and 4.26, and for maximum values in Figures 4.27 and 4.28. Our algorithms all have good communication costs when considering average values. However, when we look at the communication costs for maximum values, we see that our algorithms no longer perform so well compared to the degree-k key tree. This is especially true for the height-balanced and weight-balanced algorithms. The height-balanced algorithm has poor performance because of its tree structure. This makes sense because it has the most relaxed definition of a "balanced" node. The weight-balanced algorithm has poor performance because the restructuring cost to maintain the strict definition of "balance" is occasionally quite high. This can be seen by comparing average to maximum restructuring costs in Figures 4.29

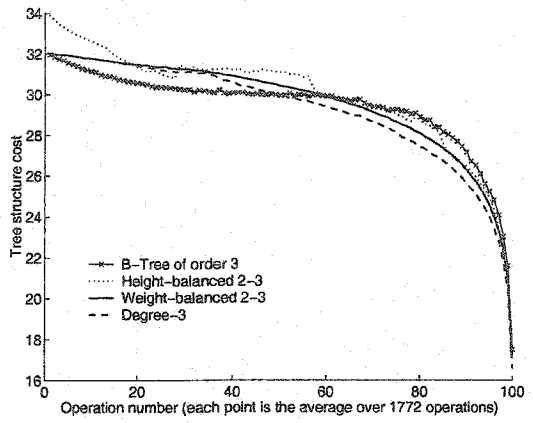

Figure 4.25: Tree structure cost measured as the ancestor weight of deleted members for the B-tree of order 3, height-balanced 2-3, weight-balanced 2-3, and degree-3 algorithms. Experiment consists of 177, 147 deletions in a random order from an initial tree of the same size. Each point is the average value over 1, 772 consecutive operations.

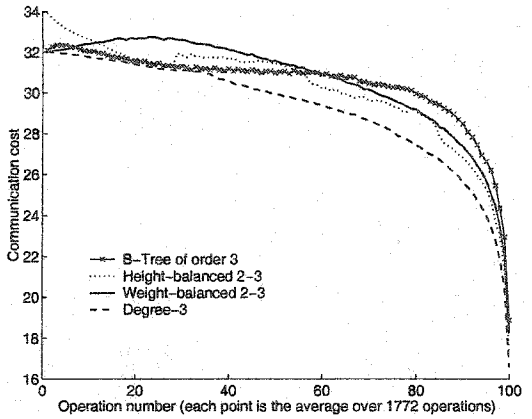

Figure 4.26: Communication cost measured as the number of encrypted messages for the B-tree of order 3, height-balanced 2-3, weight-balanced 2-3, and degree-3 algorithms. Experiment consists of 177, 147 deletions in a random order from an initial tree of the same size. Each point is the average value over 1, 772 consecutive operations.

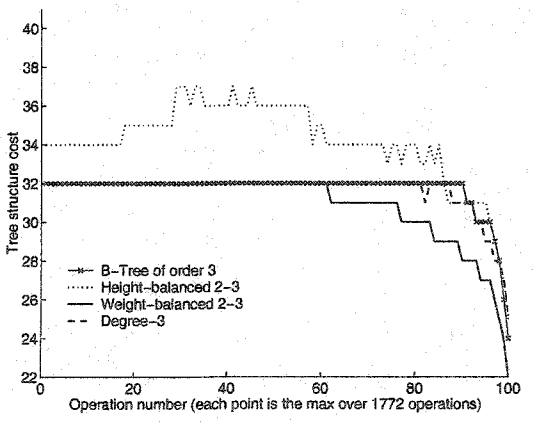

Figure 4.27: Tree structure cost measured as the ancestor weight of deleted members for the B-tree of order 3, height-balanced 2-3, weight-balanced 2-3, and degree-3 algorithms. Experiment consists of 177, 147 deletions in a random order from an initial tree of the same size. Each point is the maximum value over 1, 772 consecutive operations.

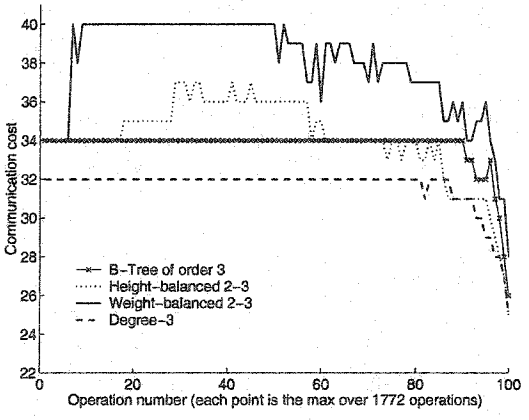

Figure 4.28: Communication cost measured as the number of encrypted messages for the B-tree of order 3, height-balanced 2-3, weight-balanced 2-3, and degree-3 algorithms. Experiment consists of 177, 147 deletions in a random order from an initial tree of the same size. Each point is the maximum value over 1, 772 consecutive operations.

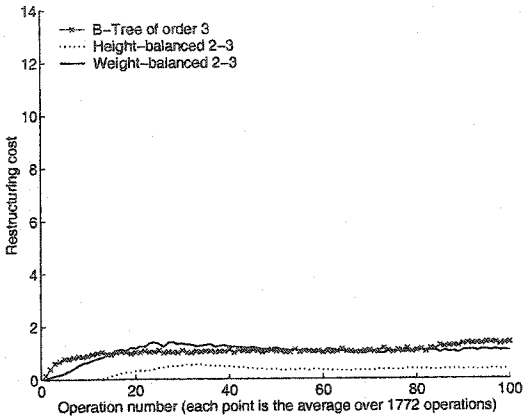

Figure 4.29: Restructuring costs for the B-tree of order 3, height-balanced 2-3, and weight-balanced 2-3 algorithms. Experiment consists of 177,147 deletions in a random order from an initial tree of the same size. Each point is the average value over 1,772 consecutive operations.

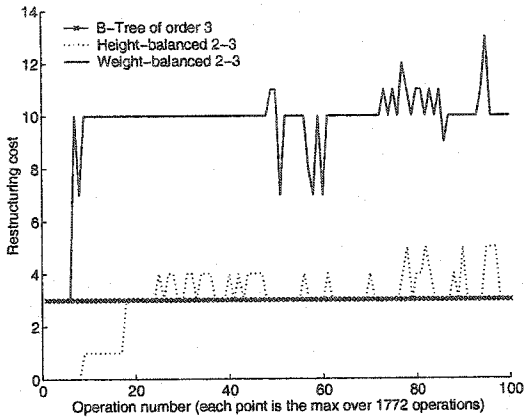

Figure 4.30: Restructuring costs for the B-tree of order 3, height-balanced 2-3, and weight-balanced 2-3 algorithms. Experiment consists of 177,147 deletions in a random order from an initial tree of the same size. Each point is the maximum value over 1,772 consecutive operations.

and 4.30. While the average restructuring costs are low, the maximum restructuring costs are fairly high for the weight-balanced algorithm. The other algorithms have less variance between average and maximum restructuring costs.

So far, it seems that we are better off using the simple degree-k trees that perform no restructuring. Our next experiment is designed to help us understand if maintaining "good" tree structures helps performance when the trees would otherwise become extremely unbalanced. We present the results of a sequence of 177,147 pathological DELETE(u) operations from an initial tree of the same size that are designed to make the degree-k key tree unbalanced. We only show the results using maximum values because the average value graphs show the same trends as the random DELETE(u) sequence (all algorithms have very similar performance).

Looking at Figure 4.31, we see again that our algorithms have good tree structure cost. Note that the tree structure cost for the degree-k algorithm is constant across the whole experiment. This is because we start with a balanced tree, and are making the tree as

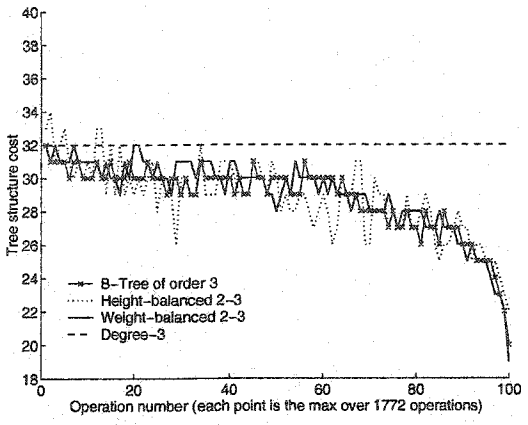

Figure 4.31: Tree structure cost measured as the ancestor weight of deleted members for the B-tree of order 3, height-balanced 2-3, weight-balanced 2-3, and degree-3 algorithms. Experiment consists of 177, 147 pathological deletions from an initial tree of the same size. Each point is the maximum value over 1, 772 consecutive operations.

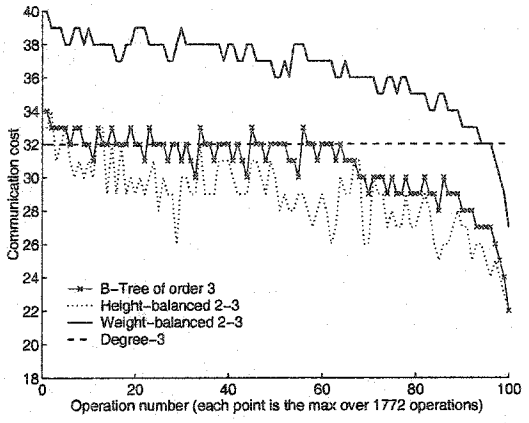

Figure 4.32: Communication cost measured as the number of encrypted messages for the B-tree of order 3, height-balanced 2-3, weight-balanced 2-3, and degree-3 algorithms. Experiment consists of 177, 147 pathological deletions from an initial tree of the same size. Each point is the maximum value over 1, 772 consecutive operations.

unbalanced as possible. This means that there is always a node with the highest cost remaining in the tree. Figure 4.32 shows the results when we measure communication costs. Once again, due to the restructuring costs, the weight-balanced algorithm does not perform as well as we hoped for. However, the height-balanced and B-tree tree algorithms outperform the degree-k key tree over most of the experiment. The reason that the weight-balanced algorithm performs poorly is again due to the occasionally high restructuring costs, as can be seen when comparing average to maximum restructuring costs in Figures 4.33 and 4.34.

4.4.3 Weighted Re-Key Sequences

Our final set of experiments is designed to see how our algorithms perform when there is a mixture of ADD(u) and DELETE(u) operations. A re-key sequence with weight $x\%$ performs random operations that are weighted such that ADD(u) operations occur roughly $x\%$ of the time. Studying this sequence type gives us an idea of how the algorithms might perform in practice (over sequences of both member joins and member leaves). We use

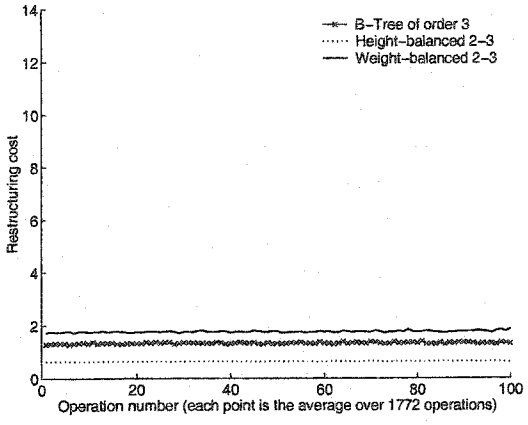

Figure 4.33: Restructuring costs for the B-tree of order 3, height-balanced 2-3, and weight-balanced 2-3 algorithms. Experiment consists of 177, 147 pathological deletions from an initial tree of the same size. Each point is the average value over 1, 772 consecutive operations.

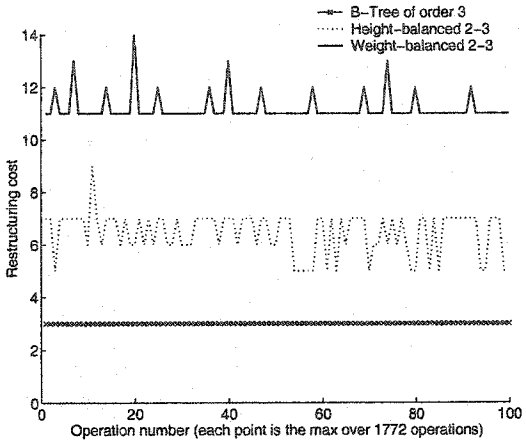

Figure 4.34: Restructuring costs for the B-tree of order 3, height-balanced 2-3, and weight-balanced 2-3 algorithms. Experiment consists of 177, 147 pathological deletions from an initial tree of the same size. Each point is the maximum value over 1, 772 consecutive operations.

maximum degree 4 for this experiment because it was shown that 4 is the best degree for 50% ADD(u) and 50% DELETE(u) operations [53].

We present the results of a simulation where each point represents either the average or maximum value over an experiment of 100, 000 weighted re-key operations starting with an initial tree of 350, 000 members. The x-axis for our graphs shows the percentage of ADD(u) operations used in the weighted re-key sequence, and the y-axis shows the average or maximum cost over the different sequences. Note that we chose our initial tree size to be somewhere between 4^9 and 4^{10} so that we do not force the tree to change height over the course of the experiment. Recall that the B-tree algorithm performs poorly when increasing the height of its tree. This experiment is aimed at understanding the performance over a sequence with a mixture of re-key operations, so we do not want to be misled by these boundary cases which were studied via the experiments using all ADD(u) and all DELETE(u) operations.

Figure 4.35 shows the average communication cost values for each of the different weighted re-key sequences. We note that all of the algorithms have very good perfor-

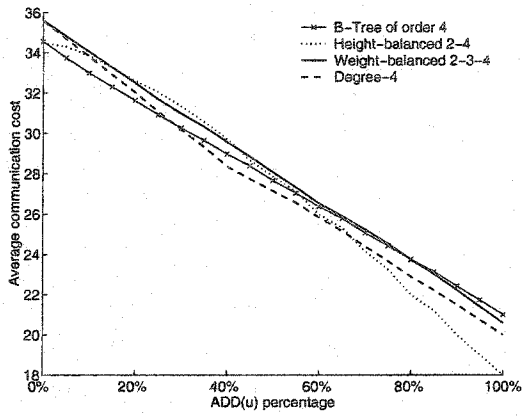

Figure 4.35: Average communication cost for various weighted re-key sequences, where each point in the graph corresponds to the average cost over an experiment consisting of 100,000 weighted re-key operations. The initial tree has 350,000 members.

mance, and that the order of performance varies with different weights. We omit the tree structure and restructuring cost graphs for this experiment because the tree structure costs are almost identical to the communication costs (meaning that the average restructuring costs are very low). Another thing to note is the downward slope of the lines. This makes sense because of the asymmetry of re-key operations (ADD(u) operations are cheaper).

The maximum values for this experiment represent the worst-case cost for each weighted sequence. The maximum tree structure costs are shown in Figure 4.36, the maximum communication costs are shown in Figure 4.37, and the maximum restructuring costs are shown in Figure 4.38. We point out a couple of interesting things. First, the maximum tree structure costs for the degree-k and weight-balanced algorithms are identical, and they are almost identical to the tree structure cost for the B-tree algorithm. Second, the weight-balanced algorithm achieves a decent trade-off between tree structure and restructuring costs. The poor restructuring costs seen for weight-balanced algorithm for the all DELETE(u) sequences do not occur with the mixture of re-key operations. Third, the height-balanced algorithm is hurt by its high tree structure costs, while the B-tree algorithm is hurt by its high restructuring costs. It is interesting to note that the trends seen for the all ADD(u) and all DELETE(u) sequences affect the height-balanced and B-tree algorithms, but not

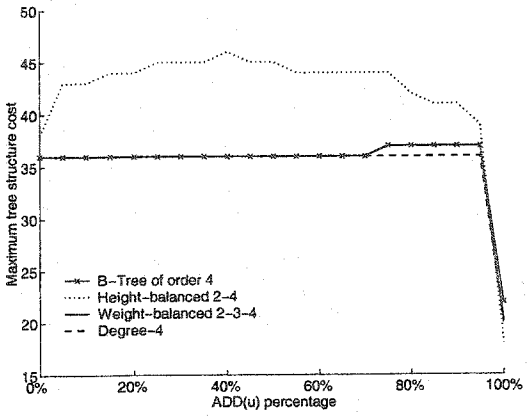

Figure 4.36: Maximum tree structure cost for various weighted re-key sequences, where each point in the graph corresponds to the maximum cost over an experiment consisting of 100,000 weighted re-key operations. The initial tree has 350,000 members.

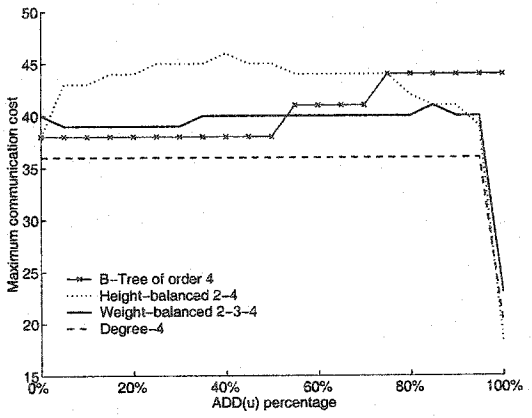

Figure 4.37: Maximum communication cost for various weighted re-key sequences, where each point in the graph corresponds to the maximum cost over an experiment consisting of 100,000 weighted re-key operations. The initial tree has 350,000 members.

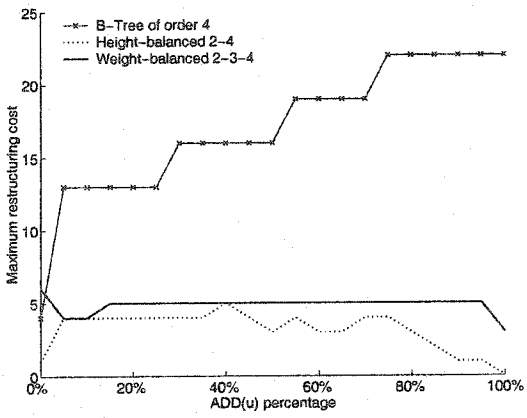

Figure 4.38: Maximum restructuring costs for various weighted re-key sequences, where each point in the graph corresponds to the maximum cost over an experiment consisting of 100,000 weighted re-key operations. The initial tree has 350,000 members.

the weight-balanced algorithm. The reason is that it only takes a few deletions to make the height-balanced algorithm increase its height. Similarly, with a few additions the B-tree algorithm will soon be inserting into full subtrees. The weight-balanced algorithm has a few bad situations where the restructuring rules would propagate in the tree, but with the mixture of re-key operations these configurations are unlikely. The above discussion seems to imply that the weight-balanced algorithm is the most robust among our three new algorithms (its performance is not as sensitive to small variations in the re-key sequence).

4.4.4 Discussion of the Simulation Results

It is interesting to note that there is a big difference when considering average values as opposed to maximum values. All of the algorithm have very similar performance when considering average values. However, the maximum tree structure costs for the height-balanced algorithm and the maximum restructuring costs for the B-tree and weight-balanced algorithms are sometimes quite high, which leads to greater differences in performance.

Looking at the results of our experiments, we attempt to answer each of the questions we posed at the beginning of this section.

1. *Is it important to maintain tree structures?* For most of our simulations, the simple degree-k key trees perform very well. The only time that it seems advantageous to maintain balance using our algorithms is for pathological cases that make the degree-k key trees highly unbalanced. We also note that for sequences with mostly ADD(u) operations, it is best not to perform any restructuring because simply choosing where to insert a new member already helps to maintain balance.

2. *Which of our algorithms achieves the best trade-off between tree structure and restructuring cost?* It depends on the sequence type. The B-tree algorithm has a good trade-off except for sequences with mostly ADD(u) operations. The height-balanced and weight-balanced algorithms have a good trade-off except for sequences with mostly DELETE(u) operations. The weight-balanced algorithm is the most robust, and has the best trade-off for sequences with a mix of ADD(u) and DELETE(u) operations.

One interesting question that we did not study is the effect of the maximum key tree degree on the performance of our algorithms. Instead, we focused on comparing the different algorithms using the same maximum degrees. We briefly consider the effects of the maximum degree choice in Table 4.3. Each entry in the table represents the average communication cost for a sequence that builds an initial group of 350,000 members, runs 100,000 weighted re-key operations, then removes the remaining members of the group. We can see that the performance of each algorithm improves as we increase the maximum degree until we reach a threshold. Then the performance decreases as the maximum degree increases. For example, we see that the degree-k algorithm performs the best when $k = 4$. The reason for this effect is the asymmetry between ADD(u) and DELETE(u) operations. In particular, the best tree for a sequence of all ADD(u) operations is a star, and 3 is the best degree for a sequence of all DELETE(u) operations.

It is interesting to note that different algorithms have the best performance for different weighted re-key sequences. In particular, for 0% ADD(u) operations, the height-balanced 2-4 algorithm performs the best (average cost of 24.99). For 25% ADD(u) operations, the height-balanced 2-5 algorithm performs the best (average cost of 25.20). For 50% ADD(u) operations, the B-tree of order 6 performs the best (average cost of 25.40). For 75% ADD(u) operations, the B-tree of order 5 performs the best (average cost of 25.22). Finally, for 100% ADD(u) operations, the B-tree of order 6 performs the best (average cost of 24.89). We leave further investigation on the performance implications of choosing maximum degrees to future work.

4.5 Conclusions and Future Work

We have designed three new online algorithms for the dynamic maintenance of balanced key trees. The algorithms used local restructuring rules and all achieved a decent trade-off between tree structure and restructuring costs. Our results so far are encouraging, but there is room for improvement. We list some of the more interesting directions for future work.

1. Continue our work on deriving worst-case communication cost bounds. This will require us to figure out how to analyze the restructuring costs for our algorithms.

Table 4.3: Comparison of the key tree algorithms using different maximum degrees. Each entry represents the average communication cost over a sequence that builds an initial group of 350,000 members, runs 100,000 weighted re-key operations, then removes the remaining members. The boldfaced entries represent the best performance for the given weights.

Algorithm	**ADD(u) Percentage**				
	0%	25%	50%	75%	100%
Degree-2	32.93	33.44	33.77	33.77	32.58
Degree-3	26.64	26.76	26.88	27.00	27.25
Degree-4	25.41	25.46	25.51	25.62	25.62
Degree-5	25.43	25.55	25.64	25.73	25.64
Degree-6	26.06	26.10	26.10	26.06	26.10
B-Tree of order 3	27.21	27.34	27.37	27.21	27.01
B-Tree of order 4	25.07	25.40	25.69	25.83	25.01
B-Tree of order 5	25.30	25.21	25.55	**25.22**	24.98
B-Tree of order 6	25.02	25.23	**25.40**	25.53	**24.89**
B-Tree of order 7	26.50	26.61	26.56	26.37	26.07
Height-balanced 2-2	33.33	33.62	33.81	34.00	33.94
Height-balanced 2-3	26.67	27.26	27.68	27.81	27.36
Height-balanced 2-4	**24.99**	25.62	25.84	26.12	25.96
Height-balanced 2-5	25.10	**25.20**	25.42	25.50	25.74
Height-balanced 2-6	25.12	25.36	25.96	26.25	26.47
Weight-balanced 2-3	27.61	27.72	27.80	28.07	28.22
Weight-balanced 2-3-4	26.22	26.31	26.46	26.61	26.74

2. Generalize our studies to consider arbitrary node degrees. In particular, we could try to derive an analysis for general B-trees of order t and height-balanced 2-k trees instead of special cases. We could try to come up with a general weight-balanced algorithm. And we could also study the general versions of our algorithms via simulations.

3. Figure out what type of re-key sequences are likely to appear in practice. This would help us perform more realistic simulations and could lead to the development of more efficient algorithms.

4. Look at security for group communication in different contexts. Our study focused on group communications over the Internet. It might be interesting to study group security in sensor networks and wireless ad-hoc networks, which are becoming more popular. There are many fundamental differences between secure key distribution for wired and wireless networks, and many of the assumptions that are made for the wired scenario are no longer valid. For example, the physical location of group members can no longer be ignored. Lazos and Poovendran found that the physical location of group members plays a major role in energy-efficient key distribution for wireless networks [33]. Another example difference between wired and wireless networks is that routing is often the job of the group members in wireless networks. Since the group members have to establish routing paths and do not know for certain the physical locations of group members, attacks are much harder to defend against. One such attack (called the wormhole attack) creates a tunnel between colluding members so that routing information that was meant for members at one end of the tunnel are replayed at the other end, allowing a few bad members to completely control routing in the network. Since it is difficult to verify the physical location of any given member, this wormhole attack is harder to detect in the wireless case. Other differences in the valid assumptions for the wired and wireless scenarios lead to numerous research challenges and open problems for the wireless case.

BIBLIOGRAPHY

[1] RealNetworks Inc., http://www.realnetworks.com.

[2] Replay TV, http://www.digitalnetworksna.com/replaytv/default.asp.

[3] TiVo Inc., http://www.tivo.com.

[4] C. C. Aggarwal, J. L. Wolf, and P. S. Yu. On optimal piggyback merging policies for video-on-demand systems. In *Proceedings of the ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '96)*, pages 200–209, 1996.

[5] S. Banerjee and B. Bhattacharjee. Scalable secure group communication over IP multicast. *IEEE Journal on Selected Areas in Communications (JSAC), Special Issue on Network Support for Group Communication*, 20(8):1511–1527, 2002.

[6] A. Bar-Noy and R. E. Ladner. Competitive on-line stream merging algorithms for media-on-demand. *Journal of Algorithms*, 48(1):59–90, 2003.

[7] A. Bar-Noy and R. E. Ladner. Efficient algorithms for optimal stream merging for media-on-demand. *SIAM Journal on Computing*, 33(5):1011–1034, 2004.

[8] R. Bayer and E.M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[9] B. Briscoe. MARKS: Zero side-effect multicast key management using arbitrarily revealed key sequences. In *Proceedings of the First International Workshop on Networked Group Communication*, pages 301–320, 1999.

[10] Y. Cai and K. A. Hua. Sharing multicast videos using patching streams. *Multimedia Tools and Applications*, 22(2):125–146, 2003.

[11] Y. Cai, K. A. Hua, and K. Vu. Optimizing patching performance. In *Proceedings of the Multimedia Computing and Networking Conference (MMCN '99)*, pages 204–215, 1999.

[12] R. Canetti, T. Malkin, and K. Nissim. Efficient communication-storage tradeoffs for multicast encryption. In *Advances in Cryptology - EUROCRYPT '99, Proceedings of the International Conference on the Theory and Application of Cryptographic Techniques*, pages 459–474, 1999.

[13] S. W. Carter and D. D. E. Long. Improving bandwidth efficiency of video-on-demand servers. *Computer Networks*, 31(1-2):111–123, 1999.

[14] W. T. Chan, T. W. Lam, H. F. Ting, and P. W. H. Wong. A 5-competitive on-line scheduler for merging video streams. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS) - Workshop on Scheduling and Telecommunications*, pages 2165–2172, 2001.

[15] W. T. Chan, T. W. Lam, H. F. Ting, and P. W. H. Wong. Competitive analysis of on-line stream merging algorithms. In *Proceedings of the Twenty-Seventh Annual International Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 188–200, 2002.

[16] M. Chesire, A. Wolman, G. Voelker, and H. Levy. Measurement and analysis of a streaming-media workload. In *USENIX Symposium on Internet Technologies and Systems (USITS)*, pages 1–12, 2001.

[17] E. G. Coffman Jr., P. R. Jelenković, and P. Momčilović. The dyadic stream merging algorithm. *Journal of Algorithms*, 43(1):120–137, 2002.

[18] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company, 1990.

[19] C. Cunha, A. Bestavros, and M. Crovella. Characteristics of WWW client-based traces. Technical Report TR-95-010, Boston University Department of Computer Science, 1995.

[20] S. E. Deering and D. R. Cheriton. Multicast routing in datagram internetworks and extended LANs. *ACM Transactions on Computer Systems*, 8(2):85–110, 1990.

[21] D. L. Eager and M. K. Vernon. Dynamic skyscraper broadcasts for video-on-demand. In *Proceedings of the 4th International Workshop on Advances in Multimedia Information Systems (MIS '98)*, pages 18–32, 1998.

[22] D. L. Eager, M. K. Vernon, and J. Zahorjan. Optimal and efficient merging schedules for video-on-demand servers. In *Proceedings of the 7th ACM International Multimedia Conference, (MULTIMEDIA '99)*, pages 199–202, 1999.

[23] D. L. Eager, M. K. Vernon, and J. Zahorjan. Bandwidth skimming: a technique for cost-effective video-on-demand. In *Proceedings of the Multimedia Computing and Networking Conference (MMCN'00)*, pages 206–215, 2000.

[24] D. L. Eager, M. K. Vernon, and J. Zahorjan. Minimizing bandwidth requirements for on-demand data delivery. *IEEE Transactions on Knowledge and Data Engineering*, 13(5):742–757, 2001.

[25] A. Fiat and M. Naor. Broadcast encryption. In *Advances in Cryptology - CRYPTO '93, Proceedings of the 13th Annual International Cryptology Conference*, pages 480–491, 1993.

[26] A. O. Freier, P. Karlton, and P. Kocher. The SSL protocol version 3.0. IETF Internet-draft, 1996.

[27] L. Gao and D. F. Towsley. Supplying instantaneous video-on-demand services using controlled multicast. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems (ICMCS '99)*, pages 117–121, 1999.

[28] L. Golubchik, J. C. S. Lui, and R. R. Muntz. Reducing I/O demand in video-on-demand storage servers. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '95)*, pages 25–36, 1995.

[29] L. Golubchik, J. C. S. Lui, and R. R. Muntz. Adaptive piggybacking: a novel technique for data sharing in video-on-demand storage servers. *ACM Multimedia Systems Journal*, 4(3):140–155, 1996.

[30] K. P. Gummadi, R. J. Dunn, S. Saroiu, S. D. Gribble, H. M. Levy, and J. Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 314–329, 2003.

[31] K. A. Hua and S. Sheu. Skyscraper broadcasting: a new broadcasting scheme for metropolitan video-on-demand systems. In *Proceedings of the ACM SIGCOMM '97 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 89–100, 1997.

[32] S. W. Lau, J. C. S. Lui, and L. Golubchik. Merging video streams in a multimedia storage server: complexity and heuristics. *ACM Multimedia Systems Journal*, 6(1):29–42, 1998.

[33] L. Lazos and R.Poovendran. Cross-layer design for energy-efficient secure multicast communications in ad hoc networks. In *IEEE International Conference on Communications (ICC)*, 2004.

[34] X. Li, Y. Yang, M. G. Gouda, and S. S. Lam. Batch rekeying for secure group communications. In *Proceedings of the tenth international World Wide Web Conference*, pages 525–534, 2001.

[35] M. Luby and J. Staddon. Combinatorial bounds for broadcast encryption. In *Advances*

in Cryptology - EUROCRYPT '98, Proceedings of the International Conference on the Theory and Application of Cryptographic Techniques, pages 512–526, 1998.

[36] S. Mittra. Iolus: A framework for scalable secure multicasting. In *Proceedings of the ACM SIGCOMM '97 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 277–288, 1997.

[37] M. Moyer, J. Rao, and P. Rohatgi. Maintaining balanced key trees for secure multicast. Internet draft, 1999.

[38] M. Moyer, J. Rao, and P. Rohatgi. A survey of security issues in multicast communications. *IEEE Network Magazine*, 13(6):12–23, 1999.

[39] J. Pâris. A simple low-bandwidth broadcasting protocol for video-on-demand. In *Proceedings of the 8th International Conference on Computer Communications and Networks (ICCCN '99)*, pages 118–123, 1999.

[40] J. Pâris, S. W. Carter, and D. D. E. Long. A hybrid broadcasting protocol for video on demand. In *Proceedings of the Conference on Multimedia Computing and Networking (MMCN '99)*, pages 317–326, 1999.

[41] J. Pâris and D. D. E. Long. Limiting the receiving bandwidth of broadcasting protocols for video-on-demand. In *Proceedings of the Euromedia Conference*, pages 107–111, 2000.

[42] P. Parnes, K. Synnes, and D. Schefstrom. Lightweight application level multicast tunneling using mtunnel. *Journal of Computer Communication*, 21:1295–1301, 1998.

[43] R. Poovendran and J. S. Baras. An information-theoretic approach for design and analysis of rooted-tree-based multicast key management schemes. *IEEE Transactions on Information Theory*, 47(7):2824–2834, 2001.

[44] R. Rejaie, H. Yu, M. Handley, and D. Estrin. Multimedia proxy caching mechanism for quality adaptive streaming applications in the Internet. In *Proceedings of the IEEE INFOCOM Conference on Computer Communications*, pages 980–989, 2000.

[45] O. Rodeh, K. P. Birman, and D. Dolev. Using AVL trees for fault tolerant group key management. *International Journal on Information Security*, 1(2):84–99, 2001.

[46] S. Sen, L. Gao, J. Rexford, and D. Towsley. Optimal patching schemes for efficient multimedia streaming. In *Proceedings of the 9th IEEE International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV '99)*, pages 265–277, 1999.

[47] S. Setia, S. Koussih, S. Jajodia, and E. Harder. Kronos: A scalable group re-keying approach for secure multicast. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 215–228, 2000.

[48] C. Shields and J. J. Garcia-Luna-Aceves. KHIP – a scalable protocol for secure multicast routing. In *Proceedings of the ACM SIGCOMM '99 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 53–64, 1999.

[49] J. Snoeyink, S. Suri, and G. Varghese. A lower bound for multicast key distribution. In *Proceedings of the IEEE INFOCOM 2001 Conference on Computer Communications*, pages 422–431, 2001.

[50] M. Steiner, G. Tsudik, and M. Waidner. Cliques: a new approach to group key agreement. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, pages 380–387, 1998.

[51] S. Viswanathan and T. Imielinski. Metropolitan area video-on-demand service using pyramid broadcasting. *ACM Multimedia Systems Journal*, 4(4):197–208, 1996.

[52] D. M. Wallner, E. J. Harder, and R. C. Agee. Key management for multicast: issues and architectures. IETF Informational RFC, 1998.

[53] C. K. Wong, M. G. Gouda, and S. S. Lam. Secure group communications using key graphs. *IEEE/ACM Transactions on Networking*, 8(1):16–30, 2000.

[54] C. K. Wong and S. S. Lam. Keystone: a group key management service. In *Proceedings of the International Conference on Telecommunications*, 2000.

[55] S. Zhu, S. Setia, and S. Jajodia. Performance optimizations for group key management schemes. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 163–173, 2003.

VITA

Justin Goshi was born and raised in Honolulu, Hawaii. He earned a Bachelor of Science in Computer Science from the University of Hawaii and a Master of Science in Computer Science from the University of Washington. In 2004 he earned a Doctor of Philosophy in Computer Science at the University of Washington.