Scheduling Protocols for Media-on-Demand Systems

Tammy VanDeGrift

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

Doctor of Philosophy

University of Washington

2005

Program Authorized to Offer Degree: Computer Science and Engineering

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

University of Washington

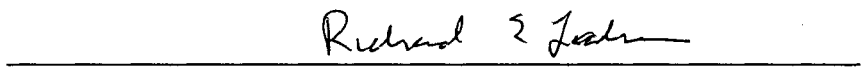
Graduate School

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

Tammy VanDeGrift

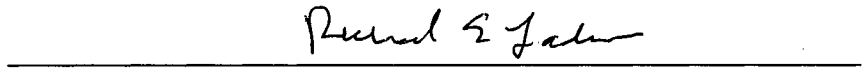
and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Chair of Supervisory Committee:

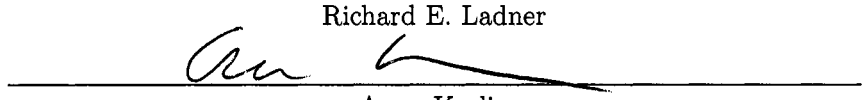


Richard E. Ladner

Reading Committee:



Richard E. Ladner



Anna Karlin

John Zahorjan

Date: May 20, 2005

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 Proquest 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 Tammy VanDeThift

Date May 20, 2005

University of Washington

Abstract

Scheduling Protocols for Media-on-Demand Systems

by Tammy VanDeGrift

Chair of the Supervisory Committee:

Professor Richard E. Ladner
Computer Science & Engineering

Today's abundance of personal electronic devices and the growth of communications networks gives rise to a new set of applications. One such application allows clients to access high-bandwidth streaming media at times of their choosing. The focus of this dissertation is on the media-on-demand application for use in high-bandwidth communications networks. A media-on-demand server stores and distributes one or more media files to clients who wish to view/play them. Because server bandwidth can be quickly consumed when delivering the full media file to each client under high loads, protocols for delivering streaming media try to reduce the server bandwidth used to satisfy client requests. Our work focuses on the stream merging technique, where clients can receive two streams simultaneously and buffer data for future parts of the media file. When clients have the same data as other clients in the system, we merge the clients into a single group and eliminate unnecessary streams. This dissertation introduces, analyzes, and empirically compares algorithms for three media-on-demand models. The first model, which we call media-on-demand, assumes the media file requested is of finite length and clients always consume the media file from its beginning. The second model, media-on-demand with time-shifting, supports access to any portion of a live broadcasted stream. Our contributions include devising an algorithm for media-on-demand systems under high client loads, modeling media-on-demand with time-shifting as a rectilinear tree, finding the complexity class for the optimal offline solution

to media-on-demand with time-shifting, and introducing an online algorithm for the time-shifting case. Finally, we introduce a network cost model where we try to optimize the network bandwidth instead of the server bandwidth. We describe an algorithm to compute the network cost given a stream merging schedule and introduce an online stream merging algorithm for the network cost model.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

GLOSSARY

α: The parameter in the dyadic algorithm that specifies sizes of dyadic intervals.

β: The parameter in the dyadic algorithm that specifies when to start new root streams.

C_O: An ordering of a client set C such that for each client in the ordering, its possible merge targets precede it in the ordering.

CLIENT: Requests the media at a certain time. In the time-shifting model, the client also requests a first segment.

CLIENT RECEIVE BANDWIDTH: The total bandwidth a client can receive per unit time.

CLIENT INTERARRIVAL RATE: The average number of media units separating client arrivals.

COMPLETE MERGE SCHEDULE: A merge schedule in the time-shifting case where every client remains in the system until after it merges to the live, continuous broadcast.

CONTINUATION: A continuation is a horizontal stream with more than one client arrival along the stream in a rectilinear merge tree.

CONTINUOUS HORIZONTAL LINE: A continuous horizontal line through client arrivals and junctions in a rectilinear merge tree.

CONTINUOUS VERTICAL LINE: A continuous vertical line through client arrivals and junctions in a rectilinear merge tree.

CONTINUOUS TIME RESPONSE: The model where clients are served the beginning of the media file immediately upon arrival to the system.

CORNER: A corner connects a horizontal stream from the right with a downward vertical stream in a rectilinear merge tree.

CROSSOVER: A horizontal stream that intersects a vertical stream without a junction marking, so the two streams do not interact with each other in a rectilinear merge tree.

DISCRETE HIGH DEMAND: The case of popular media where there is at least one client request per media segment

DISCRETE TIME RESPONSE: The model where time is broken into discrete slices and a client arriving before a time slice are served at that time slice.

DISTRIBUTION TREE: The tree corresponding to the network infrastructure of the server and clients. The server is at the root of the distribution tree and all links in the network have non-negative costs.

ELBOW PROPERTY: All junctions have horizontal streams as least as long as the vertical streams in a rectilinear merge tree.

F(C): The first segment requested by client c.

JUNCTION: Represented as a triangle, a junction is the merge point of the vertical stream from above to the horizontal stream to the right in a rectilinear merge tree.

L: The media length in number of segments. In the time-shifting model, the media length is infinity.

MERGE COST: The total server bandwidth for all clients excluding the bandwidth associated with root streams.

MERGE GROUP: The set of clients eventually merging to a common root stream. In the time-shifting case, all clients are in the same merge group since they eventually merge to the live stream.

MERGE ONCE: The algorithmic approach for the live broadcast media-on-demand case where each client simply gets a dedicated stream until it can merge directly to the live stream.

MERGE TARGET: The client to which a client c first merges in the system. In an ordered standard merge tree, the merge target of c is c's parent.

MERGE SCHEDULE: The schedule of client mergers in the system, which may be represented as a merge figure diagram, a merge tree, or a rectilinear merge tree.

MERGE SET: For a client X in a transformed rectilinear merge tree, the merge set is the subset of clients that eventually merge to X's stream.

MINIMUM CONNECTION: For the reduction proof a minimum connection is a set of rectilinear merge trees with roots at filled points such that the length of all the trees is minimal.

NORMALIZED RECTILINEAR MERGE TREE: A rectilinear merge tree that has undergone transformations such that every continuous horizontal line contains at least one client arrival and every junction has a continuous vertical line ending at the top with a client arrival.

OPTIMAL STREAM MERGING WITH TIME-SHIFTING PROBLEM: Given a client arrival set C where each client has an arrival time and requested first segment, what is the

optimal merge schedule to minimize total server bandwidth?

ORDERED STANDARD MERGE TREE: A rooted tree where each node represents a client such that each client merges to its parent client in the tree.

PARITY: A value of 0 or 1 assigned to a tile based on connections used in the reduction proof.

POSSIBLE TARGET CLIENT SET: For a client X the set of clients to which X can merge in a rectilinear merge tree.

PREMATURE START: A dashed horizontal stream, starting before a client arrival, in a rectilinear merge tree.

RECEIVE-ALL: The media-on-demand model where clients can receive all active data streams of their merge targets simultaneously.

RECEIVE-TWO: The media-on-demand model where clients can receive two data streams simultaneously.

RECTILINEAR MERGE TREE: A tree embedded on a rectilinear grid where all edges represent streams and navigation to the root of the tree is via downward and leftward edges.

ROOT STREAM: A media stream serving the entire length of a media file in the standard stream merging model. The live, continuously broadcasted stream in the time-shifted broadcast model.

SERVER: A machine or collection of machines that stores the media file and makes scheduling decisions to deliver the media to clients.

$T(C)$: The arrival time of client c.

TARGET CLIENT: In a transformed rectilinear merge tree, the target client for X is the first client to which X merges. This is the same as the merge target.

TARGET STREAM: The first stream to which a client merges.

TIME-SHIFTING MODEL: The media-on-demand model where clients request any part of the previous or currently broadcasted live stream.

TOTAL COST: The total server bandwidth for all clients including the root streams.

TOTAL NETWORK COST: The total cost of delivering media streams to a client set C where the cost of a single link in the distribution tree is the number of media segments sent along that link multiplied by the cost of the link.

ZERO LINE: In the rectilinear grid embedding, the line containing all client arrivals that request segment 0 as the first segment.

ACKNOWLEDGMENTS

This dissertation is made possible with the help, training, encouragement, and constructive criticism offered by many colleagues, professors, and mentors. First, let me thank the Department of Computer Science & Engineering for providing so many opportunities to grow as a student, scholar, and teacher.

I especially want to thank the professors who have guided my research. First, I want to thank Richard E. Ladner for his great advising, his patience, and his valuable time. Without his guidance, this dissertation would not be finished. Second, I want to thank the professors who helped train me as a scholar: my reading committee (including Anna Karlin and John Zahorjan), Richard Anderson, Jennifer Turns, Larry Ruzzo, Sally Fincher (University of Kent, Canterbury), Marian Petre (The Open University), Josh Tenenberg (University of Washington, Tacoma), Amotz Bar-Noy (Brooklyn College), and Tami Tamir (Interdisciplinary Center Herzliya). Finally, I appreciate the help of fellow graduate student Justin Goshi for getting me started on the research presented in this dissertation.

I want to thank those who have contributed to my development as a teacher, especially Hal Perkins for taking me on as a co-instructor and the attendees of the computer science education seminars. I thank my former professors at Gustavus Adolphus College who modeled excellence in teaching.

Finally, I need to thank my family and friends who provided support throughout my graduate school years. I am especially grateful for my husband, Tom, who was a constant source of strength through the good times and the rough times. I appreciate the many friends and colleagues that I met during my graduate school years, especially my research collaborators and my wonderful officemates.

DEDICATION

To my first teachers:

My parents, Mark and Eileen, for their constant love, patience, and encouragement throughout my life. As educators, they inspired me to teach others. As people, they inspired me to care about the people I teach.

Chapter 1

INTRODUCTION TO MEDIA-ON-DEMAND SYSTEMS

The recent growth of networked systems supports a broad set of new applications. Along with those applications comes a set of challenges, which can be addressed by adapting solutions for related problems and devising completely new solutions. An example application that can be supported by large communications networks is on-demand media distribution. We refer to this application as media-on-demand, which supports access and delivery of large media files, such as movies, and even continuous streams, such as television and radio broadcasts. Popular media files can pose challenges to the networked system, especially the server storing these large files. Under heavy traffic loads, the resources of the server, such as its outgoing transmission bandwidth, could be fully consumed. We provide solutions in this dissertation for reducing the load on a server by taking advantage of multicast and client buffers.

Media-on-demand applications provide new models for the entertainment, news, and education domains. A customer of a movie-on-demand service could access a movie anytime and anywhere he/she has a connection to the server. A customer could also access any part of the previous live broadcast for television or radio. Media-on-demand systems also support educational opportunities, especially distance learning environments where students have access to lecture videos.

Our work seeks to reduce the bandwidth required by a server to distribute media files to clients. In our model we assume that a single server stores the media file. One approach for reducing the load on a central server is to replicate the data and have multiple servers distributed throughout a network storing copies of the data [26, 76, 14, 41]. A similar strategy to reduce server load is to cache data at nodes throughout the network [62, 3, 71, 75].

Instead of managing multiple servers, we assume a single server in our problem definition. In our approach we assume the media data can be multicast from the server to a set of one or more clients [28, 33]. The transmission is one-to-many and in a single direction from the server to the set of clients. Multicast is similar to broadcast in that the transmission is one-to-many, but multicast only delivers data to specific addresses of group members [57]. This is especially important when considering a pay-per-view video service, where only paying customers should receive data for the video.

The objective of the algorithms presented in this dissertation is to reduce the total bandwidth a server needs to use to distribute a media file to a set of clients. We measure bandwidth as the total amount of media data delivered to a set of clients. A simple, yet not scalable, approach is to simply send each client the full media file. This is the approach used by most servers today. Our work seeks to reduce server bandwidth by distributing portions of the media file to groups of clients. The central thesis in our work is that analysis of optimal offline solutions for media-on-demand models provide insight into devising online algorithms.

1.1 Media Delivery

Several approaches have been proposed to deliver media efficiently. Our work uses the technique of stream merging [30, 32] to reduce server bandwidth. The stream merging technique assumes clients can receive at least two streams simultaneously and they can buffer the data they receive. Once they have received enough data from one stream, they can merge to another set of clients in the system. For example, let us take the example of two clients requesting the same two hour movie. Client a arrives at 9:00 PM and client b arrives at 9:05 PM. The server distributes the full movie to client a at the same rate as the playback rate. When client b arrives, the server distributes the first five minutes of the movie on one stream and has client b receive the stream for client a. At 9:10, the stream containing the first five minutes of the movie can be terminated, since client b gets the remainder of the movie from stream a. This process can continue hierarchically if there are more than two clients receiving data simultaneously. Stream merging reduces server

bandwidth by discontinuing the delivery of streams once they are no longer needed by clients in the system.

A media-on-demand system consists of three parts: a server, the clients, and the channels. First, a server controls how the content gets distributed to clients. The algorithms presented in this dissertation focus solely on server-side decisions. Second, clients request the media from the server and the system uses a collection of channels to transmit the media. In the problem formulation that we study here, we assume the set of requesting clients is not known ahead of time; instead, the server reacts to each client request in real time. In this scenario, the server must make online decisions about how to manage each new request.

1.1.1 Stream Merging

The algorithms described in this thesis use stream merging to reduce server bandwidth [30] [32]. We assume clients can listen to and buffer data from two channels simultaneously. In order for the client to receive data transmissions from two channels, the client receive bandwidth must be at least twice the media playback rate. Our work assumes that clients can receive two streams simultaneously, which we refer to as the *receive-two* model. In stream merging, clients buffer future data of the media from one channel while playing out data from a separate channel. In this manner, a client can buffer its own data while listening to a channel for a separate client or group of clients. Once a client receives enough data from its dedicated channel, it can merge with a set of clients already in the system, allowing the server to drop the dedicated channel.

A system showcasing stream merging capabilities is presented in Figure 1.1. Each line represents a channel distributing media to a client or set of clients. Additionally, each stream is delivered via multicast technology, so multiple clients can be recipients of the same stream.

Part (a) of Figure 1.1 shows the state of the client upon arrival to the system. The server grants a dedicated channel (the bottom line) to the client while scheduling the client to listen to the top channel. The data from the client's dedicated channel is buffered and

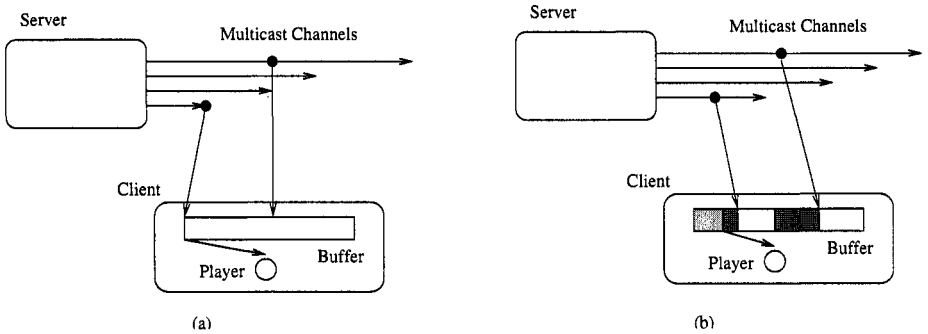


Figure 1.1: The mechanism for clients to receive two streams simultaneously. Part (a) shows the client upon arrival. Part (b) shows the client later in time in the system. The client has already buffered data from the top stream and is playing back data buffered from the bottom stream.

sent to the client's media player for immediate playback. The data from the top channel is stored in the client's buffer for later playback. Part (b) of the figure shows the state of the client after some time in the system. When the client receives enough data from its own channel, the server will discontinue the channel (the bottom line) and the client fully merges to the set of clients listening to the top channel. This process can be extended hierarchically in that clients can always listen to up to two channels simultaneously.

The stream merging model allows clients to merge when a client has buffered the required data from its own stream. Figure 1.2 shows how clients can merge in the system. Clients arrive at times 0, 1, 2, 3, and 5. We refer to the clients as their arrival times. When client 0 arrives, there are no streams in the system, so the server grants client 0 its own stream. When client 1 arrives, the server grants a new stream to send the first media segment to client 1. At the same time, client 1 buffers data from its *merge target*, the client to which client 1 will merge. Client 1 buffers the second segment of the media from client 0's stream while buffering and playing the first segment of the media from its own stream. After 1 time unit in the system, client 1 *merges* with client 0, so the dedicated stream to client 1 is terminated. This merge is illustrated with a dotted line in Figure 1.2. The figure also shows how the remaining clients can merge in the system. All the clients eventually merging to the same full-length root stream comprise the clients in a single *merge group*. We say the

initial client, like client 0 in Figure 1.2, is the *root client* and the stream serving this client is the *root stream.*

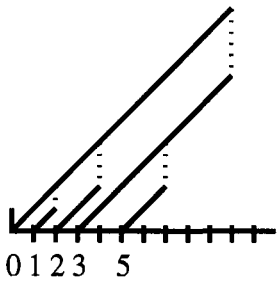


Figure 1.2: Illustrates the merging of clients 0, 1, 2, 3, and 5 in the system. The solid lines represent streams in the system and the dotted lines represent the merging of clients. The client buffers data from its merge target while getting data from its own stream.

If clients are served immediately when they request the media, there is no start-up delay. We define this model as *continuous time response* since clients can come at any time and the server must respond to them immediately. On the other hand, if the server batches client requests together, then clients will experience start-up delays. We term this *discrete time response* because clients will be served streams at the beginning of each time slice. The length of the time slice is determined by the length of a media segment. For example, if the media is a 2-hour movie and the movie is broken into 720 segments, then each segment is 10 seconds long. The worst-case start-up delay in this scenario is 10 seconds. Figure 1.2 shows an example for the discrete time response model, since the streams are initiated at the start of time slices.

1.1.2 Media-on-Demand Models

In this dissertation we introduce and analyze algorithms for stream merging for three models. The first model, which we refer to as media-on-demand, is the standard model assumed in much of the literature. This model assumes the media file has a fixed length L and clients always request the file from its beginning to its end. Furthermore, the client should be able to play/view the media file from beginning to end without interruption; therefore, the client must receive the data from the media file for time t before or at t units of time after the

client started receiving the media file. As stated before, we assume the receive-two model unless otherwise specified. Chapter 2 assumes the media-on-demand model.

The second model is a new model where the media no longer has a finite length. Instead, the media is a live, continuous broadcast which is available for clients to access. Radio and television broadcasts today do not support clients who wish to receive previous portions of the live broadcast. The media-on-demand model with time-shifting does support this arbitrary rewind capability. We assume clients request the time of the live broadcast it wishes to receive. This differs from the standard media-on-demand model where clients always request the beginning of the media file. With a live broadcast, we no longer have a beginning or end of the file. We assume the receive-two model where clients can receive two streams simultaneously. Chapters 3, 4, and 5 assume the media-on-demand model with time-shifting. We note that personal digital recording systems such as TiVo [72] provide similar time-shifting capabilities in that customers can watch previously broadcasted programs at times of their choosing. Customers who use services such as TiVo must plan to record the broadcasts ahead of time. The media-on-demand model with time-shifting allows clients to request previously broadcasted programs without specifying in advance to record those programs.

The third model, introduced in Chapter 6, extends the standard media-on-demand model to include the cost of the network links. Instead of saving server bandwidth, we try to optimize the cost of sending media data through the network to clients.

1.2 Measurement Metrics

Our main goal is to minimize the total server bandwidth used to serve client requests. Full-length streams are called *root streams*. We measure server bandwidth using two metrics. First, the *merge cost* is the total stream lengths of non-root streams where all streams eventually merge to a single root stream (independent of the actual length of the media). In Figure 1.2, the merge cost does not include the length of the stream started at time 0, but includes the length of streams for clients 1, 2, 3, and 5. The second metric, *total cost*, takes into consideration the actual media length. We may have several root streams in the

overall delivery schedule for a set of clients, since a client may request the media after which it can merge to an existing root stream. Instead, the server will grant a new root stream to this client. Total cost is the total length of all streams in the system including all root and non-root streams.

1.3 Stream Merging Algorithms

We describe two algorithms designed for stream merging in the media-on-demand model. Stream merging algorithms differ only in how they specify merge targets, the streams to which a client eventually merges. Bar-Noy *et al.* present a careful analysis of properties and performance of several stream merging algorithms in [6]. For this dissertation, we focus on two stream merging algorithms. The first, called earliest reachable merge target (ERMT), is an event-driven algorithm and performs well in simulation studies [30]. The second algorithm, called the dyadic algorithm, determines a client's complete schedule of merge targets at its arrival time [21]. The dyadic algorithm also performs well in simulations and is proved to be 3-competitive [15]. Since the dyadic algorithm is 3-competitive, the total cost for the merge schedule it produces for a client set C is no worse than three times the total cost of the optimal merge schedule for the same client set C. Therefore, we have a bound on how poorly the dyadic algorithm can schedule clients with respect to total server bandwidth cost.

1.3.1 Earliest Reachable Merge Target

The earliest reachable merge target algorithm proposed by Eager *et al.* is one the earliest algorithms using hierarchical stream merging [30, 32]. A new client or recently merged group of clients will merge to the most recently started stream that the client or group of clients can reach. The client or group of clients can reach a stream S if S does not terminate at or before the time of the potential merger. The algorithm operates on three types of events: arrivals, mergers, and stream terminations. An arrival simply schedules a new stream and merge target (if one is reachable) for the client. A merger action reschedules the merge target. A stream termination event removes the stream from the system.

The algorithm is event-driven and is best understood through an example. The progression of events is illustrated in Figures 1.3 and 1.4. Assume clients arrive at times 0, 3, 4, 5, 7 and 9 and the media length is 10 segments. We label clients according to increasing arrival time as a, b, c, d, e, and f. We label streams by uppercase letters corresponding to client sets. When client a arrives, there are no streams in the system, so a receives the full-length media object at the same rate as the playback rate. When b arrives, b's merge target is stream A since b will be able to merge to A at time 6 which is before its termination. When client c arrives, c's merge target is B since c can catch B before it terminates. At time 5, c merges to b so the merge target A is selected for clients b and c (stream labeled BC in Figure 1.3). When d arrives, it's merge target is set to the stream BC, since d can catch stream BC before its scheduled termination. At time 7, stream D terminates and d merges to the stream BC. When selecting a merge target for stream BCD, we calculate when the merge to stream A will happen. This merger would happen at time 10, which is not strictly less than A's termination time of 10. Thus, there is no merge target scheduled for BCD, so it continues as a full-length stream. Client e arrives at time 7 and is scheduled to merge to BCD. At time 9, f arrives. Since its merger to stream E would happen at E's termination time, we try merging to stream BCD. Client f cannot merge to stream BCD before its termination, so a full-length stream is issued for client f. The righthand figure of Figure 1.4 shows the future events of the system, since there are no arrivals after time 9.

The merge targets can dynamically change based on other events in the system. For example, client b is originally scheduled to merge to stream A in the example above, but when c is scheduled to merge to B the stream for BC can no longer reach stream A.

Another point to notice about the ERMT algorithm is that if clients arrive every $L/2$ time units, then each client will receive a full-length root stream. This is due to the fact that in order to set a merge target, the client must merge at a time before the target stream terminates. For example, assume the media has length 10 and clients arrive at times 0, 5, 10, 15, etc. When client at time 5 arrives, its merge time to the stream started at 0 is 10, which is exactly equal to the stream's termination time. Thus, ERMT would issue a full-length stream to the arrival at time 5. Similarly, all subsequent arrivals will receive a full-length stream. A better solution (in terms of total server bandwidth) would have the

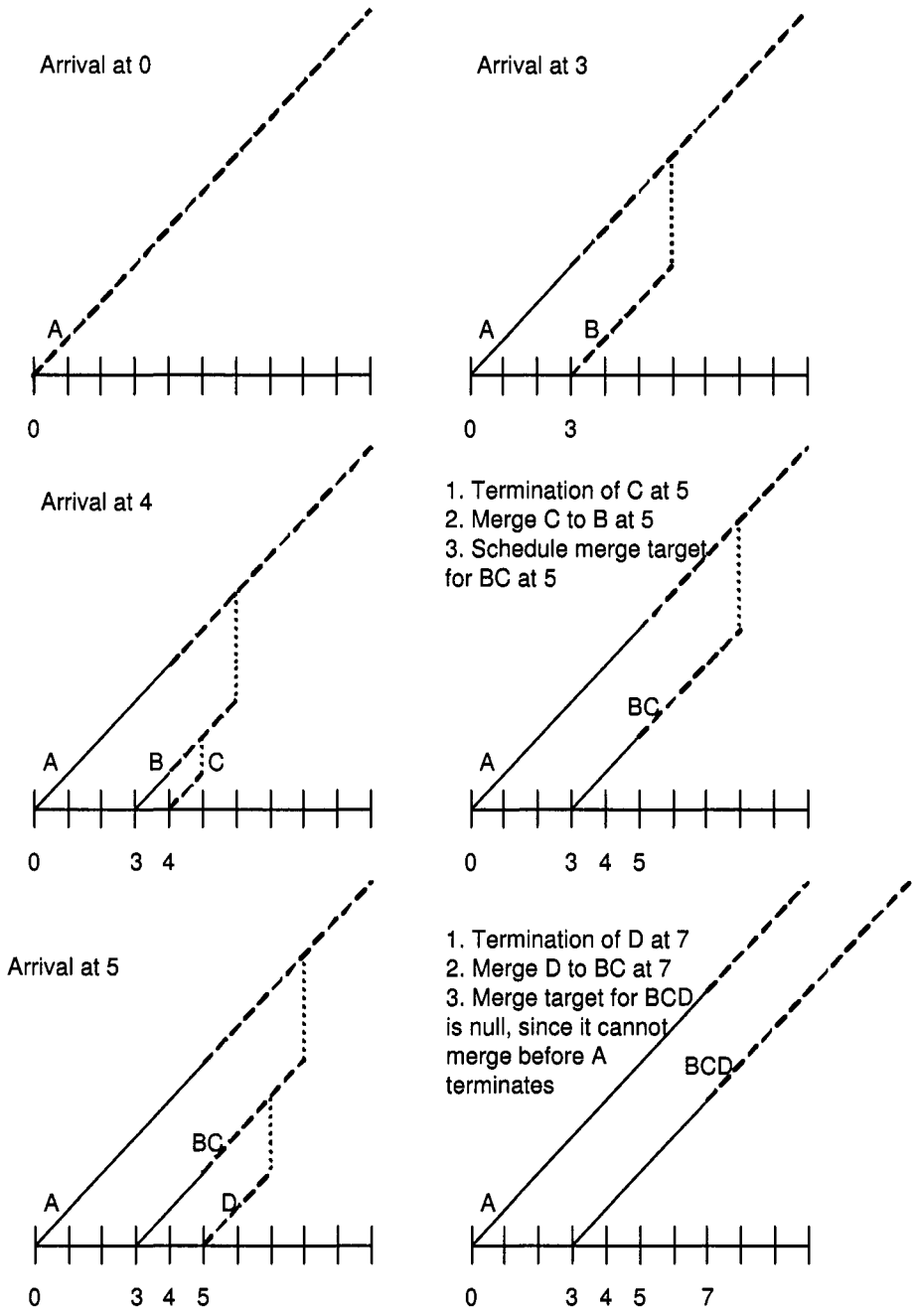


Figure 1.3: Illustrates the events up to time 7 for the ERMT algorithm, given client arrivals at times 0, 3, 4, 5, 7 and 9. The solid lines represent the actual streams delivered while the dashed lines show the future evolution of the system, given that no future clients arrive. The vertical dotted lines indicate merge events.

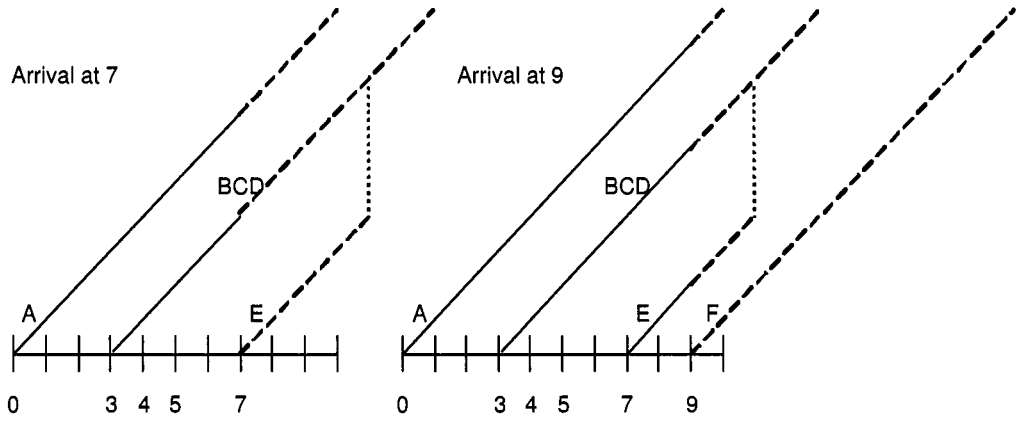


Figure 1.4: Illustrates the events after time 7 for the ERMT algorithm, given client arrivals at times 0, 3, 4, 5, 7 and 9. The solid lines represent the actual streams delivered while the dashed lines show the future evolution of the system, given that no future clients arrive. The vertical dotted lines indicate merge events.

arrival at 5 merge to the stream started at 0, the arrival at time 15 merge to the stream started at 10, etc. The better solution results in total server bandwidth that is 3/4 the cost of the bandwidth necessary for the ERMT solution.

1.3.2 The Dyadic Algorithm

The dyadic algorithm by Coffman *et al.* specifies how clients merge with other clients in the system by calculating time intervals indicating child-parent relationships among streams [21]. One can understand the algorithm from the perspective of knowing all the client arrivals to the system or from the perspective of handling clients in an online manner. We will start with the perspective of knowing all client arrivals to the system. Without loss of generality, let us assume the first client arrival is at time 0 and there are n clients that arrive before time $L/2$ where L is the media length. To find the set of clients that merge directly to the root stream that starts at time 0, we partition the interval $[0, L/2]$ into dyadic subintervals. The first subinterval is $[L/4, L/2]$, the next is $[L/8, L/4]$, and they continually get smaller by a factor of 2. The first client arrival after the beginning of each interval merges directly to the root stream. What about the other client arrivals within a dyadic

interval I? We divide I into dyadic intervals and the first arrivals after the beginning of each of these subintervals merges to the client that first arrived at the beginning of I. We continue this process recursively until all clients have a direct parent for merging.

The second perspective is to look at the online formulation of the dyadic algorithm. Let S be a stack with *push* and *pop* operations for pairs $[t_a, t_r)$. Each pair indicates a stream where t_a is the time of initiation and t_r is the time after which new streams will not be allowed to merge with it. We assume the first client request is initiated at time $t = 0$.

At request time t, *pop* the pairs $[t_a, t_r)$ from S until $t_r > t$. If S is empty, then *push* $[t, t + L/2)$ on to S (This is the start of a new *root stream.*) Otherwise, add the new stream to the stack by *pushing* $[t, t')$ where $t' = t_a + (t_r - t_a) \min\{(1/2)^{k-1} : (1/2)^k (t_r - t_a) < t - t_a\}$. The stream started at t will merge to the stream started at t_a.

We illustrate the execution of the online formulation of the dyadic algorithm with an example. Suppose $L = 20$ and we have the following clients arrive to the system: $\{a = 0, b = 3, c = 4, d = 6, e = 7, f = 8, g = 9\}$. At time 0, S is empty, so we push $[0, 10)$ onto S and start a root stream for a. At time 3, we add a new pair $[3, 5)$ to S and b eventually merges to a. When c arrives at time 4, we add a new pair $[4, 5)$ to S and c eventually merges to b which eventually merges to a. At time 6 when d arrives, we pop two pairs from S until S has at the top $[0, 10)$. We push $[6, 10)$ onto S and d eventually merges to a. At time 7, we push $[7, 7)$ on to S and e merges to d directly. At time 8, we pop $[7, 7)$ from S and push $[8, 8)$ onto S and f merges to d directly. Finally, at time 9, we pop $[8, 8)$ from S and push $[9, 10)$ onto S. Client g merges directly to d. Figure 1.5 shows the merge schedule created by the online version of the dyadic algorithm for these arrivals.

The online dyadic algorithm does not exactly mimic the first perspective of recursively finding dyadic intervals. The online version creates intervals with no length, such as $[7, 7)$ and $[8, 8)$ above. The recursive version of the dyadic algorithm always creates intervals with non-zero length and, in fact, produces a different schedule than than online version. Figure 1.6 shows the recursive version of the dyadic algorithm. On the top line, we find the first arrivals within each dyadic interval on the entire line between 0 and 10. Since there are two arrivals in $[3, 5)$ we subdivide this interval and arrival c merges to b. We subdivide the interval $[6, 10)$ into dyadic intervals and find that arrivals e and f are at the beginning of

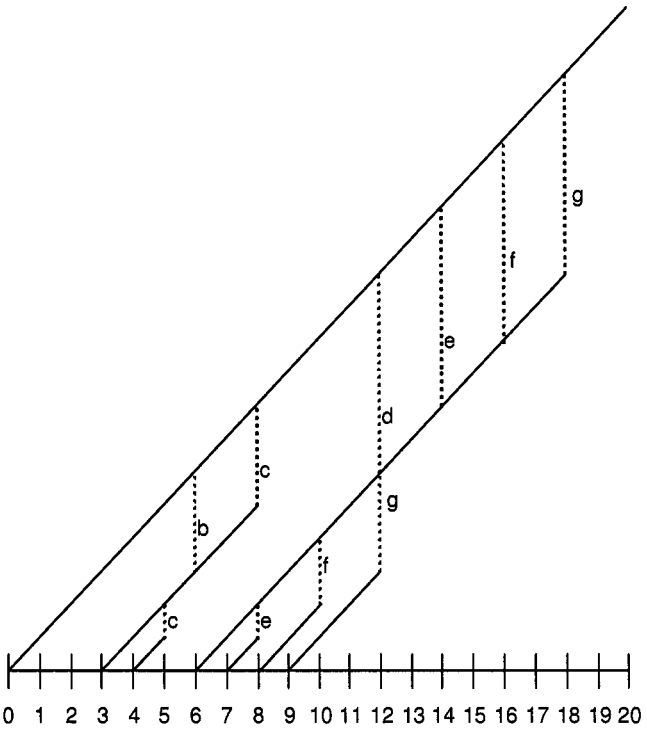


Figure 1.5: Merge schedule created by the online dyadic algorithm for client arrivals $a = 0$, $b = 3$, $c = 4$, $d = 6$, $e = 7$, $f = 8$, and $g = 9$. The vertical dotted lines indicate merges and the labels along the lines indicate the clients involved.

the intervals, so these merge directly to d. Then we subdivide the interval $[8, 10)$ and find g arrives at time 9. Thus, g merges directly to f. This last merge decision is different for the online version, since g merges directly to d. Figure 1.7 shows the merge schedule created by the recursive version of the dyadic algorithm. It differs from the online version shown in Figure 1.5 for the stream initiated for client $g = 9$. In the online version, the interval created starting at 8 has no length, so g merges to d instead. Throughout this dissertation, we use the online version of the dyadic algorithm for presentation purposes and as the implemented version for empirical comparisons.

The dyadic algorithm performs well with respect to optimal, with simulations yielding an 8% increase in total cost over the optimal offline algorithm [21].

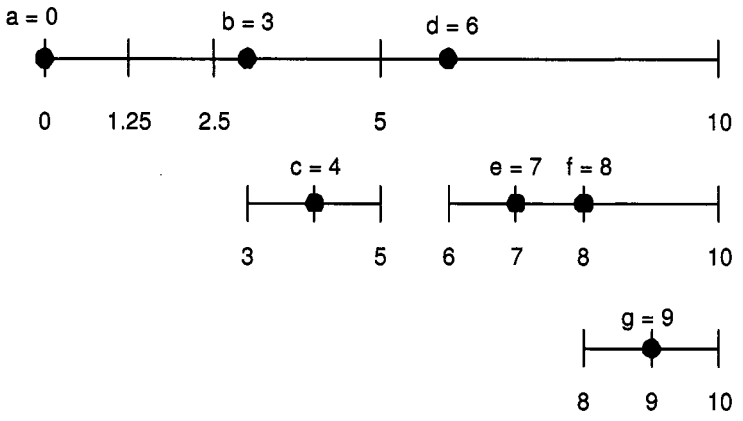


Figure 1.6: Example of subinterval creation for the dyadic algorithm on arrivals $a = 0$, $b = 3$, $c = 4$, $d = 6$, $e = 7$, $f = 8$, and $g = 9$.

1.4 Related Work

1.4.1 Media Delivery

Several approaches have been proposed for delivering media files for media-on-demand systems. Sincoskie introduces the video-on-demand scenario and the necessary server resources to host several movies and distribute them to clients [66]. An early proposal by Anderson was to simply batch client requests together, so a single stream can serve multiple clients [4]. The batching approach trades off client latency (how long a client must wait before receiving the media file) and server bandwidth. The larger the batching interval, the longer the client latency. When these intervals are large, the system does not support on-demand access. In a system supporting multiple movies, Hwang and Wu studied various batching policies for selecting movies to distribute based on several properties of the system including client waiting times, movie popularity, and total number of clients wanting to watch a common movie [45]. Sheu *et al.* extended the batching idea by creating a chain of clients that serve each other pieces of the media that they store in their local buffers [64]. The virtual batch is a chain of clients that all receive a common full-length stream, but the later clients in the chain receive the stream data from an earlier client in the chain. With these virtual batches, clients no longer need to wait to receive data at batch intervals.

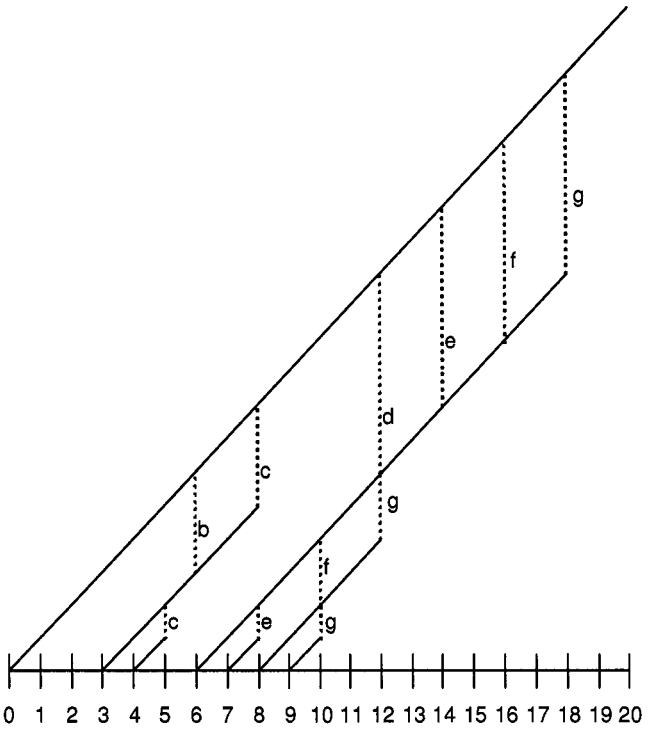


Figure 1.7: Merge schedule created by the recursive dyadic algorithm for client arrivals $a = 0$, $b = 3$, $c = 4$, $d = 6$, $e = 7$, $f = 8$, and $g = 9$. The vertical dotted lines indicate merges and the labels along the lines indicate the clients involved.

Pyramid Broadcasting, originally proposed by Viswanathan and Imielinski, divides the media file into segments to play on different channels at set times [56]. In this model, clients receive multiple streams simultaneously and buffer future media data. Clients receive schedules for tuning into the appropriate channels from the server. For example, Hua and Sheu describe skyscraper broadcasting where the media file is broken into segments of size 1, 2, 2, 5, 5, 12, 12, 25, 25 ... and so on [43]. Each segment is repeatedly played on its own channel. By having a short segment (size 1), the client start-up-delay is shorter since the delay is at most the length of this segment. When a client arrives, it receives and starts playing the unit-length segment, while receiving and buffering segments 2 and 3 from a different channel. This process keeps going until a client has received the full media stream. Note that in this model clients must have the ability to store future parts of the media

stream.

Patching was one of the first approaches to use limited stream merging. Clients receive a portion of the media file to patch the data of an existing full file transmission [42, 63, 11, 36, 12]. Patching utilizes buffer space in client workstations to decrease the number of streams sent by the server. Patching assumes clients can receive at least two streams simultaneously. Each client receives its "patch" stream and the existing full-length media stream until the client has enough data to simply receive the full-length media stream. For example, assume a client arrives while a full-length root stream is currently distributing segment 10 of 99. The patch stream for this client would include segments 0 through 9 of the media, since the client also starts receiving segment 10 from the root stream upon its arrival to the system.

Aggarwal *et al.* proposed a similar concept to patching by using two play rates instead of client buffer storage [1]. They assume the media can be delivered at two play rates – a slow rate and a fast rate and people cannot perceive the differences in play rates when the video is shown. Here a faster stream can catch up to a slower root stream and the client can simply receive the slower root stream after the faster stream has caught up to the root stream. Lau *et al.* also study how the display rates of videos can be altered so that clients can merge and share a single stream [37, 52].

When the client buffer space can only support B media segments and a client arrives when the most recently started full-length root stream has already distributed B segments, a simple patch stream will not suffice. The patch stream would have to be longer than B media segments. Carter and Long introduced a technique called stream tapping where clients can receive a partial patch stream in this situation and alternately display data directly from a stream and data from its buffer [13]. For example, assume a client arrives at time 8 after a full-length transmission started at 0. Also, the client can buffer 5 media segments. The client receives a partial patch stream of length 5 containing media segments 0 through 4 while buffering segments 8 through 12 from the full-length transmission. Since the client needs segments 5 through 7, it gets another partial stream for these segments while keeping its buffer full. For media segment 8, it starts playing data from its buffer while buffering the full-length stream which is transmitting segment 16. Once it plays out

segment 12 from its buffer, it requests another patch stream for segments 13 through 15. This process continues repeatedly until the client receives all media segments.

The original hierarchical stream merging algorithms were event-driven, such as ERMT as described above [30, 32]. Other stream merging algorithms, such as the dyadic algorithm [21], are not event-driven. Algorithms that are not event-driven are easier to reason about which is helpful for their analysis. Chan *et al.* created an algorithm called the connector algorithm that uses rectilinear trees in order to show that it is 5-competitive with respect to the optimal offline stream merging schedule [17]. They model client arrivals as leaves of a rectilinear tree and use a canonical rectilinear tree to make online decisions about clients' merge targets. Bar-Noy *et al.* performed simulations to compare the hierarchical stream merging algorithms described above [6]. Most of the existing stream merging algorithms do quite well in practice, coming close to the performance of the optimal offline algorithm under simulated workloads.

The hierarchical stream merging algorithms mentioned above assume clients can receive two full streams simultaneously. Eager *et al.* proposed bandwidth skimming to allow hierarchical merges when clients have bandwidth greater than the bandwidth necessary to receive a single stream but less than the bandwidth necessary to receive two streams [31]. In bandwidth skimming clients can skim off their target stream, but not at the full receive rate. Thus, each media segment must be divided and distributed on several channels. For example, let us assume clients can receive 1.33 data streams simultaneously. Assume a client arrives and is granted to receive a full-length stream at time 0. At time 4, a new client arrives and is scheduled to merge to the client arriving at time 0. Because the new client can receive 1.33 streams simultaneously, it skims one-third of the stream started at 0. (This is why media segments are distributed on three separate channels. The client can only receive one of the channels that uses .33 of its available bandwidth.) Between times 4 and 8, the client receives all of segments 0 through 3 from its own dedicated stream while receiving one-third of media segments 4 through 7. At time 8, the client only needs two-thirds of the media segments 4 through 7, so it receives two channels worth of segments 4 through 7 from its own stream plus two-thirds of sègments 8 through 11 from its target stream. During times 8 through 11, the client receives one-third of segments 8 through 11 from its own

stream and the entire set of segments 12 through 15 from its merge stream. At time 16 the client merges to its target stream. The bandwidth skimming technique allows for clients to eventually merge to each other without the condition that clients can receive at least two full streams simultaneously.

When comparing online stream merging algorithms, it is useful to compare their performance against the optimal offline solution. In fact, determining the optimal stream merging schedule and the optimal total cost for a set of client arrivals is efficient [8]. A key insight into determining the optimal solution is that optimal schedules are guaranteed to have a certain structure. Bar-Noy and Ladner prove that in an optimal schedule, if client c_j is the last to merge to the root stream, then the clients arriving after c_j must eventually merge to c_j before merging to the root stream [8]. This property allows the client set to be divided into two disjoint sets of consecutive client arrivals which is suited for a dynamic programming algorithm. The dynamic programming algorithm for determining the best merge cost (where all clients eventually merge to a common root stream) is $O(n^2)$ for n clients [8]. Finding the best total cost is simply a task of selecting clients to receive full-length streams. Finding the optimal offline solution for n clients with an average of m clients arriving during the length of the full media file is $O(nm)$ [8].

1.4.2 Time-shifting Model

As far as we know, Goshi is the first to introduce the media-on-demand with time-shifting model in [38]. He adapted the dyadic algorithm for the media-on-demand model to work in the time-shifting case. Because clients are no longer guaranteed to be able to merge to clients arriving before them, the original dyadic algorithm [21] using a stack does not work in the time-shifting case. Therefore, Goshi modified the dyadic algorithm to use a list of dyadic intervals. When a client arrives to the system, a search through the list is conducted to find a suitable merge target. Because clients request an arbitrary point of the previously broadcasted stream, the algorithms proposed for the original media-on-demand model must be modified to account for arbitrary access into the live broadcast.

A related model allows arbitrary access to a fixed-size media file, such as a movie. Instead

of clients always requesting the first segment of the media file and playing the file to its end, clients in this model can request any part of the media file. This differs from the time-shifting model, since the media has finite length. Jin and Bestavros study the scalability of multicast and stream merging in this arbitrary access model [49]. They found that the lower bound of total server bandwidth (total cost) is on the order of $\sqrt{n}$. They analytically found the lower bound by determining how often each media segment needs to be transmitted under the arbitrary access model. They experimentally confirmed the lower bound through simulations. Tan, Eager, and Vernon studied the minimum server bandwidth under similar access models [69], but under the assumption that clients can receive an arbitrary number of streams concurrently. First, they found that arbitrary accesses from the beginning of the file to random ending points require server bandwidth that scales with $lg(n)$ for n client arrivals. Their results are similar to [49] with server bandwidth scaling with $\sqrt{n}$ for clients receiving an arbitrary first segment through the end of the media file and for clients receiving arbitrary first and last segments. Supporting arbitrary first and last segments provides clients with VCR-like functions of rewind and fast-forward.

Guo and Ammar present a mechanism for clients to receive a lossless live video stream in the event data does not reach the clients, perhaps due to network congestion or because a client who is distributing the stream to other clients decides to leave the multicast group [40]. In their model, clients are part of a distribution tree where a client could receive streaming data from another client in the tree or directly from the server. If a client's parent in the tree decides to leave, the client will be disconnected and miss portions of the live stream while it rejoins the tree at a different location. Guo and Ammar propose a patching scheme so that clients receive a patch consisting of the lost data while simultaneously receiving the live stream once they rejoin the distribution tree [40]. Once the client receives the lost data, it can simply receive the live stream. Their scheme supports lossless reception of a live stream, but does not support the arbitrary access into previous parts of the live stream as does the media-on-demand with time-shifting model.

Zhao, Eager, and Vernon study a model with "choose your own ending" non-linear media [77]. Instead of receiving a movie from beginning to end, clients can dynamically select branches of the video to follow. Instead of arbitrary access points into a single media

file, clients now choose arbitrary versions of the movie at decision points. When a client arrives to the system, choosing a merge target becomes more complicated since the stream to which a client will merge may not transmit the movie version the client will decide to view. Zhao, Eager, and Vernon found that using a periodic broadcast for this model performs better than stream merging [77]. In their periodic broadcasting scheme, clients can receive the first part of each movie version after a branch point. Once the client has decided on the movie branch to take, it can receive just the data for the chosen branch.

1.4.3 Network Cost Model

Throughout the discussions above, the cost to be minimized is server bandwidth. The media-on-demand system also uses other resources, such as the network to transmit media data from the server to the set of clients. A model which we consider in Chapter 6 assigns the distribution network with link costs for file transmission. Dammicco and Mocci present a network cost model in the context of determining local server locations in a distribution tree [26]. In their optimization problem, they consider both the storage cost of the main server and the transmission costs along links in the distribution tree. Wong and Chan study a similar problem where local server locations are fixed and can serve both local clients (close in the network topology) and remote clients (clients who are physically closer to a different local server) [76]. Each local server has a limited number of concurrent streams. Wong and Chan analyze the probability of clients being blocked when they issue requests and determine the required network bandwidth for their model. Both [26] and [76] assume that clients receive a single media stream (no stream merging). To relieve the burden on the server and the network, Wang *et al.* study the impact of caching media content at proxies close to the clients [75]. Since multicast is not widely supported today, they developed caching policies close to the clients to save in transmission costs in unicast-enabled networks.

Zhao, Eager, and Vernon also study network bandwidth for multicasting media data [78]. They analyze three network topologies and determine the minimum network bandwidth required to serve media to client sites. They found that creating a chain of client sites achieves the best network bandwidth usage, but creating a distribution tree in the form of

a chain might not be practical for a given set of clients. Simulations show that aggregating clients from the same site in the distribution tree can save network bandwidth at the expense of additional server bandwidth.

Waldvogel and Janakiraman studied the impact of periodic broadcasting on the network bandwidth [74, 47]. Recall that in periodic broadcasting, media segments are repeatedly broadcasted on channels and clients tune into the appropriate channels to receive media segments. Waldvogel and Janakiraman use the broadcast scheme where media segment 1 is broadcasted repeatedly on a channel, media segment 2 is broadcasted every two time units (where one time unit is the time to transmit one segment), segment 3 is broadcasted every 3 time units, etc. They studied the impact of forming multicast groups, both on the client receive bandwidth and the overall network bandwidth. Since joining and leaving multicast groups require communication-intense operations, having a multicast group for each media segment is not practical. Thus, their model is to have clients join all multicast groups upon arrival to the system and then shed their membership from groups that are receiving media data they no longer need. They found that a small number of multicast groups, on the order of 10, results in network efficiency that is close to optimal for broadcasting schemes.

1.5 Contributions

Central to this dissertation is the insight that analysis for the offline case can inform the development of online algorithms for media-on-demand models. Our goal in this work is to reduce total server bandwidth cost for delivering a media file to a set of clients. In Chapter 2 we study the dyadic algorithm and show how to optimize the set of parameters under high client arrival intensities. We show the optimization for the case where clients can receive at most two streams simultaneously and the case where clients can receive streams from all of its eventual merge targets simultaneously. By analyzing the dyadic algorithm, we make a connection between dyadic intervals and the use of Fibonacci numbers in the offline analysis of stream merging for high client loads [5]. This offline analysis proves helpful in devising an online algorithm that can periodically achieve the optimal server bandwidth under high client request intensities. We also present a modification to the online algorithm

for uniformly-spaced client arrivals.

The second area in which offline analysis informs online decisions is the case of media-on-demand with time-shifting. We are the first to model the media-on-demand with time-shifting problem on a rectilinear grid. Chapter 3 introduces the rectilinear merge tree embedded in a rectilinear grid to model streams, clients, and mergers in the system. We introduce properties and transformations on rectilinear merge trees to prove that a single, normalized rectilinear merge tree is optimal for a given merge schedule. We prove that the number of potential trees for a given client set is finite, making an exhaustive search possible. In Chapter 4 we show that finding the optimal offline solution for media-on-demand with time-shifting is NP-hard. This might be surprising since the media-on-demand model has an efficient optimal offline dynamic programming algorithm [8]. The reduction proof follows from a key insight between rectilinear merge trees and the Rectilinear Steiner Arborescence Problem [65]. As in [65], we reduce an instance of the planar 3SAT problem to a rectilinear merge tree. The rectilinear merge tree has total cost less than or equal to a given integer K if and only if the original planar 3SAT instance has a satisfying assignment. In Chapter 5 we introduce and empirically study an online algorithm for the time-shifting case based on rectilinear merge trees. The online algorithm is based on the structure of the rectilinear merge tree. We compare our rectilinear algorithm to the dyadic algorithm modified for the time-shifting case and show that the rectilinear algorithm achieves lower total cost through simulations. We also modify our rectilinear algorithm for the standard media-on-demand model and show that it performs comparably to the dyadic algorithm and ERMT. We conclude chapter 5 by showing that finding the optimal offline solution to the arbitrary access problem (clients request an arbitrary media segment through the end of the media file) and the arbitrary range access problem (clients request arbitrary start and end media segments) is NP-hard.

In the final chapter we introduce a network cost model, where instead of minimizing total server bandwidth, we try to minimize the total cost of sending streams through a network that supports multicast. We present the model, show how to perform the cost calculation, and provide examples of online algorithms that take into account the distribution network. We also show a case where the optimal merge schedule in terms of network cost does not have

the property that merge groups consist of sequentially ordered clients. Another difference in this model is that it might be advantageous to treat clients arriving at the same time differently. In both the standard media-on-demand model and the time-shifting case, we treat clients with the same arrival times and requested first segments as a single client. This is not necessarily the case in the network cost model.

1.5.1 Guide to Reader

This dissertation need not be read sequentially from Chapter 2 to Chapter 6. Chapter 2 and portions of Chapter 5 assume the standard media-on-demand model, where the background information is presented in this introductory chapter. Chapter 3 should be read before Chapters 4 and 5 in order to understand the time-shifting model. You may wish to skip Chapters 2 and 6 if you are most interested in reading about the time-shifting model.

Chapter 2

OPTIMIZING THE DYADIC ALGORITHM FOR STREAM MERGING

The dyadic algorithm, presented in the introductory chapter, creates stream merge schedules based on dyadic intervals. The originally proposed dyadic algorithm has a set of constants used to determine when new root streams (full-length files) should start and how to determine cutoff times for dyadic intervals [21]. The parameter used to determine cutoff times for dyadic intervals is 1/2, as shown in Section 1.3.2.[1] In this chapter, we generalize the constants and perform analysis to determine their optimal values. Throughout this chapter, our objective is to minimize total server bandwidth necessary to distribute media streams to a set of client requests. Our contributions include showing how to optimize the dyadic algorithm, making a connection between Fibonacci numbers and the dyadic algorithm, and proposing a new algorithm that can achieve optimal server bandwidth under intense client arrivals.

The remainder of the chapter is organized as follows: we introduce the method and analysis to optimize the parameters for the dyadic algorithm for both the receive-two and receive-all models. We then introduce an algorithm that works well for high demand client loads. We extend this algorithm to adapt to the client arrival rate and conclude with future research directions.

2.1 Optimal Parameters for the Dyadic Algorithm

Let us assume the *continuous time response* model, where the server initiates a stream immediately upon a client request. In the original algorithm described in Section 1.3.2, the

[1]Other papers make reference to this constant as 2, instead of 1/2. We use 1/2, since it makes the notation cleaner in the analysis. In other chapters of this dissertation, we will refer to the original dyadic algorithm as the 2-Dyadic.

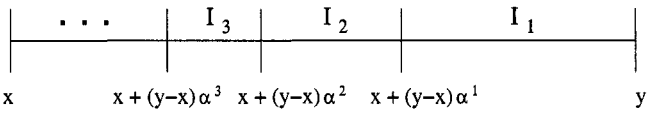


Figure 2.1: Illustrates the creation of the subintervals in the dyadic algorithm. We show the subintervals for the interval $[x, y)$. The constant α dictates the sizes of these intervals.

pair for a new root stream started at t is $[t, t + L/2)$ where L is the length of the media. Clients who arrive $L/2$ time units after the initial root stream (started at time 0) will not be allowed to merge with the initial root stream. Thus, such a client at time t will start a new root stream and pushes the pair $[t, t + L/2)$ on to the stack S. We generalize the parameter β for dictating the interval corresponding to a root stream. In the generalized version of the algorithm, we instead push $[t, t + \beta L)$ on to S for new root streams.

The original dyadic algorithm produces intervals with the constant factor 1/2. Instead, we generalize the constant 1/2 as α as illustrated in Figure 2.1. The figure shows the partitioning of the interval $[x, y)$. When a client c arrives within $[x, y)$, we find the dyadic subinterval into which c falls. The right-hand side of this subinterval dictates the value t_r for the pair that will be pushed onto the stack. The arrival c will start receiving the stream started at time x and eventually merge to the stream initiated at time x.

Our approach for finding optimal values for α and β models the merge cost and total cost for continuous arrivals in a bounded time interval. Below we show a recurrence that models the cost for the arrivals and we find a solution to the recurrence. The solution that we present does satisfy the recurrence, but it is an approximate solution and we verify that it does achieve costs similar to those found in simulating dense client arrivals.

2.1.1 Optimizing the Merge Cost

Bar-Noy *et al.* show a recurrence for the merge cost of the dyadic algorithm and optimize a function satisfying the recurrence to achieve $\alpha = 1/\phi = 2/(1 + \sqrt{5})$ (ϕ is the golden ratio) [6]. They assume continuous arrivals in the interval $[0, x]$ where a single root stream starts at time 0. They assume the root stream is long enough for the last client at time x to

eventually merge to the root stream. The merge cost approximately satisfies:

$$M_\alpha(x) = M_\alpha(\alpha x) + M_\alpha((1-\alpha)x) + (2-\alpha)x \tag{2.1}$$

This recurrence demonstrates the recursive structure of the dyadic algorithm. We measure the merge cost for arrivals in $(0, x]$ by breaking the cost into two recursive parts. The first term of the equation represents all the streams started before time αx. The second term is the cost of the streams started after time αx and the third term represents the length of the stream started at time αx.

The following equation satisfies the above recurrence and is minimized when $\alpha = 1/\phi$ [6].

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

Notice that this solution is negative when $x < 1$. However, the function does approximate the merge cost for large values of x. Figure 2.2 shows simulation results for the merge cost divided by $N \ln N$ where N is the number of arrivals. The function $M_{1/\phi}(x)$ divided by $x \ln x$ is simply a flat line equal to the coefficient in Equation 2.2. Because the merge cost simulation has an overall flat trend approaching $M_{1/\phi}(x)/x \ln x$, we conclude that $M_\alpha(x)$ defined in Equation 2.2 approximates the merge cost for large x. We use Equation 2.2 to minimize the merge cost.

$M_\alpha(x)$ it is minimized when $\alpha = 1/\phi$. The simulation results for α in Figure 2.3 confirm that $\alpha = 1/\phi$ is the optimal value. In this experiment, $\beta = .48$ and we graph the total bandwidth in terms of the number of full-length streams (media file length). The experiment length was 50 media files with a Poisson client request intensity of 10,000 per media length. The total bandwidth is simply the merge cost plus the full-length streams serving the root clients. Since β is constant, the number of full-length streams is the same for each experiment, and the difference in the total cost is directly related to the difference in the merge cost.

2.1.2 Optimizing the Total Cost

We have shown how to optimize α which dictates the merge cost for the dyadic algorithm. Let us consider the case of optimizing the total cost. Recall that the total cost is simply the

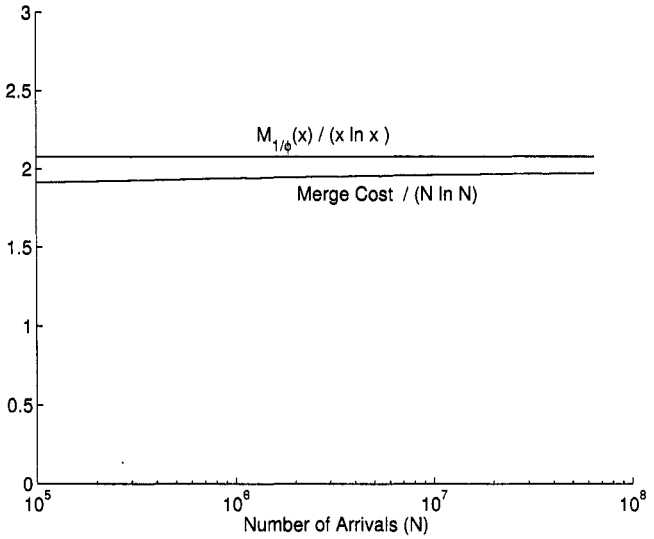


Figure 2.2: Simulation results showing the merge cost divided by $N \ln N$ for clients arriving at each time unit. Also shown is Equation 2.2 divided by $x \ln x$ with α equal to $1/\phi$.

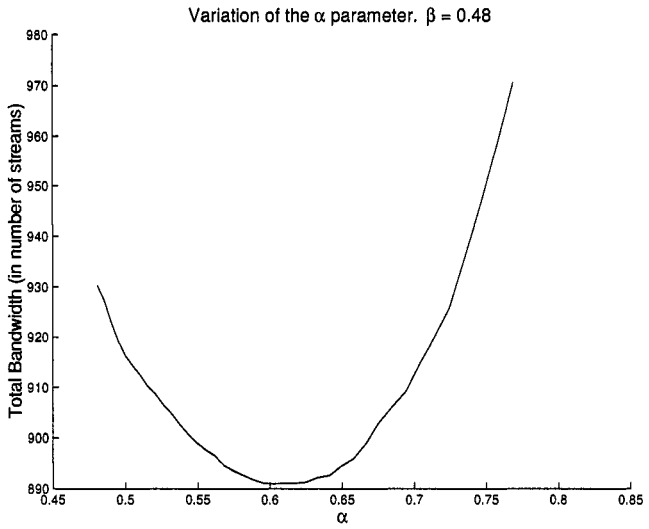


Figure 2.3: Experimental results for the dyadic algorithm with various α from .5 to .77. The total bandwidth is the total cost in terms of the number of full-length streams to serve all clients. The experiment of length 50 media files used a Poisson request intensity of 10,000 clients per media length.

merge cost plus the cost of the root streams. The number of root streams is determined by β, the fraction of the media after which we start a new root stream in the dyadic algorithm. Let L be the length of the media file. We approximate the total cost for continuous arrivals on $[0, x]$ by the following:

$$T_{\alpha,\beta,L}(x) = \frac{x}{\beta L} M_\alpha(\beta L) + \frac{x}{\beta} \tag{2.3}$$

The first term represents $x/\beta L$ merge groups with merge cost $M_\alpha(\beta L)$ on the interval $(0, \beta L]$ and the second term represents the number of merge groups $x/\beta L$ times the length of the media, L. We assume a negligible cost for the last, incomplete merge group. When βL is large, as with a large media file, we can approximate the merge cost with Equation 2.2. We substitute this equation for $M_\alpha(\beta L)$ in Equation 2.3 to get:

$$T_{\alpha,\beta,L}(x) = \frac{x(\alpha - 2)\ln(\beta L)}{\alpha \ln(\alpha) + (1-\alpha)\ln(1-\alpha)} + \frac{x}{\beta} \tag{2.4}$$

Our objective is to minimize the total cost with respect to β, so we minimize the function $T_{\alpha,\beta,L}(x)/x$. After substituting $1/\phi$ as the optimal α, we let C be a constant and minimize $T_{\alpha,\beta,L}(x)/x$:

$$C = \frac{\frac{1}{\phi} - 2}{\frac{1}{\phi}\ln(\frac{1}{\phi}) + (1 - \frac{1}{\phi})\ln(1 - \frac{1}{\phi})} \tag{2.5}$$

$$\frac{T_{\alpha,\beta,L}(x)}{x} = C\ln(\beta L) + \frac{1}{\beta} \tag{2.6}$$

By elementary calculus, the optimal β satisfies the equation $0 = C/\beta - 1/\beta^2$, so $\beta = .4812$. Thus, choosing β to be .4812 is optimal for the *receive two* model and continuous time response. Figure 2.4 shows simulation results for the total bandwidth in terms of the number of streams. Here, we have a Poisson client arrival intensity of 10,000 arrivals per media length. We average the total cost over experiments of length 45 media files to 55 media files.

2.1.3 Optimizing the Dyadic Algorithm for the Receive-All Model

We presented above the analysis to determine optimal parameters for the receive-two model. We can perform a similar analysis to determine the optimal values for α and β in the *receive-all* model. In the receive-all model [5], clients can receive data from all existing streams in

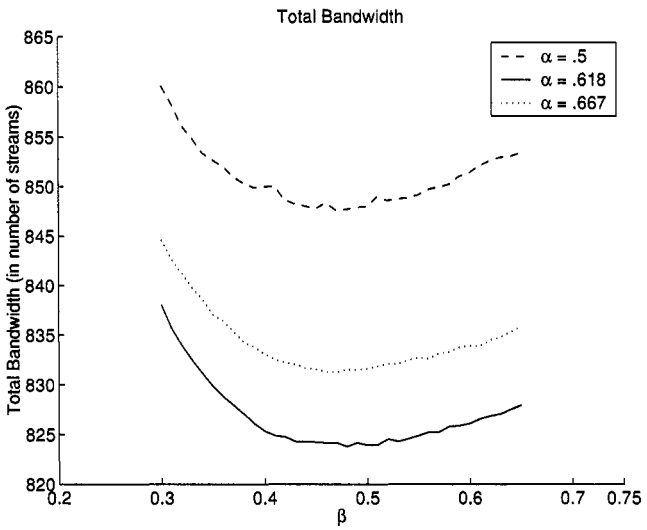


Figure 2.4: Simulation results showing the total bandwidth for various values of β. The simulation results are consistent with the theoretical optimum of .4812 for α equal to .618.

the system. This requires that a client's receive bandwidth is capable of receiving all streams simultaneously. When a client arrives to the system, the client can receive all streams being distributed to its series of merge targets. In this model, the length of a stream x is equal to the start time of the last stream to merge to x minus the start time of the stream to which x merges. We again assume continuous time response for the arrivals and assume continuous arrivals in the interval $[0, x]$, where a root stream starts at 0.

The merge cost for the dyadic algorithm in the receive all case approximately satisfies:

$$M_\alpha(x) = M_\alpha(\alpha x) + M_\alpha((1-\alpha)x) + x \tag{2.7}$$

A solution to Equation 2.7 is below and can be verified algebraically.

$$M_\alpha(x) = \frac{x \ln(x)}{-\alpha \ln(\alpha) - (1-\alpha)\ln(1-\alpha)} \tag{2.8}$$

By taking the derivative of the coefficient in Equation 2.8 and setting it to zero, we determine the optimal value for α is $1/2$. Thus, when executing the dyadic algorithm in the receive-all model for continuously arriving clients, the best value for α is $1/2$, which was the

originally proposed constant.

We can optimize β by performing the same analysis as in the receive-two case. We start with the same equation for approximating the total cost, shown in Equation 2.3. We substitute the right-hand side of Equation 2.8 in for $M_\alpha(\beta L)$ to get the following:

$$T_{\alpha,\beta,L}(x) = \frac{-x\ln(\beta L)}{\alpha\ln(\alpha) + (1-\alpha)\ln(1-\alpha)} + \frac{x}{\beta} \tag{2.9}$$

After substituting the optimal $\alpha = 1/2$ and dividing Equation 2.9 by x, we can solve for the best β value. The best β value is $-\ln(1/2)$ which is equal to .69314. In the receive-all model, it is best to start a new root stream $.69314L$ units after the existing root stream when clients arrive continuously. We have shown how to optimize α and β for the dyadic algorithm in the receive-two and receive-all models.

2.2 An Algorithm for Popular Media

We have assumed that clients are served immediately upon request in the continuous time response scenario. Let us assume now that client requests are batched together and treated as a single client. Clients will be served according to units determined by the size of a segment in the media file. For the remainder of this dissertation the media file length L is the number of segments constituting the media. For example, if a 2-hour movie is broken into 720 segments, then $L = 720$. Clients arriving within the same 10-second window (a single segment) will be treated as a single client from the standpoint of the server. In this scenario, clients have a maximum start-up delay of 10 seconds. If the media is popular, we may have a client request every media segment length. We call this scenario *discrete high demand* since the server receives client requests during each media segment and the media is broken into a discrete number of segments.

Bar-Noy *et al.* show how to construct optimal offline algorithms in terms of merge cost and total cost for the case where we have N client arrivals 0, 1, ..., N - 1 = $[0, N-1]$ [5]. We show how to convert this offline algorithm into an online algorithm that achieves optimal total cost periodically and whose total cost approaches the optimum asymptotically. First, we show how to create an online algorithm when L has an upper bound. We show how to

extend the algorithm to work for general L. The following theorem gives the optimal total cost using Fibonacci numbers. The Fibonacci numbers are defined as $F_0 = 0$, $F_1 = 1$, and $F_i = F_{i-1} + F_{i-2}$ for $i \geq 2$. $M(p)$ is the optimal merge cost for p clients in a single merge group.

Theorem 2.2.1 (Theorem 12 [5]) *Let h be such that $F_{h+1} < L + 2 \leq F_{h+2}$, where F_h is the h^{th} Fibonacci number, and let $s_1 = \lfloor N/F_h \rfloor$. Either $f(s_1)$ or $f(s_1 + 1)$ is the optimal full cost where $f(s) = sL + rM(p+1) + (s-r)M(p)$.*

When s_1 is exactly equal to N/F_h, we have s_1 merge groups of size F_h. Using the dyadic algorithm, we choose β such that $F_h - 1 < \beta L < F_h$, creating merge groups of size F_h. Whenever N is a multiple of F_h, we create the optimal number of groups.

Now we must ensure that the α in the dyadic algorithm gives us the optimal merge cost according to the offline algorithm. According to the offline algorithm, when we have a Fibonacci number, F_h, of elements in the interval $[0, F_h - 1]$, the last client that can merge to the root in the optimal merge group is the client at time F_{h-1} [5]. This applies recursively for all subintervals within the interval. Along the interval $[0, F_{h-1} - 1]$, the last client to merge to the root is F_{h-2}. Since we defined β to create groups of size F_h, the dyadic intervals must align such that $F_i - 1 < F_h \alpha^{h-i} < F_i$ for $i = 3$ to $i = h - 1$. Figure 2.5 shows the relationship graphically.

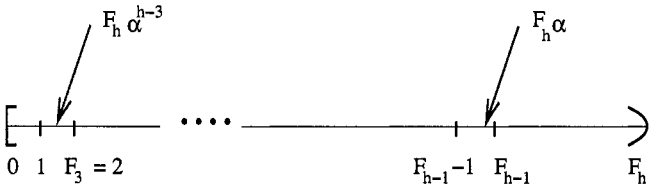


Figure 2.5: Indicates the constraints on α for the optimal merge cost in the discrete high demand case.

We can numerically find bounds for α for $F_4 = 3$ to $F_{15} = 610$ as indicated in Table 2.1. Thus, the length of the media can be up to 1595 segments, ($F_{16} = 987$ and $F_{17} = 1597$), and we can find an α that splits the dyadic intervals optimally for the merge cost.

Table 2.1: Shows the possible values for α in the Dyadic Algorithm for the discrete high demand scenario. F_h indicates the size of the interval and the Min α and Max α dictate the range of α that splits the interval optimally for the merge cost.

Min α	Max α	F_h	Min α	Max α	F_h
.3335	.6668	3	.6031	.6158	55
.4473	.6001	5	.6090	.6161	89
.5001	.6124	8	.6124	.6163	144
.5547	.6133	13	.6146	.6165	233
.5774	.6147	21	.6159	.6167	377
.5941	.6153	32	.6167	.6168	610

We give an example of how the choice of α creates the correct subintervals for the interval [0, 610). Table 2.2 shows the subintervals created by using $\alpha = .61677$ for an interval of length $F_{15} = 610$. Each subinterval has a start time and an end time created by α. Note that the arrivals come on integer values, so we extend the endpoints created by α to be integer values. We see that the integral intervals are split such that the beginning of each subinterval is a Fibonacci number for $i > 3$. Also, $F_i - 1 < F_h\alpha^{h-i} < F_i$ for $3 \leq i \leq (h-1)$. The lower values of i may not fit these constraints, but for some $i < 3$, $F_h\alpha^{h-i}$ will be between 1 and 2, thus creating an interval for the client arrival at time 1 unit after the root client. Each subinterval created using integral endpoints is of length equal to a Fibonacci number. For example, the interval [233, 377) in Table 2.2 has length 144. When creating subintervals for [233, 377), we use an α value between Min α and Max α in Table 2.1 for $F_{12} = 144$.

We have shown how to choose β and α in the dyadic algorithm to achieve the optimal total cost in bandwidth when the number of arrivals is a multiple of F_h. Figure 2.6 shows the factor increase of the discrete high demand algorithm versus the optimal total cost for media length of 720. The big spikes indicate a root stream being sent to a client group consisting of a single client. Because the optimal offline algorithm can construct an optimal merge pattern based on all arrivals, the optimal algorithm will create more comparably-

Table 2.2: Shows how the dyadic intervals are created with $\alpha = .61677$ on the interval [0, 610). The third column indicates the intervals with fractional numbers and the fourth column shows the integer-valued intervals. We use the integer-valued intervals, since we assume arrivals come at integral times.

i	$F_h\alpha^{h-i}$	Interval with endpoint	Integral Interval
15	610.00	[376.23, 610)	[377, 610)
14	376.23	[232.05, 376.23)	[233, 377)
13	232.05	[143.12, 232.05)	[144, 233)
12	143.12	[88.27, 143.12)	[89, 144)
11	88.27	[54.44, 88.27)	[55, 89)
10	54.44	[33.58, 54.44)	[34, 55)
9	33.58	[20.71, 33.58)	[21, 34)
8	20.71	[12.77, 20.71)	[13, 21)
7	12.77	[7,88, 12.77)	[8, 13)
6	7,88	[4.86, 7,88)	[5, 8)
5	4.86	[2.99, 4.86)	[3, 5)
4	2.99	[1.85, 2.99)	[2, 3)
3	1.85	[1.14, 1.85)	[2, 2)
2	1.14	[.70, 1.14)	[1,2)

sized merge groups. Any online algorithm will produce these spikes when the server decides to distribute a new root stream to the final client arrival.

2.2.1 General Discrete High Demand Algorithm

Let S be a stack with *push* and *pop* operations for pairs $[t_a, t_r)$. Each pair indicates a stream where t_a and t_r indicate a window dictating the clients that can merge with the first stream initiated at t_a or later. Let L be the media length in segments and F_h be such that $F_{h+1} < L + 2 \leq F_{h+2}$.

At request time T, *pop* the pairs $[t_a, t_r)$ from S until $t_r > T$. If S is empty, *push*

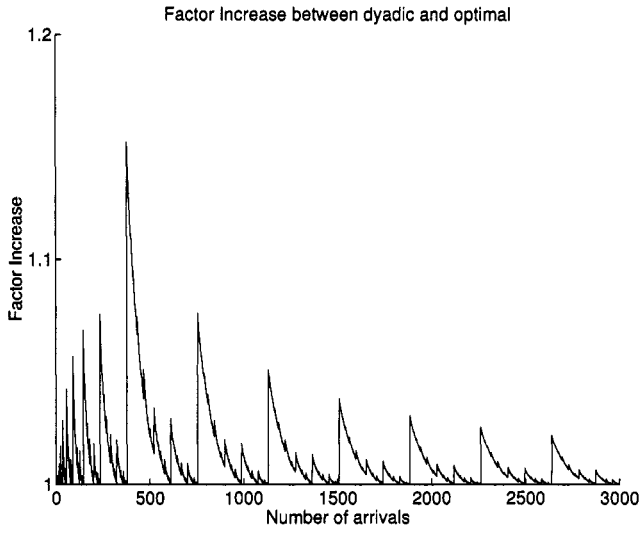


Figure 2.6: Simulation results for the discrete high demand variation of the dyadic algorithm. The media length is 720 with client arrivals every time unit. When the number of arrivals is a multiple of 377, the algorithm achieves the optimal total cost.

$[T, T + F_h)$ on to S. Otherwise, find the Fibonacci number F_k such that $t_a + F_k \leq T < t_a + F_{k+1}$. Add the new stream to the stack by *pushing* $[t_a + F_k, t_a + F_{k+1})$ to S. The stream started at T will merge to the client who arrived at time t_a.

2.2.2 Discrete High Demand Example

We show how the algorithm creates client merge decisions with an example. Suppose $L = 25$. Then $F_h = 13$. Let us assume clients arrive between time 0 and time 12, with an arrival every unit. Table 2.3 shows the client arrival, the state of S after popping pairs until $t_r > T$, and the state of S after pushing the pair for the new stream. We also indicate to whom the client at time T will merge. The merge decisions are the same as in [5] for arrivals on $[0, 12]$. The total cost for these 12 arrivals is optimal.

2.2.3 Experimental Comparison

The Discrete High Demand Algorithm does achieve optimal total cost and achieves a better average factor increase than the original dyadic algorithm. Figure 2.7 (top) shows the factor

Table 2.3: Illustrates the Discrete High Demand Algorithm. The Client column shows the client arrival time T. The second column shows the stack S after popping pairs until $t_r > T$. The third column shows the state of S after pushing the pair representing the new stream for client T and the final column shows to whom the client arriving at T will merge.

Client	S Before	S After	Merge
0	null	$[0, 13)$	none
1	$[0, 13)$	$[0, 13), [1, 2)$	0
2	$[0, 13)$	$[0, 13), [2, 3)$	0
3	$[0, 13)$	$[0, 13), [3, 5)$	0
4	$[0, 13), [3, 5)$	$[0, 13), [3, 5), [4, 5)$	3
5	$[0, 13)$	$[0, 13), [5, 8)$	0
6	$[0, 13), [5, 8)$	$[0, 13), [5, 8), [6, 7)$	5
7	$[0, 13), [5, 8)$	$[0, 13), [5, 8), [7, 8)$	5
8	$[0, 13)$	$[0, 13), [8, 13)$	0
9	$[0, 13), [8, 13)$	$[0, 13), [8, 13), [9, 10)$	8
10	$[0, 13), [8, 13)$	$[0, 13), [8, 13), [10, 11)$	8
11	$[0, 13), [8, 13)$	$[0, 13), [8, 13), [11, 13)$	8
12	$[0, 13), [8, 13),$ $[11, 13)$	$[0, 13), [8, 13), [11, 13),$ $[12, 13)$	11

increase of the total cost for three algorithms: the original dyadic algorithm with $\alpha = .5$ and $\beta = .5$, the dyadic algorithm with optimal continuous time parameters $\alpha = .618$ and $\beta = .48$, and the Discrete High Demand Algorithm. The simulation used media with length 720. Over the experiment of 24 hours (assuming each media segment is 10 seconds long), the average factor increases are as follows: 1.0321 for the original dyadic algorithm, 1.0085 for the continuous arrivals dyadic algorithm, and 1.0059 for the discrete high demand algorithm.

For the Discrete High Demand Algorithm, $F_h = 377$ is chosen for $609 \leq L \leq 985$. We also compared the three algorithms for $L = 609$ and $L = 985$. Note that in terms of the dyadic algorithm, β is .619 to .383. The results for $L = 609$ are consistent with those for

$L = 720$, where the mean factor increases are as follows: 1.0281 for the original dyadic algorithm, 1.0079 for the continuous arrivals dyadic algorithm, and 1.0062 for the discrete high demand algorithm. When $L = 985$, we do achieve optimal total cost when N is a multiple of 377, but β is .383, meaning that we create more merge groups and root streams than when β is closer to .5. The average factor increases are as follows: 1.0336 for the original dyadic, 1.0073 for the continuous arrivals dyadic, and 1.0083 for the discrete high demand algorithm. Figure 2.7 (bottom) shows the factor increases for the three algorithms for $L = 985$ and over 24 hours of client requests.

We also ran experiments comparing the algorithms when the media is popular and clients arrived according to a Poisson process with mean intensity of one arrival per media segment. Figure 2.8 shows experimental results of the factor increase in the total cost. In this experiment we have 1875 arrivals over 3000 media segment units of time. The mean factor increase for each algorithm is as follows: 1.0817 for the original dyadic, 1.0527 for the continuous arrivals dyadic, and 1.0508 for the discrete high demand algorithm.

2.3 An Adaptive Algorithm for Stream Merging

In the previous section we assumed the media is popular and the clients arrive almost every time unit. We present a variation of the discrete high demand algorithm that adapts to the client interarrival rate. We can adapt the algorithm based on the historical patterns of client requests. We define a history window W, in units of time, where we measure the number of clients requesting the media in W units of time. Let T be the time when we calculate and, possibly, adjust R, the client interarrival rate. We define an approximation for R at time T as follows. Let $N[t_a, t_r)$ be the number of clients who arrived at or after time t_a and before time t_r:

$$N_T = N[T - W, T) \tag{2.10}$$

The client interarrival rate R (average number of segments between arrivals) at time T is simply the window size divided by the number of clients, so we have $R = W/N_T$. If we want to use prior history before the most recent W units of time, we can instead use a weighted

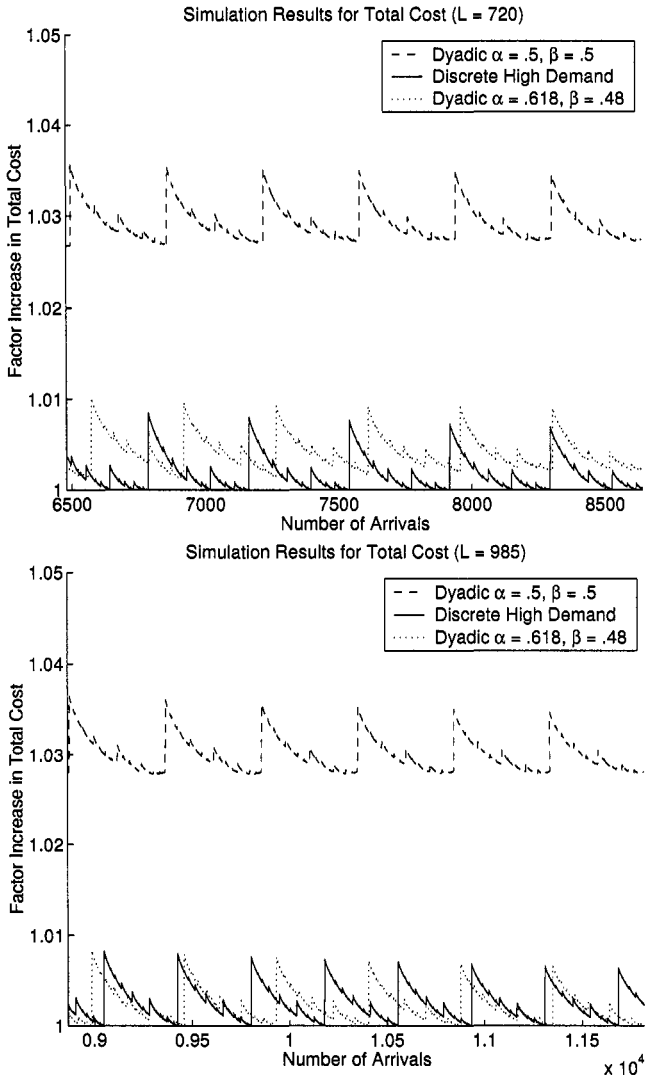


Figure 2.7: Simulation results for clients requesting media every time unit. The number of arrivals, N, indicates that the clients requesting the media are [0,N-1]. The figure shows the time period between 18 hours and 24 hours, with the top using $L = 720$ and the bottom using $L = 985$. The Discrete High Demand Algorithm achieves optimal total cost when N is a multiple of 377.

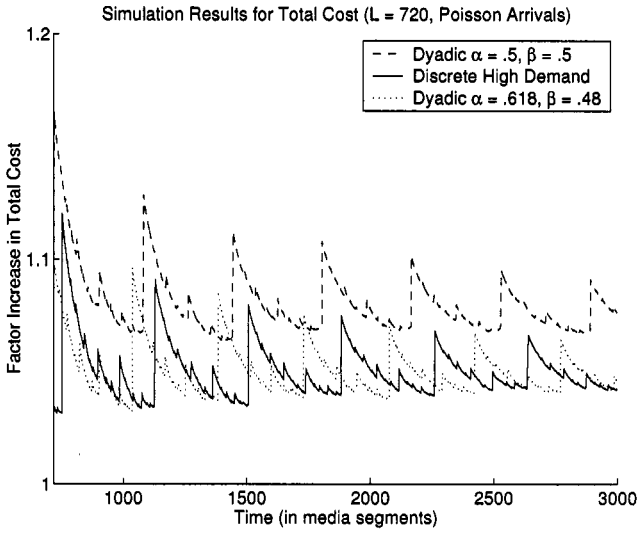


Figure 2.8: Simulation results for Poisson arrivals with mean intensity of 1 for media with length 720. Time is represented in media segments. Over the course of 3000 time units, 1875 clients arrived in this experiment.

sum of the current rate and the previous rate:

$$R_T = \lambda \frac{W}{N_T} + (1 - \lambda) R_{T-W} \tag{2.11}$$

Here, λ, where $0 \leq \lambda \leq 1$, distributes the influence of each rate from the current window and previous windows.

2.3.1 Adaptive Discrete Uniform Demand Algorithm

Let S be a stack with *push* and *pop* operations for pairs $[t_a, t_r)$. Each pair indicates a stream where t_a and t_r indicate a window dictating the clients that can merge with the first stream initiated at t_a or later. Let L be the media length, R be the client interarrival rate, and F_h be such that $F_{h+1} < \text{round}(L/R) + 2 \leq F_{h+2}$. Intuitively, we create a new length of the media to be L/R and use the Fibonacci number F_h for L/R. The pairs on the stack will be of size a Fibonacci number multiplied by R.

At request time T, *pop* the pairs $[t_a, t_r)$ from S until $t_r > T$. If S is empty, *push* $[T, T + F_h R)$ on to S. Otherwise, find the Fibonacci number F_k such that $t_a + RF_k \leq T <$

Table 2.4: Illustrates the execution of the Adaptive Discrete Uniform Demand Algorithm. The Client column represents the client arrival time T. The second column shows the stack S after popping pairs until $t_r > T$. The third column shows the state of S after pushing the pair representing the new stream for client T and the final column shows to whom the client arriving at T will merge. The client intensity request rate R is 2 and L is 14.

Client	S Before	S After	Merge
0	null	$[0, 10)$	none
2	$[0, 10)$	$[0, 10), [2, 4)$	0
4	$[0, 10)$	$[0, 10), [4, 6)$	0
6	$[0, 10)$	$[0, 10), [6, 10)$	0
8	$[0, 10), [6, 10)$	$[0, 10), [6, 10), [8, 10)$	6

$t_a + RF_{k+1}$. Add the new stream to the stack by *pushing* $[t_a + RF_k, t_a + RF_{k+1})$ to S. The stream started at T will merge to the client who arrived at time t_a.

2.3.2 Example

Let L be 14 and R be 2. Let {0, 2, 4, 6, 8} indicate the client request times. Table 2.4 shows the arrival times, the stack S after pairs are popped, the stack S after the new pair for the stream is added, and the client to which the new arrival will merge. F_h is 5 in this case.

Figure 2.9 shows the factor increase in total cost for the adaptive algorithm. The experiment used media of length 650 with mean client interarrival rate of 3.49. Clients arrived according to a Poisson process. The mean factor increases for the algorithms are as follows: 1.1024 for $R = 1$, 1.0773 for $R = 3.49$, and 1.0941 for $R = 5$. The adaptive algorithm using $R = 3.49$ achieves the smallest factor increase.

2.4 Conclusions and Future Work

Our work seeks to limit the server bandwidth necessary to serve clients requesting popular media from a media-on-demand system. Optimizing the bandwidth requires dictating the

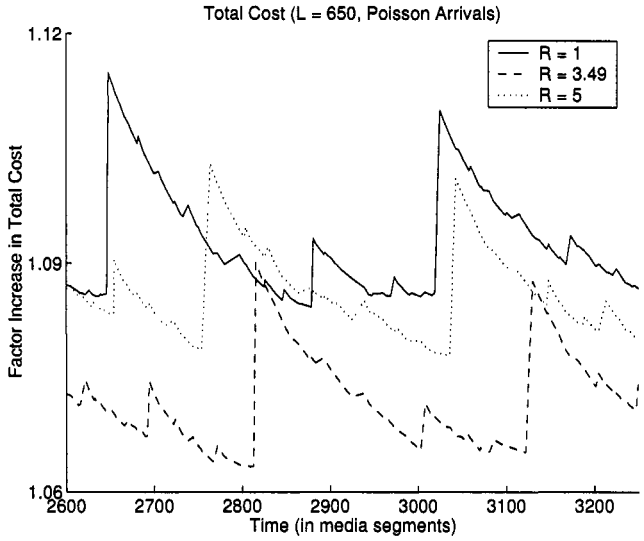


Figure 2.9: Simulation results for the adaptive discrete uniform demand algorithm using four different rates R. The simulation used media with length 650 and ran for 3250 media segments with clients generated from a Poisson process. The total number of clients was 932 over the 3250 media segments, for an average interarrival rate of 3.49.

stream merging schedules as clients arrive in the system. Data sent from the server to the clients consumes network bandwidth and server bandwidth, which becomes quite costly if the server gives each client a dedicated channel. Our contributions build on stream merging techniques that take advantage of existing streams of data in the system.

We have demonstrated and empirically compared the performance of new online stream merging algorithms based on the dyadic algorithm. We have shown how to optimize the dyadic algorithm parameters α and β in the continuous time response case. We extended and introduced two new online algorithms for the discrete time response case based on an optimal offline algorithm for discrete arrivals. Our algorithms achieve optimal total bandwidth when the number of arrivals is a multiple of the Fibonacci number chosen based on the media length L.

Future work includes gathering data to understand more accurately the client request intensity distribution, so we may tune our algorithms for realistic client request sequences. Our simulations in this chapter assume uniform time arrivals or arrivals generated from a

Poisson process, neither of which may accurately describe the client request distribution for a media-on-demand system. BitTorrent, a file distribution system that utilizes downloaders to service other downloaders, indicates that the number of clients grows just after a file is made available, reaches a peak, and then falls off exponentially [22]. Chesire *et al.* study the workload of streaming media and show that more than 50% of the overlapping streams could be served by multicast for at least 80% of the overlapping period [19]. Therefore, stream merging techniques could be quite effective for real workloads. Comcast now offers television shows and movies on demand [23]. As more customers use these on demand services, we can learn more about customer request distributions. The client request rate R may depend on time of day, time during the week, and time since introduction of the media. Future work includes simulations to better understand how to adapt the client request rate R. We may also need to adjust W, the window size, if the request rate fluctuates considerably between two adjacent windows. Another scenario for testing our algorithms includes a server with fixed bandwidth. In this scenario, we want to minimize both the total server bandwidth and the client waiting time. When a server has fixed bandwidth, a client may need to wait longer than a single media file segment until the server has sufficient bandwidth to serve the request. With limited server bandwidth, we may want to minimize the maximum bandwidth of the system instead of the total server bandwidth.

Chapter 3

MEDIA-ON-DEMAND WITH TIME-SHIFTING

With current media broadcast systems (such as radio and television), people who want to receive the broadcast must tune in as the broadcast is transmitted. For example, someone wishing to view a television program at 8 PM must start watching the program at that time. We propose a system that allows people to receive previously or currently broadcasted programs. In this system, a server continuously stores the live broadcasted stream and responds to clients who can request any part of the previous or currently broadcasted stream. Because the server bandwidth can quickly become a bottleneck when serving several clients, we seek to minimize the server bandwidth while satisfying all client requests.

A simple solution for satisfying client requests is to dedicate a transmission channel for each client at the time of its request. This requires available server bandwidth that scales linearly with the number of clients receiving simultaneous transmissions. Since this solution does not scale, we instead consider using stream merging techniques to reduce the overall server bandwidth. The live, continuous broadcast with time-shifting problem, originally proposed by Goshi [38], is a modification of the media-on-demand problem. In the original media-on-demand problem, the model used in previous chapters of this dissertation, clients always request the beginning of the media, receive the media in its entirety, and the media is of finite size. Jin and Bestavros analytically studied the problem where clients can request any part of the movie in the original media-on-demand system and their request wraps around from the end of the movie to the beginning [49]. They found that the total server bandwidth required to service N clients has order $\sqrt{N}$. Tan, Eager, and Vernon studied a similar problem where clients access intervals of media data, rather than the full file [69]. The time-shifting problem that we study differs from the problem with access intervals for a single file and from the standard media-on-demand model since we assume a single, live broadcast that has no beginning or end. We consider stream merging techniques for

the time-shifting problem because online stream merging algorithms have been shown both theoretically and experimentally to perform well in the original media-on-demand case [5], [6], [15], [16], [21]. There exists a polynomial time algorithm for optimal offline stream merging [8]. Although the offline algorithm for time-shifting would not be used in practice, it serves as a valuable benchmark for online algorithms. We consider finding an offline algorithm for the time-shifting model in this chapter.

The outline for the remainder of this chapter is as follows: we first define the optimal offline stream merging problem for live, continuous broadcasts [38]. After formalizing the problem, we show how the problem can be modeled using client embeddings in a rectilinear grid. We extend the rectilinear merge tree developed by Chan *et al.* where nodes and edges describe merge schedules of clients in the system [15], [16]. We then prove properties of rectilinear merge trees and introduce transformations that can be performed on optimal rectilinear merge trees before describing an algorithm to find the optimal rectilinear merge tree given a set of client arrivals.

3.1 Stream Merging with Time-Shifting Model

This section formally describes the stream merging problem for live, continuous broadcasts [38]. As in the original media-on-demand model, the system consists of a server, a set of clients, and a set of channels that can transmit data from the server to a set of clients.

The live stream is divided into segments of equal size where size is determined by the length of playback time. Therefore, media segments and time are in the same unit where it takes one time unit to play one segment of the media. For simplicity, we assume the live stream, called the *root stream*, starts at time $t = 0$ with segment 0 and continues forever.

The server continues to store and transmit a stream for the live, continuous broadcast. The server can transmit multiple streams simultaneously, one of which is always the root stream. The other streams transmit segments preceding the live broadcast.

Clients can receive two streams simultaneously, buffer the data from both streams, and playback buffered segments when necessary. Client receive bandwidth must be at least twice the playback rate in order for clients to receive two streams at once and have the segments

required for uninterrupted playback. We assume each client receives media segments until *after* it merges to the root stream. All clients arrive at time $t = 0$ or later. Each client c is represented by the pair $(t(c), f(c))$.

Definition 3.1.1 *$t(c)$ is the time that client c arrives $[t(c) \geq 0]$.*

Definition 3.1.2 *$f(c)$ is the first media segment that client c requests $[f(c) \leq t(c)]$.*

For example, if a client arrives at time 16 requesting segment 7, it is represented by (16, 7). If there are other clients that arrive at time 16 requesting segment 7, we treat the entire set of clients as a single client in the system.

The server must create a feasible solution for each client c such that client c will receive all segments of the broadcast at or before the time necessary for playback. Thus, each client will be able to play the broadcast without interruption. We call feasible solutions *merge schedules* which dictate to whom and when clients merge in the system.

Figure 3.1 shows a set of clients and their merge schedules in the stream merging with time-shifting case. The clients are $a = (15, 0)$, $b = (16, 7)$, $c = (18, 7)$, $d = (20, 7)$, $e = (23, 14)$, $f = (25, 23)$, $g = (26, 24)$, and $h = (30, 27)$. Each client is positioned according to its arrival time and first segment. Time is indicated along the bottom axis. The root stream starts at time 0 and broadcasts each media segment in turn. The streams are represented by diagonal lines. To reduce clutter on the figure, only the final merge (dotted vertical line) from one stream to another is shown. Clients can merge to their targets before the end of the stream, if the clients have sufficient data. For example, client f can merge to the live stream after receiving segment 24 from stream f, while client g merges to the root stream after receiving segment 25 from stream f.

The diagram of Figure 3.1 can also be represented by a merge tree, where each node represents a client. The root stream is the root of the tree. A client c is a child of client c_p if and only if c merges directly to c_p. The merge schedule for a client is given by the path from the client to the root of the tree. Figure 3.2 shows the merge tree for the clients in Figure 3.1.

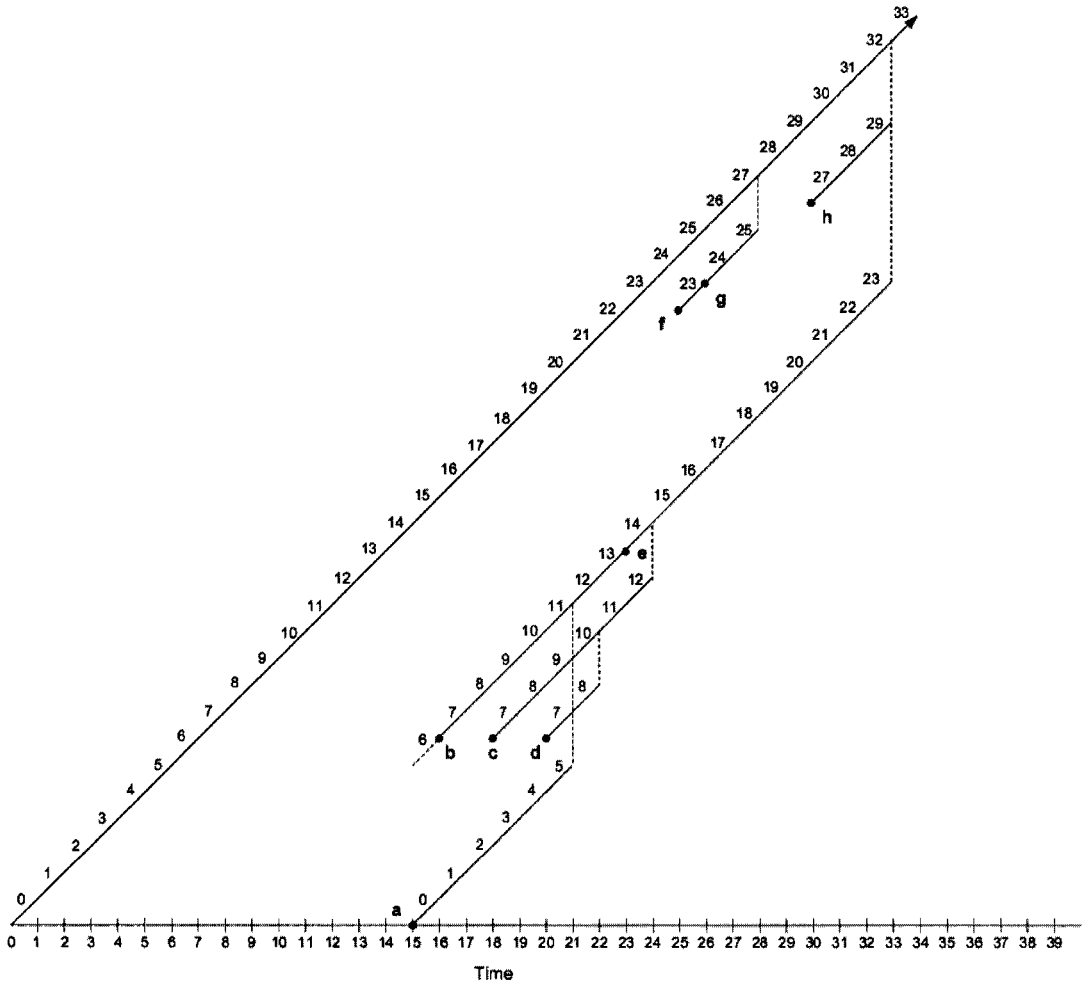


Figure 3.1: Stream merging with time-shifting example. The clients are: $a = (15, 0)$, $b = (16, 7)$, $c = (18, 7)$, $d = (20, 7)$, $e = (23, 14)$, $f = (25, 23)$, $g = (26, 24)$, and $h = (30, 27)$. Solid lines represent streams and vertical dotted lines show the final client merge along the stream. Clients can merge to their target earlier. For example, client f can merge to the live stream after it receives segment 24 from stream f. The diagonal dashed line indicates that a stream is started before an actual client arrival.

3.1.1 Optimal Stream Merging

We define optimality in terms of total server bandwidth (in media segment units) needed to satisfy all client requests. Because the time-shifting model assumes a continuous root stream, we consider optimality over a finite span of time and set of clients. We can choose

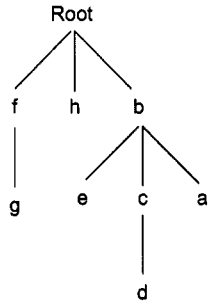


Figure 3.2: The standard merge tree representation for the clients shown in Figure 3.1. Each client's schedule is determined by its ancestors in the tree.

to include or exclude the cost of the root stream in the optimal stream merging solution, since the root stream is a fixed cost. In this chapter we include the cost of the finitely-sized root stream in our model. Optimal stream merging with time-shifting applies only to the offline case, where all client arrivals and requests are known over a finite span of time.

Definition 3.1.3 *The* optimal stream merging with time-shifting problem *is: Given a set of clients* $C = \{c_0, c_1, \ldots, c_{n-1}\}$, *what is the merge schedule producing the smallest server bandwidth cost?*

3.2 On a Rectilinear Grid

Instead of using a merge tree (as shown in Figure 3.2), where parent nodes are merge targets for their children, we instead use rectilinear merge trees to model the merge patterns. Chan *et al.* first used rectilinear merge trees for the original media-on-demand model, where clients always request the first media segment [16]. We use a variant of their rectilinear merge tree to capture merge events and the lengths of client streams. Below we show that optimal merge schedules are in the form of rectilinear merge trees, where the trees are embedded in rectilinear grids. First, we show how clients are embedded in the grid. Then we describe the representations for client mergers and further show that modeling the time-shifting problem as a tree in the rectilinear grid captures the server bandwidth cost.

3.2.1 *Client Embedding in the Rectilinear Grid*

We embed clients and streams in a rectilinear grid, where clients lie at grid intersections and streams are horizontal and vertical lines aligned with the grid. Each grid unit represents one time unit or, equivalently, one media segment. Time progresses from right to left and from top to bottom in the grid. The root stream is indicated by a solid horizontal line from right to left in the embedding and all clients are positioned on or above the root stream. A grid position is not represented by Euclidean coordinates, but rather the client pair $(t(c), f(c))$ corresponding to the grid position. Each client's position is based off the start of the root stream:

Definition 3.2.1 *A* client arrival c *is indicated by a point on the grid at position* X *units to the left of the start of the root stream and* Y *units above the root stream, where* $X = 2t(c) - f(c)$ *and* $Y = t(c) - f(c)$.

Figure 3.3 shows the positioning of clients in the rectilinear grid. The clients in the figure are $a = (15, 0)$, $b = (16, 7)$, $c = (18, 7)$, $d = (20, 7)$, $e = (23, 14)$, $f = (25, 23)$, $g = (26, 24)$, and $h = (30, 27)$.

If a client c requests segment 0 and starts at time $t(c)$, then it is placed at the grid position $2t(c)$ units to the left and $t(c)$ units above the root stream. All clients requesting segment 0 lie on the *zero line*, which is shown in Figure 3.3. Back in Euclidean space, the zero line has slope $-\frac{1}{2}$ and intersects with the start of the root stream. For example, in Figure 3.3, client a requests segment 0, so it lies on the zero line.

All client arrivals are positioned on or below the zero line. For each client, $X = 2t(c) - f(c)$ and $Y = t(c) - f(c)$. If a client is positioned above the line, X must be less than $2Y$. From the definition, $X \geq 2Y$ so all clients lie on or below the line. Furthermore, all clients lie on or above the horizontal line designating the root stream.

Clients lying on the same horizontal line have equal $Y = t(c) - f(c)$ values. These clients are behind the transmission of the root stream by Y segments. For example, client $b = (16, 7)$ in Figure 3.3 is 9 units above the root stream, meaning that it needs to receive 9 segments before merging directly to the root stream. Clients b and e lie on a single

horizontal line, and they both have Y values of 9. The $t(c)$ and $f(c)$ values each increase by 1 when going one unit to the left.

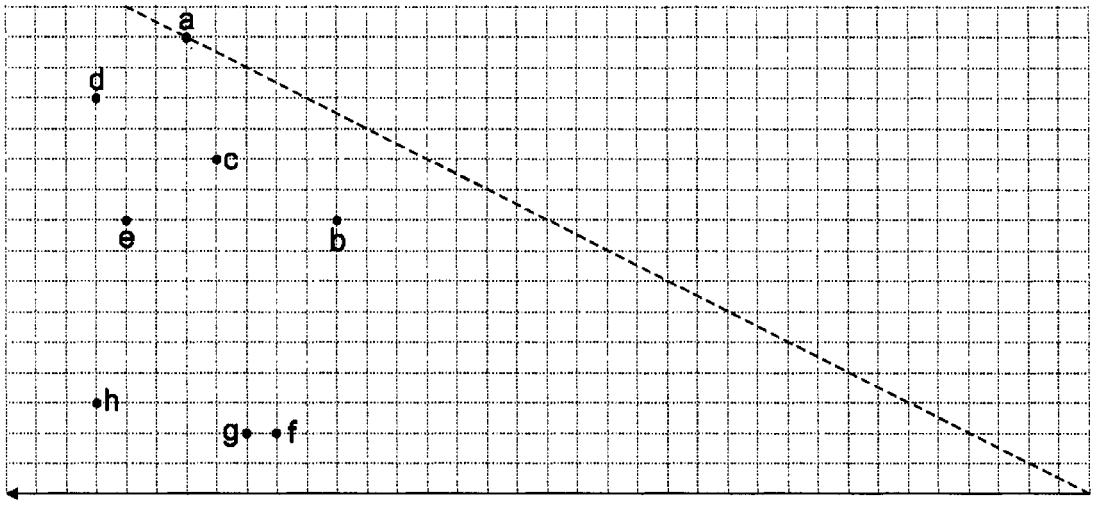


Figure 3.3: Embedding of eight clients (black points) embedded in the rectilinear grid (dotted lines). The root stream (solid line) starts at the right side of the figure and continues to the left. The clients are: $a = (15, 0)$, $b = (16, 7)$, $c = (18, 7)$, $d = (20, 7)$, $e = (23, 14)$, $f = (25, 23)$, $g = (26, 24)$, and $h = (30, 27)$.

3.2.2 Representing Merge Schedules

We connect client arrivals in the grid with horizontal and vertical lines to form merge schedules. Figure 3.4 shows a possible merge schedule for clients in the top of the figure. Each line represents a stream of transmitted data. Dashed horizontal lines indicate streams that start before an actual client arrival. Stream lines may cross over one another, so the merge schedule is not necessarily planar. For example, the vertical segment under a in Figure 3.4 crosses over the horizontal segment to the left of c. Numbers along edges indicate segments being transmitted by the stream. A triangle indicates a junction between two streams, where clients listening to the vertical stream merge with clients receiving the horizontal stream.

Time is represented by rectilinear distance. Time progresses from right to left and from top to bottom. Each rectilinear unit is equivalent to the unit of time it takes to transmit

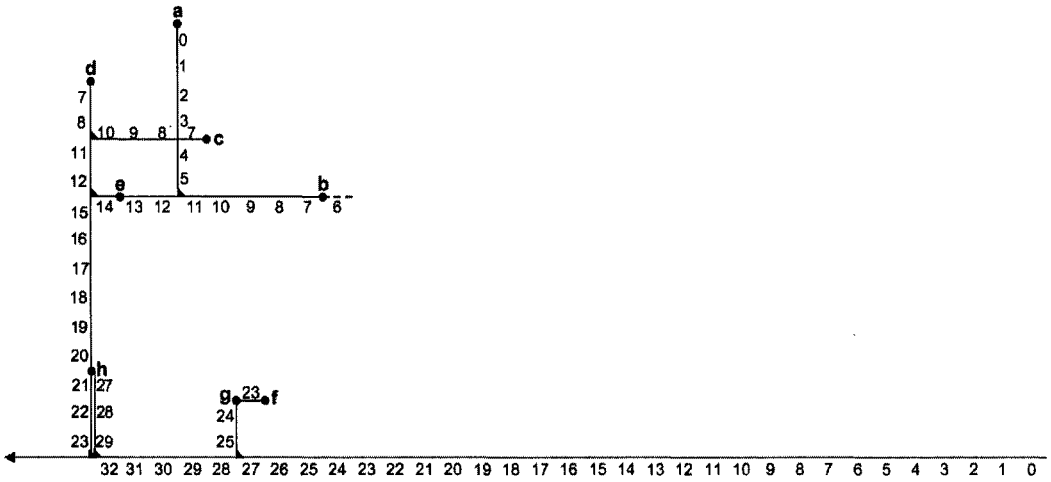


Figure 3.4: Shows the merge schedule for eight clients: $a = (15, 0)$, $b = (16, 7)$, $c = (18, 7)$, $d = (20, 7)$, $e = (23, 14)$, $f = (25, 23)$, $g = (26, 24)$, and $h = (30, 27)$. Lines represent streams and label numbers indicate the segments transmitted along streams. The dashed line indicates a stream starting before b's arrival. Junctions are indicated by triangles. Notice that the vertical line from a crosses the horizontal line to c (a crossover), so the schedule is not necessarily planar.

one media segment. In Figure 3.4, f arrives at time 25 and g arrives at time 26. Notice that g is one unit to the left of f. The junction below g is 2 rectilinear units away, so the junction happens at time 28.

Clients receiving vertical streams are receiving two streams simultaneously. They are receiving the stream represented by the vertical line and the stream represented by the horizontal line connecting to the junction at the bottom of the vertical line. In Figure 3.4, when client d arrives it receives segments 7 and 8 from its own stream and segments 9 and 10 from the stream started at client c. The junction indicates that d merges to c just before segment 11 is sent via c's stream. At this point, the stream to d terminates, saving server bandwidth.

Clients receiving a single stream are represented by *horizontal lines*. In Figure 3.4, client c is receiving a single stream of segments 7 through 10 before client d merges to c. We name the following stream relationships:

Definition 3.2.2 *A* junction, *represented by a triangle, is the merge point of the vertical stream from above to the horizontal stream to the right. A junction may be positioned at a client arrival. There could be multiple junctions at the same position as shown by the two junctions under h with the root stream in Figure 3.4.*

Definition 3.2.3 *A* corner *connects the left endpoint of a horizontal line with the top endpoint of a vertical line. There may or may not be a client positioned at the corner connection. Client g in Figure 3.4 is at a corner.*

Definition 3.2.4 *A* continuation *is a horizontal line with more than one client arrival along the line. In Figure 3.4, there is a continuation of the stream through client arrival e.*

Definition 3.2.5 *A* crossover *is a horizontal line that intersects a vertical line without a junction marking. The two streams represented by these lines do not interact with one another. The line under client a and the line to the left of c form a crossover in Figure 3.4.*

Definition 3.2.6 *A* premature start *is a dashed horizontal line that indicates a stream is started before a client arrival. Figure 3.4 shows a premature start lasting 1 segment before the arrival of client b.*

Merge Schedule Description

Figure 3.5 shows the set of clients listening to each stream in the system. For simplicity, the streams are named for the first client that starts listening to that stream. For example, the vertical line from client a will be called Stream A. The schedule can be interpreted by following the streams for each client. Below is a textual description of the merge schedule depicted in Figures 3.4 and 3.5.

Client a arrives at time 15 requesting segment 0. Client a receives segments 0 through 5 from Stream A. Meanwhile, a premature stream for b is started at time 15 and transmits segment 6. Client a receives segments 6 through 11 from Stream B. At time 21, Stream A terminates and a listens solely to Stream B for segments 12 - 14. At time 24, a starts listening to the root stream in addition to Stream B. Client a gets segments 15 - 23 from

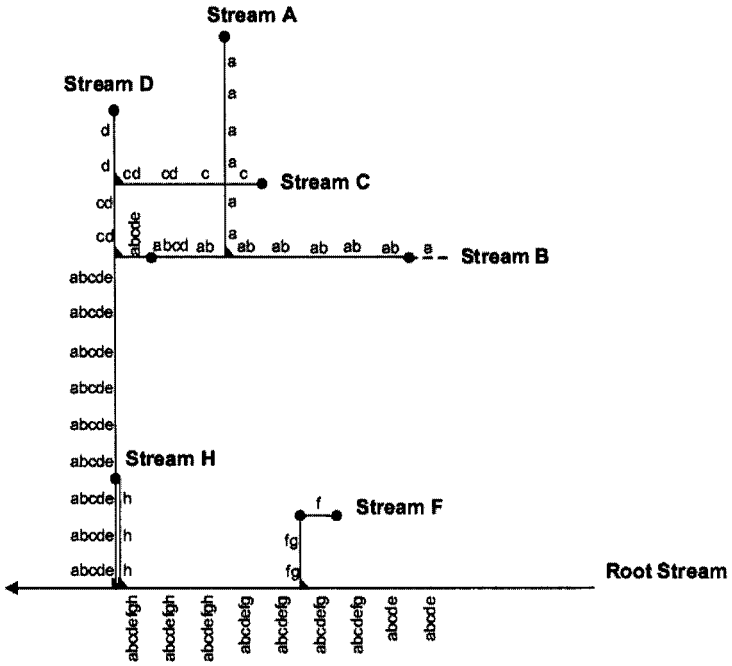


Figure 3.5: Shows the streams for eight clients: $a = (15, 0)$, $b = (16, 7)$, $c = (18, 7)$, $d = (20, 7)$, $e = (23, 14)$, $f = (25, 23)$, $g = (26, 24)$, and $h = (30, 27)$. Lines represent streams and the labels indicated which clients are listening to the streams.

Stream B and segments 24 - 32 from the root stream. At time 33, a merges to the root stream.

Client b arrives at time 16 requesting segment 7. It receives segments 7 - 14 from Stream B. At time 21, b starts listening to the root stream in addition to Stream B. It receives segments 15 - 23 from Stream B and segments 24 - 32 from the root stream. At time 33, Stream B terminates and b merges to the root stream.

Client c arrives at time 18 requesting segment 7. Stream C is initiated at time 18 and receives segments 7 - 10 from Stream C. At time 22, c starts listening to Stream B and receives segments 11 - 12 from Stream C and segments 13 - 14 from Stream B. At time 24, Stream C terminates. c gets segments 15 - 23 from Stream B, gets segments 24 - 32 from the root stream, and merges to the root stream at time 33.

Client d arrives at time 20 requesting segment 7. Stream D is initiated and d receives

segments 7 - 8 from Stream D while listening to Stream C where it gets segments 9 - 10. At time 22, Stream D is terminated. Client d gets segments 11 - 12 from Stream C while receiving segments 13 - 14 from Stream B. At time 24, Stream C terminates and d starts listening to the root stream. d gets segments 15 - 13 from Stream B and segments 24 - 32 from the root stream. It merges to the root stream at time 33.

Client e arrives at time 23 requesting segment 14. A new stream for client E is not initiated. Instead e starts listening immediately to Stream B. e gets segment 14 from Stream B. At time 24, e starts listening to the root stream in addition to Stream B. It gets segments 15 - 23 from Stream B and segments 24 - 32 from the root stream before merging to the root stream at time 33.

Client f arrives at time 25 requesting segment 23. Stream F is initiated and f gets segment 23 from Stream F. At time 26, f starts listening to the root stream. f gets segments 24 - 25 from Stream F and segments 26 - 27 from the root stream before merging to the root stream at time 28.

Client g arrives at time 26 requesting segment 24. A new stream for g is not initiated. Instead, g starts listening immediately to Stream F. When g arrives, it listens to the root stream in addition to Stream F. It gets segments 24 - 25 from Stream F and segments 26 - 27 from the root stream before merging to the root stream at time 28.

Client h arrives at time 30 requesting segment 27. Stream H is initiated and client h listens to Stream H and the root stream. It gets segments 27 - 29 from Stream H and segments 30 - 32 from the root stream. Client h merges to the root stream at time 33.

3.2.3 Merge Schedules as Rectilinear Trees

We have shown how to embed and connect client arrivals in a rectilinear grid to form merge schedules. Our goal is to find the optimal schedule for a set of client arrivals in the time-shifting model. We define a *complete merge schedule* as a merge schedule where every client eventually merges to the root stream. Here we show that every complete merge schedule is in the form of a rectilinear tree, where each junction satisfies the elbow property.

Lemma 3.2.1 *In complete merge schedules every client arrival lies on a path consisting of*

lines going down and/or to the left toward the root stream.

Proof: Every client eventually merges to the root stream in a complete merge schedule. (We assume every client continues the broadcast at least until it merges to the root stream). There must be some path from each client to the root stream. Assume to the contrary that the there is a path from a client c to the root stream containing a line going to the right. Then the client is listening to a stream going backward in time, which is impossible. Now assume the path from c to the root stream contains a line going up. c must be receiving data going backward in time from the vertical stream. Therefore, c must lie on a path consisting of lines going left and/or down. ∎

Lemma 3.2.2 The Elbow Lemma: *Let (i, j) be a client arrival, junction, or corner point connected by a downward vertical line in a merge schedule. Let $(i + d, j + 2d)$ be the first junction connected to and under (i, j), where $d > 0$ indicates the vertical distance between (i, j) and $(i + d, j + 2d)$. Then the horizontal line coming from the right of $(i + d, j + 2d)$ must be at least d long. All junctions must have a horizontal entry from the right that is at least as long as the vertical entry from above.*

Proof: Because (i, j) is on a vertical line, it represents a client or a set of clients listening to two streams simultaneously. The set of clients listen to segment j starting at time i. Also, the set listens to the horizontal stream starting at time i, so the horizontal stream must start at or before time i. The length of the horizontal stream must be at least d units long. See Figure 3.6 for the diagram. ∎

To ensure that horizontal lines meeting a vertical line at a junction has sufficient length, premature starts may be necessary. The dotted line in Figure 3.4 is a stream extended backward in time from client $b = (16, 7)$ to position $(15, 6)$. The stream for client b starts one unit of time before b's arrival. There might be multiple vertical lines forming junctions with a single horizontal line. There are two junctions on the horizontal line with client b in Figure 3.4.

The point (i, j) on a vertical line may form a junction with any point below (not necessarily the closest junction point) For example, in Figure 3.4, the junction under client d

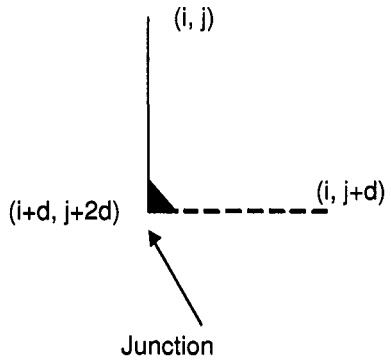


Figure 3.6: Shows the restriction on horizontal lines meeting at a junction: The horizontal line (dashed) must be at least as long as the vertical segment (solid). The point (i, j) could be a client arrival, another junction, or a corner point. The horizontal line may need a premature start to satisfy the elbow property.

could form a junction with client h. Instead, the junction under client d forms a junction with the root stream to save bandwidth. Therefore, there are two vertical lines under client h, one representing the stream for the junction under client d and one representing the stream for h. The *elbow property* requires that all junctions satisfy the Elbow Lemma.

Definition 3.2.7 *The* elbow property *is that all junctions satisfy the requirements of the Elbow Lemma. That is, all junctions must have horizontal line entries at least as long as vertical line entries.*

Lemma 3.2.3 *In every complete merge schedule, each junction or client arrival (other than the left-most junction with the root stream) has exactly one line going down or to the left.*

Proof: Assume there are no lines leaving a junction or client arrival to the left or below in the merge schedule. Then, the set of clients at the junction never merges with the root stream, which violates Lemma 3.2.1. Now assume, there are two lines leaving a junction or client arrival, one line horizontally to the left and one line vertically below. (Note: Since all clients must be on paths that consist of lines going left and down, we do not need to consider the case of more than two lines leaving a junction or client arrival.) A horizontal line leaving a junction or client arrival indicates that the set of clients is listening to one

stream. A vertical line leaving a junction or client arrival indicates that the set of clients is listening to two streams. Thus, the set of clients is listening to a total of three streams, which violates the model condition that each client can receive two streams simultaneously. ∎

Definition 3.2.8 *A* rectilinear merge tree *is a rooted tree connecting all client arrivals positioned on a rectilinear grid such that all edges of the tree, aligned vertically or horizontally with grid lines, are oriented away from the root. Furthermore, the tree satisfies the elbow property.*

Theorem 3.2.4 *Every complete merge schedule on the rectilinear grid must be in the form of a rectilinear merge tree where the root of the tree is at the left-most junction with the root stream.*

Proof: Since all clients and junctions have exactly one line leaving to the left or below, every client or junction has exactly one parent client or junction. All clients are connected to the root stream, so there is a path from each client to the root stream (and thus to the root of the tree). The final merge to the root stream indicates the root of the rectilinear tree, since there are no other client arrivals or junctions that are located to the left of the left-most junction. ∎

Property 3.2.5 *The* total server bandwidth *(in number of media segments) of a complete merge schedule represented by a rectilinear merge tree is equal to the total length of horizontal and vertical lines.*

In Figure 3.4, the root stream terminates after sending segment 32, since the left-most clients merge to the root stream just before the root stream transmits segment 33. Thus, the root of the rectilinear merge tree is at grid location $(33, 33)$.

All horizontal lines represent streams in time going from right to left, so one unit of line length is exactly equal to the server sending one media segment. Vertical lines represent clients listening to two streams. As Figure 3.6 shows, the client set at (i, j) receives segments j through $j+d-1$ from its own stream and receives segments $j+d$ through $j+2d-1$ from

the horizontal stream at the junction (dashed line), so it will merge with the horizontal stream at time $i + d$, at which point it will request segment $j + 2d$. This is exactly the point at which the stream for client (i, j) ends. Thus, the server bandwidth for the vertical streams is exactly the length of the vertical lines. Notice that a junction could have more than one vertical segment, as shown by the left-most junction at the root stream in Figure 3.4. The total cost for the merge schedule shown in Figure 3.4 is 71, which is the total length of the horizontal and vertical lines.

3.2.4 Comparison to Chan et al.*'s Rectilinear Trees*

Chan and others used a rectilinear tree for competitive analysis of their connector algorithm to the optimal stream merging algorithm in the original receive-two case where clients always request the beginning of the media [16]. Our time-shifting rectilinear trees connect clients positioned on and under the zero line. Since the original rectilinear merge tree models the original media-on-demand case, clients are positioned only on the zero line. Our trees also have explicit junctions, since a vertical segment may not form a junction with the closest client below. The junctions may involve more than two streams, with multiple streams coming in from above. The original trees proposed by Chan *et al.* always have binary junctions, with exactly one vertical entry and one horizontal entry. Our trees may have crossovers, so our trees are not necessarily planar, as are the trees of Chan *et al.* Finally, our merge trees must satisfy the elbow property at each junction and may have premature starts. The original rectilinear merge trees always satisfy the elbow property, since client arrivals are restricted to the zero line. Thus, in original rectilinear trees, each horizontal segment is at least twice as long as the vertical segment.

3.3 Optimal Rectilinear Merge Trees

We showed above that all complete merge schedules must be rectilinear trees that satisfy the elbow property at every junction. Are there other properties that are guaranteed to be satisfied for complete rectilinear merge trees with the optimal server bandwidth? Recall that our goal is to minimize total server bandwidth, so an optimal merge tree for a given set

of clients must have the minimum total cost for those clients. Also recall that we assume all clients stay in the system until after they merge to the live, continuous broadcast. We show in this section that any rectilinear merge tree can be transformed such that each horizontal line (continuous through client arrivals and junctions) contains at least one client arrival. A second transformation ensures that each continuous vertical line has a client arrival at its top-most point. With these transformations, we can limit the number of possibly optimal rectilinear merge trees for a finite set of clients and perform an exhaustive search for the rectilinear merge tree with the optimal cost.

3.3.1 Properties of Optimal Rectilinear Merge Trees

In this section we introduce two properties of optimal rectilinear merge trees. The first property limits the length of streams with premature starts and the second property ensures there are no premature starts for vertical streams.

Property 3.3.1 *For each junction J containing a premature start, the premature start is no longer than the length needed to satisfy the elbow property for each junction co-located with J.*

Proof: Assume we have an optimal rectilinear merge tree with a junction J containing a premature start. Also assume the premature start creates a horizontal stream H with length k units and the longest vertical segment V (between any junction at J and the client arrival above, the corner above, or the junction above) is d units where $d < k$. Then, the clients along V will start listening to H d time units before the merge at J's grid position. Therefore, only the leftmost d units of H are needed. We have two cases with respect to the positioning of the premature start to H:

1. Assume the premature start ends $\geq d$ units to the right of J. Then we can remove the entire premature start, since H is sufficiently long enough without the premature start. Removing the premature start reduces the total server bandwidth, so we have a contradiction with the assumption that the tree is optimal.

2. Assume the premature start ends $< d$ units to the right of J. Then we can remove the rightmost $k - d$ units of the premature start and H will have length d which is long enough to satisfy the elbow property. Since $k > d$, we remove at least one unit of length, decreasing the total server bandwidth.

■

Property 3.3.2 *The top endpoint of any vertical segment must be a client arrival, a corner, or a junction. There are no premature starts in the vertical direction.*

Proof: Assume we have an optimal rectilinear merge tree with a vertical segment whose top endpoint is not a client arrival, a corner, or a junction. The remaining option is that the endpoint is the start of a stream. There are no clients listening to this vertical stream from the endpoint down to the first junction or client arrival. Thus, we can remove the portion of the vertical segment between the endpoint and the first client or junction below, which decreases the cost of the rectilinear merge tree. This creates a contradiction with the assumption that the tree is optimal. ■

3.3.2 Rectilinear Merge Tree Transformations

Here we show that any rectilinear merge tree can be transformed to satisfy certain properties. Furthermore, these transformations do not increase the total server bandwidth cost and in some cases they reduce the total server bandwidth cost. These transformations will prove helpful when determining the possible structures for optimal rectilinear merge trees.

Definition 3.3.1 *A* continuous horizontal line *of a rectilinear merge tree is a continuous horizontal line through client arrivals and junctions.*

Property 3.3.3 *Any rectilinear merge tree can be transformed (without incurring additional cost) to satisfy the following: Except for the root stream, every continuous horizontal line must contain at least one client arrival.*

Proof: Assume we have a rectilinear merge tree with a horizontal line H (other than the root stream) without any client arrivals (the entire horizontal line is a premature start).

There are four cases for H, as shown by dashed lines in Figure 3.7. There must be an endpoint on the left, since H is on a path to the root stream and the root stream is below H. The endpoint to the left of H could be a corner or a junction. Cases (a) and (c) in Figure 3.7 have corners to the left and cases (b) and (d) have junctions to the left. We show how to transform each case without increasing the total cost of the tree.

1. Case (a) [SINGLE CORNER ON LEFT]: Assume the horizontal line H meets a corner to the left and H does not contain any client arrivals or junctions. Let k denote the length of H and let d denote the vertical distance from the corner to the first client arrival and/or junction below. Because there are no clients that use H, we can remove H completely, decreasing the total cost by k. Also, we can remove the top d units of the vertical segment, since there are no clients using this part of the vertical stream. We have decreased the total cost in this transformation.

2. Case (b) [SINGLE JUNCTION ON LEFT]: Assume the horizontal line H meets a junction J to the left and there are no other junctions or client arrivals along H. Let d be the distance above J to the first junction and/or client arrival. Then H must be at least d in length to satisfy the elbow property. We can remove H and append it to the right of the horizontal stream of the first junction J' below J. (We know that such a junction J' exists since there is always a root stream below.) Let k be the distance from J to J'. The horizontal stream to the right of J' must be at least k long (and possibly longer). By adding the premature start of length d to the horizontal stream of length k, we ensure the elbow property holds at J'. Also, if the horizontal stream at J' is longer than k, then we can add fewer than d units to the right, which decreases the total cost of the rectilinear merge tree.

3. Case (c) [CORNER ON LEFT, MULTIPLE JUNCTIONS]: Assume the horizontal line H meets a corner to the left and contains one or more junctions along H. Call the junctions $J_1, J_2, \ldots, J_k$. Find the closest junction and/or client arrival above H along the vertical segments of J_1 through J_k. Let J_i be the junction containing the vertical segment with this closest point or junction and let d be the distance to this

closest point or junction. Move H up d units. (We know we are not moving H through other junctions or clients, since we chose the closest junction or client which is d units above.) By moving H up d units, we remove Kd units of length from each of the vertical segments of the junctions $J_1, \ldots, J_k$. Also, for each unit that H moves up, we can remove the rightmost unit of H. (The rightmost unit is no longer necessary since the vertical segments of the junctions to H are now one unit shorter.) Thus, we can remove d units from the right side of H. In total, we have removed $(K+1)d$ units. Because there is a corner to the left, we must add d units to the vertical segment of the corner. Therefore, we remove Kd units in the rectilinear merge tree, where $K > 0$, so we have decreased the total cost of the rectilinear merge tree.

4. Case (d) [JUNCTION ON LEFT, MULTIPLE JUNCTIONS]: The argument is similar to case (c). We now assume there is a junction on the left side of H. We again move H up to the first client or junction above that lies along vertical segments of junctions with H. With K junctions along H (not including the leftmost junction), we remove Kd in total length vertically and remove the rightmost d units of H. The d vertical units from the leftmost junction were included in the original structure, so we remove $(K+1)d$ in total length.

In all four cases, we preserve all paths from clients to the root stream, since H is never moved past a junction or client arrival above. We showed that in all four cases, we can either transform the tree to preserve the total bandwidth cost or we can transform the tree to lower the bandwidth cost. Therefore, all rectilinear merge trees can be transformed such that every horizontal line contains a client arrival. Also note that optimal rectilinear merge trees cannot have horizontal lines in cases (a), (c), and (d) since the transformation definitely reduces the overall server bandwidth. ∎

The next transformation alters the placement of vertical lines in rectilinear merge trees.

Definition 3.3.2 *A* continuous vertical line *of a rectilinear merge tree is a continuous vertical line through client arrivals and junctions. Because there could be multiple junctions*

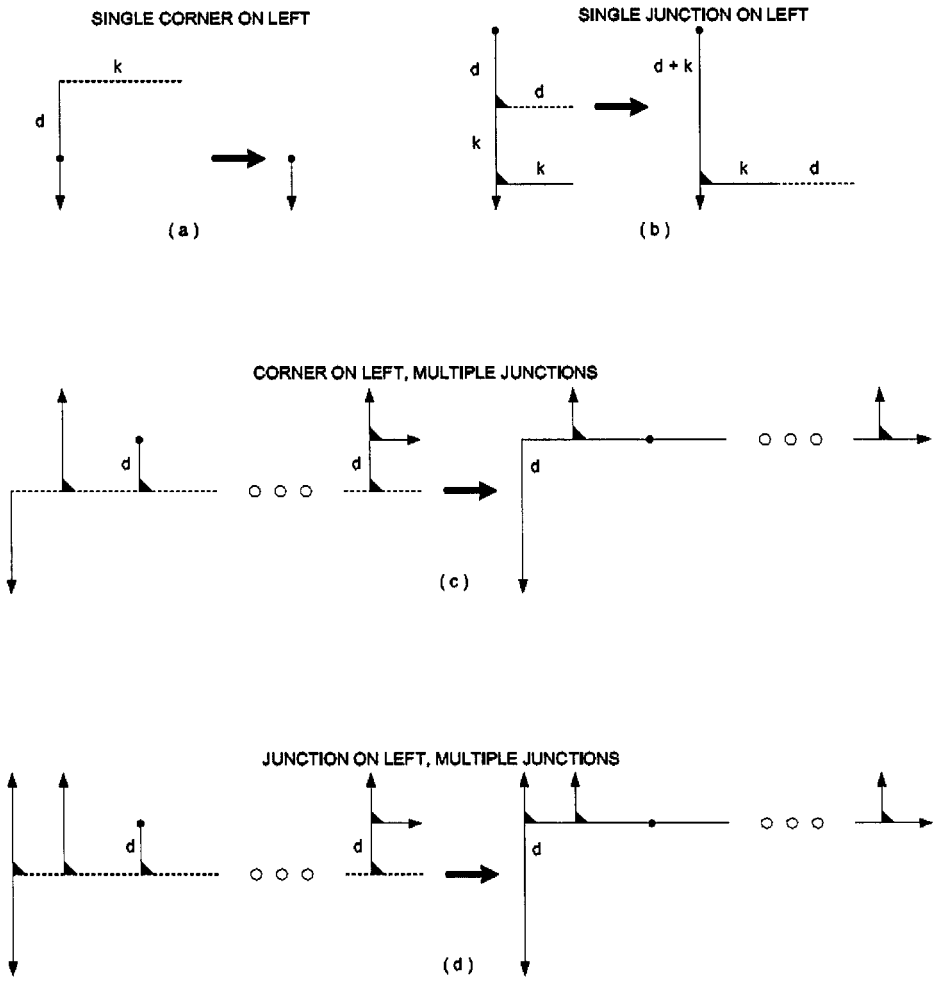


Figure 3.7: Shows the four possible cases of a horizontal line (dashed) without client arrivals. Case (a) shows a horizontal line meeting at a single corner. Case (b) shows a horizontal line meeting at a single junction. Case (c) has a corner on the left endpoint of the horizontal line with several junctions. Finally, case (d) has a junction at the left endpoint with one or more junctions on the line. The unfilled circles represent the continuous horizontal line, which could have additional junctions. In each case, we can transform it to the diagram shown to the right of the arrow, without the risk of increasing the total cost for the rectilinear merge tree.

at a single grid location, there could be multiple continuous vertical lines continuing above a single vertical line in the grid.

A junction J could have several continuous vertical lines above. Consider an arbitrary junction J. At junction J, there is a single vertical segment above which forms the junction. This vertical segment is the first piece of every continuous vertical line above J. If this vertical segment ends at a grid position with K co-located junctions, then there are K continuous vertical lines above J for each of the vertical segments of the K junctions. This continuous upward through the rectilinear merge tree.

Property 3.3.4 *Any rectilinear merge tree can be transformed (without incurring additional cost) to satisfy the following: For each junction J, there is at least one continuous vertical line above J such that the top endpoint is a client arrival.*

Proof: Assume we have a rectilinear merge tree with a junction J such that there is no client arrival at the top endpoint of one of its continuous vertical lines. The continuous vertical lines must end with the beginning of a stream or a corner. If one of the continuous vertical lines ends with the beginning of a stream, we can remove the top section of stream until the first client arrival or junction below, which reduces the total cost of the rectilinear merge tree. We assume that there are corners at the top of each of J's continuous vertical lines. We have two cases:

1. Case 1 [CLIENT ALONG A VERTICAL LINE]: There is at least one client arrival along V where V is one of the continuous vertical lines above J and there is no client arrival at the top endpoint (corner) of V. Figure 3.8 shows an example scenario. Let c be the top-most client arrival on V. Since c is not at the top endpoint of V, c must be located at another junction point J' above J. Apply the transformation in case (2) below at J'. The transformation will move the vertical segment above J' to the right (since there were no other client arrivals at junctions above c), so c will be at a corner to the left of J'. Part (b) of Figure 3.8 shows the result of the transformation.

2. Case 2 [NO CLIENTS ALONG VERTICAL LINES]: There are no client arrivals on any continuous vertical line above the junction J. Figure 3.9 shows an example

scenario of this case. Choose one of the continuous vertical lines above J and call it V. Find the first junction and/or client arrival to the right of V. (Note: There could be multiple junctions at the same location as J and all clients and junctions along paths for different junctions positioned at J are ignored.) Let d be the distance to the closest client and/or junction to the right of V. We move V d units to the right. If there are K junctions along V (excluding junction J), then we remove Kd units in length. Because the junctions are now d units to the right of their original locations, we may need to add d units of horizontal premature streams to the right of the horizontal lines for each junction. We may not need to add all d units for each junction (if the horizontal streams are already long enough), which reduces the total cost. We also remove d units from the horizontal stream at the top corner of V, which we add to the right of the horizontal line at J, since J is moved d units to the right. This transformation is shown in part (b) of Figure 3.9. After the transformation, there is a continuous vertical line using junction J_2 from J to an endpoint with a client arrival. The structure in part (b) can be transformed once again, since J_1 does not have a continuous vertical line that ends at a client arrival. Part (c) of Figure 3.9 shows the result of transforming J_1. (Note: Because the client arrival at the endpoint of the vertical line above J_2 is part of a path using a a different junction, we can ignore this client arrival when searching for the closest client or junction to the right of J_1.) This transformation does not increase the total cost of the rectilinear tree since we remove $Kd + d$ units and add up to $Kd + d$ units in the transformed structure.

In both cases, we ensure that every client arrival still has a path to the root stream since we never move V to the right past one of the junctions or clients that eventually joins at the junction J. We showed that the rectilinear merge tree can be transformed such that the total cost is preserved or there is a reduction in the total cost. Therefore, each junction has a continuous vertical line above with an endpoint at a client arrival. ∎

Property 3.3.5 *Any rectilinear merge tree can be transformed (without incurring additional cost) to satisfy the following: Every corner contains a client arrival at the corner point.*

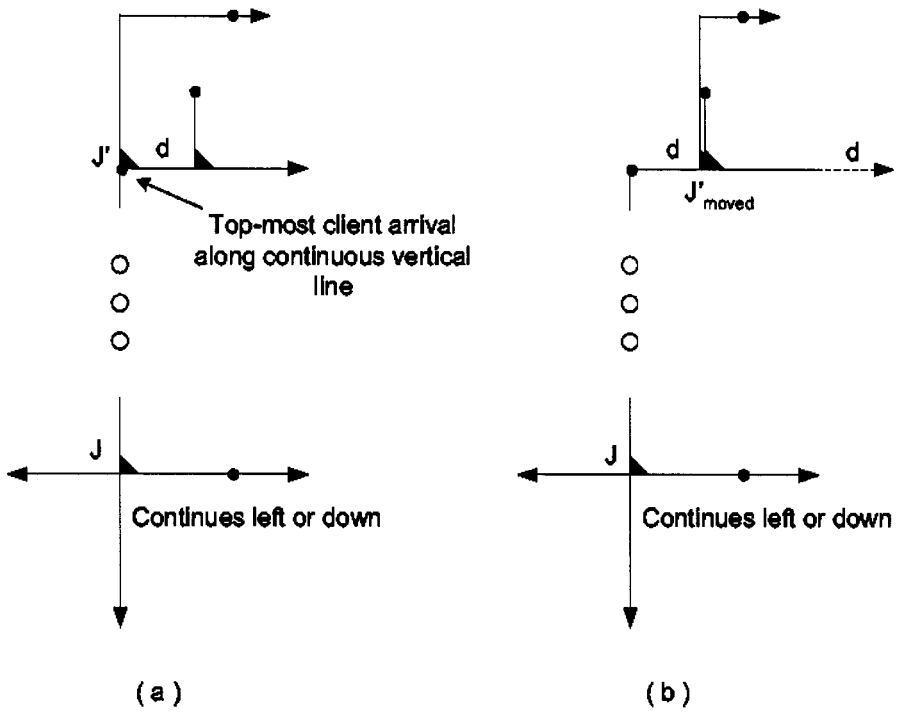


Figure 3.8: Shows a transformation such that the junction J has a continuous vertical line above that ends in a client arrival. Part (a) shows an example where a vertical line above junction J contains a junction J' with a client arrival and does not contain a client arrival at the top endpoint of the continuous vertical line. Pat (b) shows junction J' moved to the right, creating a client arrival at the corner above J.

Proof: Assume a rectilinear merge tree has a corner without a client arrival at the corner. Because the root stream is below all corners, there must be a junction J under the corner connected to the vertical segment of the corner. This junction J does not have a client arrival at the top endpoint of its only continuous vertical line. We can transform the junction J via Property 3.3.4 so there is a client arrival at the top of the continuous vertical line. Thus, every optimal rectilinear merge tree can be transformed such that each corner has a client arrival at the corner point. ∎

Definition 3.3.3 *A* normalized rectilinear merge tree *is a rectilinear merge tree that has*

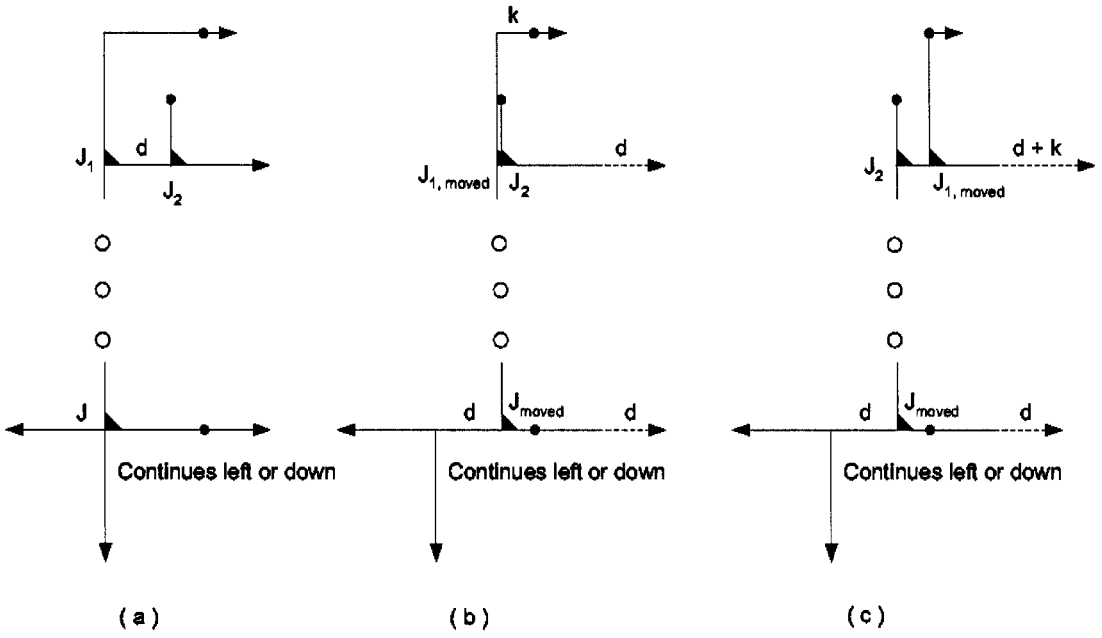


Figure 3.9: Shows a series of transformations until each junction has a continuous vertical line above with an endpoint of a client arrival. Part (a) shows the initial configuration. Part (b) shows the transformation for junction J. Part (c) shows the transformation for junction J_1. Moved junctions are indicated by *moved* in the figure.

undergone the transformations in Properties 3.3.3 and 3.3.4.

3.3.3 Possible Rectilinear Merge Trees for a Client Set

Now that we have established that rectilinear merge trees can be normalized to obey the properties described above, we can further show that the number of possible normalized rectilinear merge trees for a client set C is finitely bounded. By having a finite set of rectilinear merge trees, we can generate the set of trees and find the one that minimizes total server bandwidth. We first show that there are restrictions on to whom clients can merge. These restrictions allow us to build ordered standard merge trees to specify the merge schedule. By using the transformation properties of rectilinear merge trees, we limit the structure of the possible rectilinear merge trees that can be generated from an ordered

standard merge tree. In fact, the transformations in the properties guarantee that each ordered standard merge tree maps to the least-cost rectilinear merge tree, which can be used to find the total server bandwidth for the set of clients.

Definition 3.3.4 *The* target stream *for a client c in a normalized rectilinear merge tree is the first stream to which c merges. A target stream for c must be represented by a horizontal line, since c must receive media segments from the horizontal stream before merging to it.*

Since we know from Property 3.3.3 that every horizontal stream has a client arrival, we can name each horizontal stream by its right-most client arrival. In the rectilinear merge tree, the target stream for c can be found by the following:

- If c is on a vertical line, its target stream is the horizontal stream that meets at the first junction along the vertical line under c.

- If c is on a horizontal line H, and there are client arrivals to its right along H, its target stream is H named by the right-most client on the line.

- If c is on a horizontal line H, and there are no client arrivals to its right along H, then follow the path to the left and down toward the root stream. Let VS be the first vertical segment along this path. Then c's target stream is the horizontal stream meeting at the first junction along VS.

Definition 3.3.5 *The* target client *for a client c in a normalized rectilinear merge tree is the client to which c first merges. The target client for c must lie on the line representing c's target stream and lie to the right of the merge junction where c meets its target stream.*

Definition 3.3.6 *The* merge set *for a client c in a normalized rectilinear merge tree is the set of clients that eventually merge to stream c. If a client is positioned at the top-most point of a vertical line, then its merge set is empty. If a client is positioned along a horizontal stream named for a different client, then its merge set is empty. Therefore, only the right-most clients along horizontal streams have non-empty merge sets. To assemble the merge set for such a client c, find all clients along paths that eventually merge to stream c.*

Given a client c, are there restrictions as to which clients can be c's target client? If clients have a limited set of target clients, this restricts the number of possible merge schedules for the entire client set C.

Definition 3.3.7 *The* possible target client set *for a client c is the set of clients to which c can merge in any rectilinear merge tree.*

If we have a client c in the rectilinear grid, where can its target client be located in the grid? Client c could lie on its target stream, meaning its target client is located horizontally to the right of c in the grid. If client c does not lie on its target stream, its target stream must be below the position of c in the grid. Thus, c's target client could be located anywhere below c in the grid. Therefore, c's target client must be to its right or anywhere below in the rectilinear grid. Figure 3.10 shows the region outlining the possible target client set for the client shown in grey. The possible target client set for the client shown in grey is the set of clients under the dotted line.

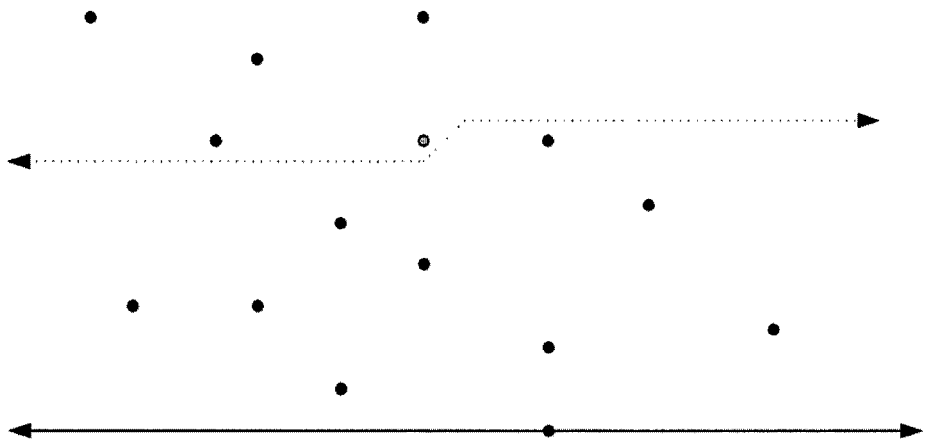


Figure 3.10: Shows a set of client arrivals on the rectilinear grid. The root stream is represented by the solid line. The possible target clients for the grey client are located under the dotted line.

The restriction of possible target client sets for each client allows us to define an ordering on a finite set of clients. We order clients by increasing distance from the root stream. For

clients lying on the same horizontal line, we order them by ascending arrival times. The first client arrival in the ordering is positioned at the right side of the lowest horizontal grid line that contains client arrivals. The last client will be leftmost client on the highest horizontal grid line that contains client arrivals.

Definition 3.3.8 *The ordered set* $C_O =< c_0, c_1, \ldots, c_{n-1} >$ *is a permutation of the original client set* C *where clients are ordered first by increasing vertical distance to the root stream in the rectilinear embedding and within the same vertical distance from the root from right to left.*

Property 3.3.6 *The ordered set* $C_O =< c_0, c_1, \ldots, c_{n-1} >$ *contains all clients in the original client set* C *such that for each client* c_i *in* C_O*, its possible merge target client set is* $\{c_0, c_1, \ldots, c_{i-1}\}$.

Proof: Consider client c_i in C_O. Due to the ordering of clients, c_i must come after all clients below it and to its right in the rectilinear grid. Therefore, the possible merge target client set for c_i is exactly the set of clients that appear before c_i in C_0. ∎

The clients in Figure 3.3 produce the ordered client set C_O: $< f, g, h, b, e, c, d, a >$. In this example, client $a = (15, 0)$ can merge to any other client in C_O. Client c cannot merge to client d or client a.

Now that we have a client ordering, we can create a client merge schedule, represented as an ordered standard merge tree, that completely specifies the target client for each client in the set. Recall that the target client for a client c is the client to which c first merges. Every client must have a target client, since each client eventually merges to the root stream. If a client did not have a target client, it would never listen to another stream in the system. We also know that for each client c_i in the ordered representation, its target client must have an index smaller than i. We use ordered standard merge trees to specify the client schedules.

Definition 3.3.9 *An* ordered standard merge tree *for a client set* C_O *of* n *clients is a rooted tree of* $n + 1$ *nodes such that the root stream is always the root node and each client* c_i *is represented by node* n_i*. Each node* n_i *must be a child of the root node or a child of a*

node n_j where $j < i$. Furthermore, for each non-leaf node n_i and the root node, its children are ordered from left to right by increasing indices.

An ordered standard merge tree completely specifies the entire merge schedule for each client. The succession of merges for a client c_i can be found by following the path from node n_i up to the root stream. A client c_i has a target client c_j if and only if n_i is a child of node n_j in the ordered standard merge tree. How many ordered standard merge trees (or merge schedules) are possible given an ordered client set C_O of size n?

Lemma 3.3.7 *Given an ordered client set C_O with $n > 0$ clients, there are $n!$ distinct ordered standard merge trees.*

Proof: This proceeds by induction on n, the number of clients in the ordered client set C_O.

Base Case: C_O has 1 client. Given that the root stream must be the root node, the only possibility is for c_0 to be the child of the root node. There is 1 possible tree for a single client and $1! = 1$.

Induction Hypothesis: Assume for a client set C_O of size n, there are $n!$ distinct ordered standard merge trees.

Inductive Step: Consider the client set $C'_O =< c_0, c_1, \ldots, c_n >$ which contains $n+1$ clients. Consider the set C_O that contains the first n clients of C'_O. From the induction hypothesis, C_O has $n!$ distinct ordered standard merge trees. Each tree for C_O has $n+1$ nodes, including the root node designating the root stream. Since c_n is the last client arrival in the ordering of C'_O, the node representing c_n in an ordered standard merge tree cannot have any children. Thus, c_n must be a leaf node. For each tree containing nodes for C_O, we can place c_n as the right-most child of any node c_0 through c_{n-1} or place c_n as the right-most child of the root node. This creates $n+1$ possible parents for c_n and each placement of c_n creates a distinct tree. There are $n!$ trees representing C_O and for each tree, we can create $n+1$ new trees. The total number of distinct ordered standard merge trees for C'_O is $n!(n+1) = (n+1)!$.

Thus, for each ordered client set C_O of size n, there are $n!$ distinct ordered standard merge trees. ∎

We have shown that there are $n!$ different ways to completely schedule how clients merge to one another in the system. How do we embed this schedule into a rectilinear merge tree? Might there be more than one way to embed the schedule in a rectilinear merge tree? Given the schedule in the ordered standard merge tree, is there a corresponding rectilinear merge tree with the lowest cost for that schedule? We first describe an algorithm to create a rectilinear merge tree from an ordered standard merge tree. Next, we show that the rectilinear merge tree created by the algorithm must have the minimum cost for any rectilinear merge tree that preserves the merge schedule dictated in the ordered standard merge tree. This guarantees that there is a finite number (at most $n!$) of possible rectilinear merge trees for a client set C of size n and one can search for the optimal rectilinear merge tree from this set.

Lemma 3.3.8 *Given an ordered standard merge tree T with n clients, there is at least one way to connect the clients in a rectilinear merge tree.*

Proof: Suppose we have an ordered standard merge tree T and there exists a pair of clients c_i and c_j where n_i is a child of node n_j in T such that c_i and c_j cannot be connected in the rectilinear merge tree. Since n_i is a child of n_j, c_i must be above or to the left of c_j in the rectilinear grid. Rectilinear merge trees can have lines going down and to the left. If c_i and c_j are co-linear, then we can connect them via a horizontal line. If c_i is above c_j and c_j is to the right of c_i, we can connect the clients at a junction that is below c_i and to the left of c_j. We may need to add a premature start to the right of c_j if the horizontal line is not long enough for the junction. If c_i is above c_j and c_j is to the left of c_i, then we can create a junction below c_i and to the right of c_j by extending a horizontal line to the right of c_j. In all cases, we can create a connection such that c_i merges to c_j in the rectilinear merge tree. ∎

Now that we have shown that we can form a rectilinear merge tree given an ordered standard merge tree, we describe an algorithm to transfer the merge schedule specified in the ordered standard merge tree to a rectilinear merge tree.

The Rectilinear Merge Tree (RMT) Creation Algorithm

Input: A set of n clients C with (t, f) pairs in an ordered standard merge tree T.

Output: A rectilinear merge tree RMT with the client merge schedule specified in T.
Procedure:

1. Place all clients in C in the rectilinear grid formed above the root stream. The root stream should start with time 0 and segment 0 and extend to the left with an endpoint under the left-most client positioned in the grid.

2. For $i = (n-1)$ to 0, do the following: Select node n_i from the tree T. Let p_i be the parent node of n_i in T. Let c_i and ct_i be the clients representing n_i and p_i, respectively. We place lines on the rectilinear grid according to the following cases. Vertical lines are always drawn, if instructed to do so, so that they do not overlap with existing vertical lines (creating multiple junctions at a single grid position). If instructed to draw a horizontal line, simply draw it on top of another horizontal line, if one exists.

   (a) If c_i lies within an existing horizontal line (dashed or solid) that extends to the right in RMT, then add nothing to RMT.

   (b) If c_i is to the left of ct_i at the same horizontal level, then draw a solid line connecting c_i and ct_i.

   (c) Else, the other case is where c_i is somewhere above ct_i in the rectilinear grid. Let d be the vertical distance between c_i and ct_i. Find the left-most point p_1 of the horizontal line extending to the left of c_i, if one exists. If no such line exists, then let p be at position c_i. Draw a vertical line down from p for d units. Let the endpoint of this vertical line be position p_2.

      i. If ct_i is to the left of p_2, draw a dashed horizontal line from ct_i to the right to p_2. Connect the vertical line and dashed line at p_2 with a junction marker. Add a dashed line of length d to the right of the junction marker to satisfy the elbow property.

      ii. If ct_i is at or to the right of p_2, draw a solid horizontal line from p_2 to the right to ct_i. Connect the vertical line and horizontal line at p_2 with a junction marker. If the solid horizontal line to the right of p_2 is shorter than d units

long, then add a dashed line so the total horizontal line to the right of p_2 has a total length of d.

The following example shows how the ordered client set C_O is created from a set of clients. Using C_O, we create a single ordered standard merge tree T and then show each step of the RMT Creation Algorithm.

Example 3.3.1 *There are five client arrivals:* $a = (7,5)$, $b = (7,6)$, $c = (8,3)$, $d = (10,8)$, *and* $e = (13,10)$. *Figure 3.11 shows how these clients are positioned in the grid. Therefore,* $C_O =< b, a, d, e, c >$. *We show the transformation from the ordered standard merge tree in Figure 3.12 to the rectilinear merge tree in Figure 3.13 using the RMT Creation Algorithm. The cost of the rectilinear merge tree is 34.*

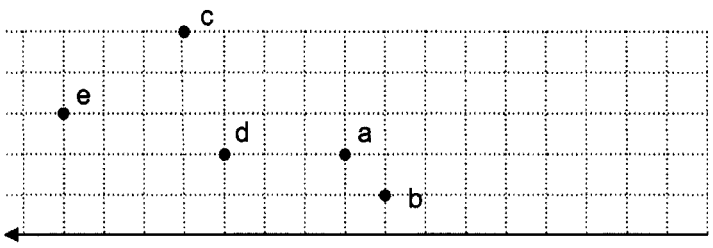


Figure 3.11: Shows the rectilinear grid embedding for clients $a = (7,5)$, $b = (7,6)$, $c = (8,3)$, $d = (10,8)$, and $e = (13,10)$. The order of the clients in C_O is $< b, a, d, e, c >$.

We create the rectilinear merge tree corresponding to the ordered standard merge tree shown in Figure 3.12. The first client that we process is c, since c is the last client in C_0. From the ordered standard merge tree, c merges to e, so we draw a vertical line below c and draw a premature start horizontal line through e that is just long enough for the junction under c. This is shown as the second figure in the first column in Figure 3.13. The next client we process is client e. From the ordered standard merge tree, e merges to a. Thus, we draw a vertical line below e and introduce a solid horizontal line between a and the junction under e. This is shown in the bottom figure of the first column in Figure 3.13. Next, we process client d. Since d already lies within a horizontal line that extends to the right, we

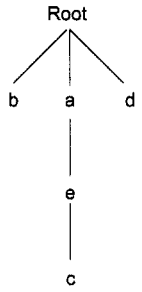


Figure 3.12: Shows one of the ordered standard merge trees for the clients $a = (7, 5)$, $b = (7, 6)$, $c = (8, 3)$, $d = (10, 8)$, and $e = (13, 10)$.

add nothing to RMT. Client a is processed next and will merge to the root stream. Thus, we follow the existing horizontal line to the left of a until it ends and draw a vertical line from there to the root stream. This is shown in the second figure of the second column in Figure 3.13. Last, we process client b, which merges to the root stream. We simply draw a vertical line from b to the root stream, which is shown in the final figure of Figure 3.13.

Lemma 3.3.9 *A RMT created by the RMT Creation Algorithm does not contain premature starts appended to horizontal streams that are longer than necessary to satisfy the elbow property.*

Proof: Assume the RMT created by the algorithm contains a premature start to a horizontal stream that is longer than necessary. The only case where we add dashed horizontal lines in the algorithm is in case C. If ct_i is to the left of p_2, we only add a premature start of length d to the right of p_2 which just satisfies the elbow property at the created junction. If ct_i is to the right of p_2, we only add a dashed line if ct_i is fewer than d units to the right of p_2. Again, the dashed line is added to create a horizontal stream that is d units long for the junction. None of the premature starts are longer than necessary in RMT. ∎

Lemma 3.3.10 *The RMT created by the RMT Creation Algorithm satisfies Properties 3.3.3 and 3.3.4 of normalized rectilinear merge trees.*

Proof: Assume we have RMT for client set C created through the process described in the RMT Creation Algorithm. First, we show that all horizontal lines in RMT have at least

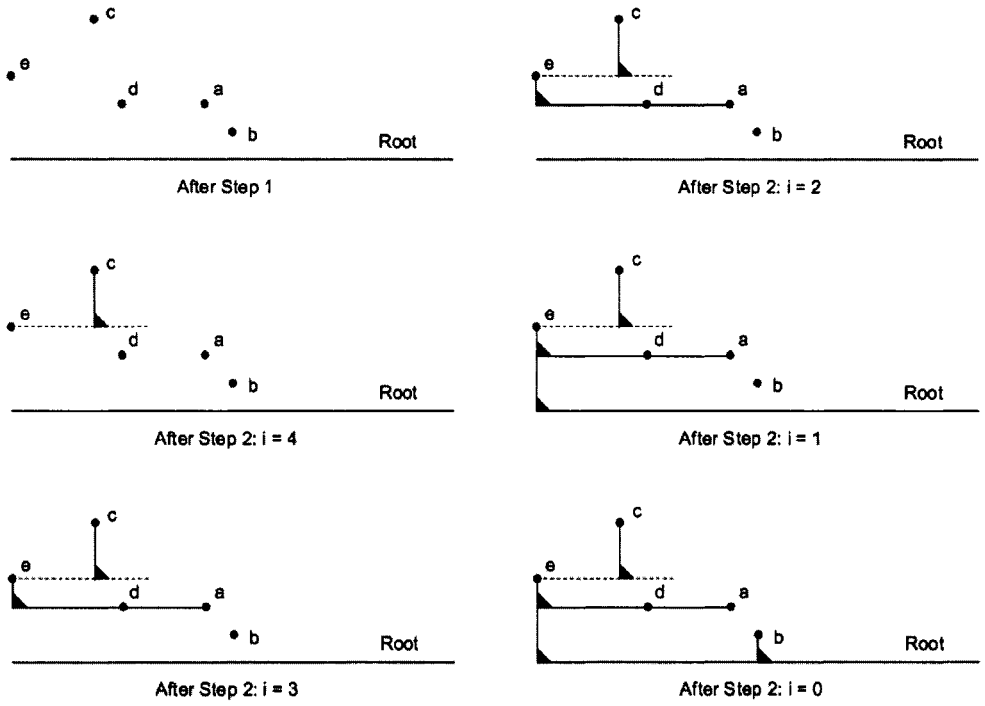


Figure 3.13: Shows the steps of the RMT Creation Algorithm from the ordered standard merge tree shown in Figure 3.12. The final rectilinear merge tree is shown in the bottom figure of the second column.

one client arrival. Horizontal lines are only drawn from clients embedded in the rectilinear grid. Therefore, each horizontal line contains at least one client.

Now we show the case that all junctions have at least one continuous vertical line whose top-most point is a client arrival. Assume to the contrary that there is a junction J in RMT without a continuous vertical line ending at a client arrival. Then each vertical line either ends with the start of a stream or a corner. We never start a vertical stream without a client arrival in the algorithm, so this case cannot happen. Therefore, each continuous vertical line must end with a corner where there is no client arrival at the corner. At some point in the algorithm, a horizontal line without a client arrival at its left endpoint must have been added to RMT. Let us see how horizontal lines are added in the algorithm. In the second case, where c_i and ct_i are at the same horizontal level, the algorithm only adds a horizontal

line between c_i and ct_i, so client c_i is definitely at the left endpoint of this horizontal line. The third case also produces a horizontal line. If ct_i is to the left of p_2, we draw a horizontal line from ct_i to the right to p_2. Again, there is definitely a client arrival, namely ct_i that is the left endpoint of this horizontal line. The other case, where ct_i is located at or to the right of p_2 adds a horizontal line from p_2 to the right. There may not be a client arrival at p_2, but a junction is created at p_2. Thus, this horizontal line has a left endpoint with a junction and not a corner. We never create a horizontal line that has a corner without a client arrival at its left endpoint. Therefore, the RMT created satisfies Property 3.3.4. ∎

At this point we know that the RMT created from an ordered standard merge tree satisfies the properties of normalized rectilinear merge trees. Additionally, in a RMT each client starts listening to its target client as soon as possible, since all corners contain client arrivals. Now we can prove the claim that the RMT created from T has the least server bandwidth cost of all possible RMT's generated from T.

Lemma 3.3.11 *Let RMT be the rectilinear merge tree created from T by the RMT Creation Algorithm. Then RMT has the least server bandwidth cost of all possible rectilinear merge trees with the merge schedule dictated in T.*

Proof: Let T be an ordered standard merge tree for a client set C and let RMT be the rectilinear merge tree created from T through the RMT Creation Algorithm. Assume there exists an optimal RMT' with the merge schedule dictated in T that has lower cost than RMT. Since RMT' has the merge schedule dictated by T, all clients in RMT' merge to other clients. This guarantees that there are no horizontal lines without client arrivals in RMT'. Because the costs differ, RMT and RMT' must differ structurally in some way. We create a new rectilinear tree RMT'' from RMT' by applying transformations and show that RMT and RMT'' must be the same tree. Since RMT' is a rectilinear merge tree, apply the transformations in Property 3.3.4 to RMT' to get RMT''. These transformations do not alter the client merge schedule as dictated in T. Vertical lines are moved to the right, but all junctions to horizontal streams remain intact. Therefore, each client keeps its target client after the transformations. The transformations are guaranteed to preserve or reduce the overall server bandwidth, so the cost of RMT'' is less than or equal to the cost

of RMT' which we assumed is less than the cost of RMT. In fact, since RMT' is assumed to be optimal, the cost of RMT' and RMT'' must be the same. Lemma 3.3.10 specifies that RMT satisfies the properties of the transformed tree RMT''. The transformations guarantee that each client starts listening to its target client as soon as possible, Since RMT'' is optimal for the client schedule in T, RMT'' cannot have horizontal premature starts that are longer than necessary and RMT'' cannot have premature vertical starts. The RMT Creation Algorithm does not create premature starts either. Therefore, both RMT and RMT'' must be the same rectilinear merge tree and both have the same cost. We have a contradiction, so RMT must have the lowest server bandwidth of any rectilinear merge tree with the client schedule specified in T. ∎

There are $n!$ distinct ordered standard merge trees for a client set of size n and for each ordered standard merge tree T, we can create a rectilinear merge tree with the same merge pattern as in T that has the least cost for this merge pattern. Once all rectilinear merge trees are produced, we can choose the one that has the least total edge length. This rectilinear merge tree is guaranteed to have the optimal cost of all rectilinear merge trees for the client set. The running time of the following algorithm is exponential in the number of clients n, since we need to search through all $n!$ possible rectilinear merge trees.

Optimal RMT Search Algorithm

Input: A client set C that contains n clients with (t, f) pairs.

Output: The optimal rectilinear merge tree RMT for C along with the server bandwidth cost of RMT.

Procedure:

1. Create the set C_O from the client set C.

2. Generate the $n!$ possible ordered standard merge trees from C_O.

3. For each ordered standard merge tree, apply the RMT Creation Algorithm to create the rectilinear merge tree. Count the total edge length of rectilinear merge tree.

4. Return the rectilinear merge tree produced in Step 3 that has the least total edge length.

3.3.4 Crossovers and Optimality

The optimal time-shifting rectilinear merge tree for a client set C could contain crossovers. Consider the set $C : a = (28, 0)$, $b = (49, 43)$ and $c = (51, 44)$. Since this client set has three clients, there are 3! possible ordered standard merge trees. $C_O =< b, c, a >$. Figure 3.14 shows the six standard ordered merge trees for the client set. Figure 3.15 shows the six rectilinear merge trees created using the RMT Creation Algorithm. Of these, the best way to connect the clients is shown in part (I), where there is a crossover below a and to the left of b. The server bandwidth cost (eliminating the root stream) for the three clients in the tree shown in part (I) is 24. The other five trees have costs exceeding 24. Thus, we have shown a case where the optimal rectilinear merge tree is not planar.

3.3.5 Eliminating Redundant Rectilinear Merge Trees

Two or more distinct ordered standard merge trees can produce the same rectilinear merge tree. We can eliminate ordered standard merge trees that produce redundant rectilinear merge trees in the RMT Search Algorithm. This will reduce the number of rectilinear merge trees that must be generated in order to find the optimal tree. The reason distinct ordered standard merge trees can produce the same rectilinear merge tree is due to the fact that the algorithm does not add anything to the RMT if a client already lies on a horizontal line. If a client already lies on a horizontal line, then that client could be placed in any legal position for the ordered standard merge tree. Recall that clients must have parents with smaller indices and the set of children for each node are ordered by increasing index from left to right.

Going back to Example 3.3.1 and the example ordered standard merge tree in Figure 3.12, we see that there are other ordered standard merge trees that produce the rectilinear merge tree in Figure 3.13. Figure 3.16 shows Trees 2 and 3 that produce the same rectilinear merge tree as Tree 1. The connection between client c and client e always ensures that client

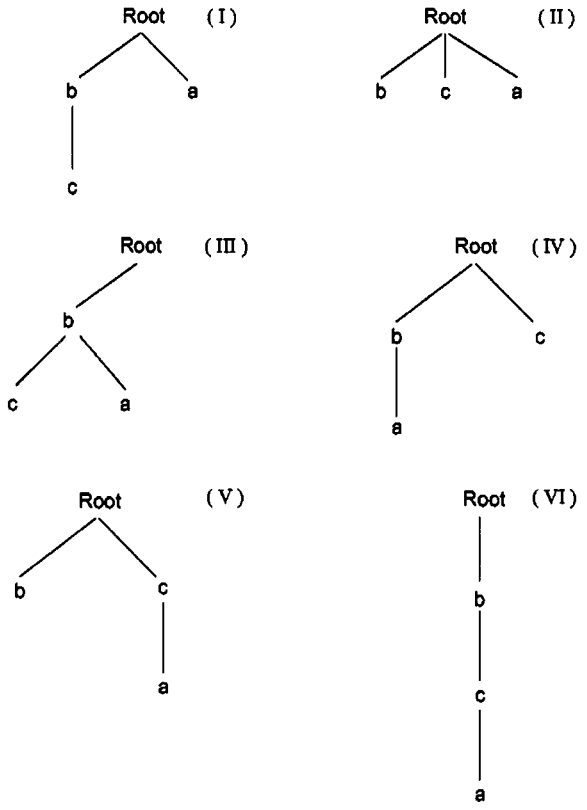


Figure 3.14: Shows the six possible standard merge trees for clients $a = (28, 0)$, $b = (49, 43)$, and $c = (51, 44)$. The client ordering is $\{b, c, a\}$.

d will lie on a horizontal line. Therefore, if an ordered standard merge tree has c as a child of e, we can ignore the placement of client d in the tree.

We can extend this example to the general case. Let C be the set of clients. Let c_i be a client that can merge to c_j. Connect these clients as described in the RMT Creation Algorithm. Find the set of clients that lie on the horizontal line created (other than c_i and c_j). This set could be empty. If this set is non-empty, then we can ignore these clients when creating the rectilinear merge tree from any standard ordered merge tree that contains n_j as a parent of n_i.

We formalize this elimination procedure with the following algorithm. Before applying the algorithm, we assume all clients lying on the root stream have been eliminated from

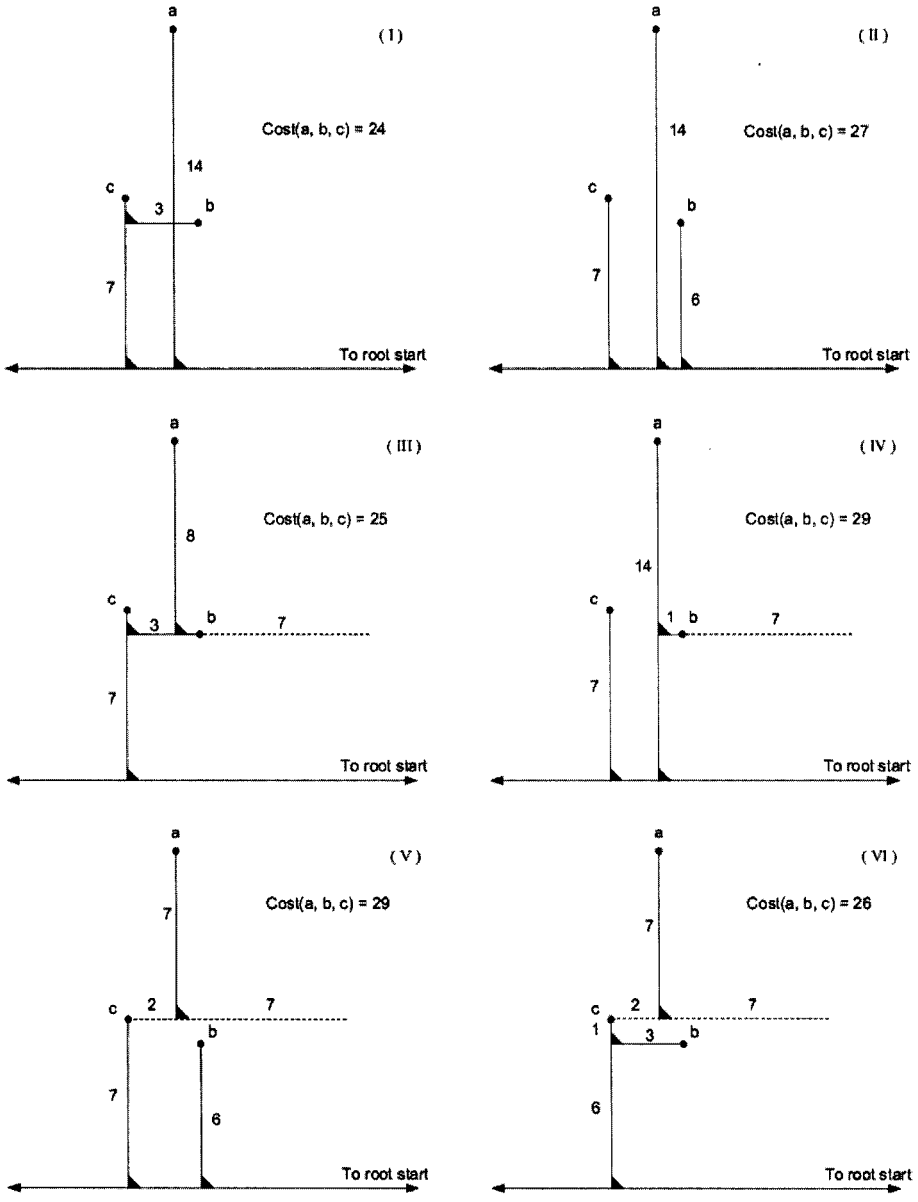


Figure 3.15: Shows the six possible rectilinear merge trees for clients $a = (28, 0)$, $b = (49, 43)$, and $c = (51, 44)$. The tree shown in part (I) is the optimal tree and contains a crossover, with total cost for the three clients (not including the root stream) of 24.

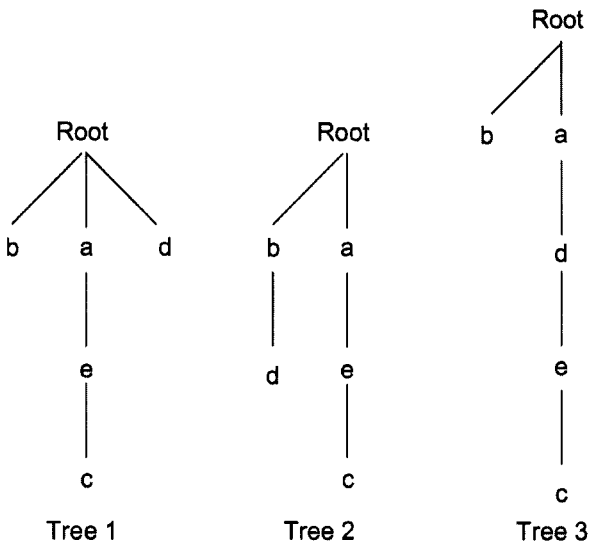


Figure 3.16: Shows three ordered standard merge trees that produce the same rectilinear merge tree for the five clients in Example 3.3.1.

C_O.

Ordered Standard Merge Tree (OSMT) Elimination Algorithm

Input: Set S of all ordered standard merge trees for a client set C_O. There are n clients in C_O and no client lies on the root stream.

Output: Subset S' of S of size k that contains ordered standard merge trees producing k distinct rectilinear merge trees, where $k \leq n$.

Procedure:

1. Here we let the root stream be a target client for any client in C_O. For each possible pair c_i and c_j where c_i can have as its target client c_j, connect these clients according to the RMT Creation Algorithm and find the set E_{ij} of clients that lie on the horizontal line (dashed or solid) created by the connection. [Note: Since the clients are ordered, there are $n + (n-1) + (n-2) + \ldots + 1 = n(n+1)/2$ possible pairings.] This creates a list L of tuples (c_i, c_j, E_{ij}). We can assume all clients with a target client as the root stream will have empty elimination sets, since there are no client arrivals lying on the root stream.

2. For each tree T in S, perform the following actions from leaves to children of the root node: For each parent-child node pair (c_i, c_j) in T, find the elimination set E_{ij} for (c_i, c_j) in L. Remove each client c_r in E_ij from T. If c_r is a leaf node, simply remove it and the edge to its parent node. If c_r is not a leaf, connect the parent of c_r to all c_r's children and remove c_r and its edges from T.

3. With the resulting trees T in S, put T in S' if S' does not already contain a tree equivalent to T.

4. Return S'. This set can be used to create the set of possible rectilinear merge trees.

Example 3.3.2 *Let the client arrivals be those as in Example 3.3.1. The pairing list L contains the following:* $(a, b, empty)$, $(d, b, empty)$, $(d, a, empty)$, $(e, b, empty)$, $(e, a, \{d\})$, $(e, d, empty)$, $(c, b, empty)$, $(c, a, \{d\})$, $(c, d, empty)$, $(c, e, empty)$. *Consider applying Step 2 of the OSMT Elimination Algorithm to Tree 1 of Figure 3.16. The pair* (c, e) *does not remove anything from the tree. The pair* (e, a) *removes d from the tree. The pairs with the root as the target client never remove a client from the tree. All three trees in Figure 3.16 have the same structure after applying Step 2 to each tree. Just one of these trees will be inserted into* S' *in Step 3.*

3.3.6 Direct Calculation of Stream Lengths

In [38] Goshi presents the time-shifting stream merging model and gives the following formula for calculating stream lengths. In Equation 3.1 c is a client arrival with time $t(c)$ and requested first segment $f(c)$. Clients are nodes in a merge tree (not a rectilinear merge tree), such that the parent of a client c has notation $P(c)$ and $Z(c)$ is the last client to eventually merge to the stream started for c. In the merge tree representation, $Z(c)$ is the right-most leaf node of the subtree rooted at c.

$$\ell(c) = [2t(Z(c)) - t(c) - t(P(c))] + [f(P(c)) - f(Z(c))] \tag{3.1}$$

An implicit assumption in the proof that the above equation calculates stream lengths is that $P(c)$ must start at or before c for all clients c in the system. As we have shown

earlier, clients can indeed merge to clients not yet in the system by creating premature starts. Equation 3.1 does not work in the general case, where clients can merge to any other client that is below and/or to its right in the rectilinear grid embedding. Here we show how to calculate stream lengths in the general setting using ordered standard merge trees. The direct calculation alleviates the need to use the RMT creation algorithm in order to calculate the total server bandwidth.

Let T be an ordered standard merge tree in the set S' after running the ordered standard merge tree elimination algorithm. Therefore, all elimination sets for ordered client pairs are empty for the clients remaining in T. We use the structure of T to determine stream lengths. Instead of just finding the last client $Z(c)$ to merge to c, we also need to account for the stream extension for premature starts, and a premature start could result from any child of c in T. Thus, we need to keep track of the entire tree when calculating stream lengths.

Let N be the collection of nodes in T. The root node in T is always the root stream. Each node n in the set N will contain information about the client arrival and the leftmost junction location for the stream initiated for the client arrival. Additionally, each node will contain the location of the junction to its parent in the tree T. We also keep track of the length of each stream. Each node has the following properties:

- $\ell(n)$ – the length of the stream initiated for client arrival c that corresponds to the node n in T.
- $c(n)$ – the client arrival pair with values $t(c)$ and $f(c)$ for this node.
- $LJ(n)$ – the leftmost junction on the stream initiated for c with values $t(LJ(n))$ and $f(LJ(n))$.
- $PJ(n)$ – the junction location formed between the client and its parent node with values $t(PJ(n))$ and $f(LJ(n))$.

Given T, we perform the calculations from leaf nodes to the root of the tree. The length of a stream for a client c cannot be calculated until the node values (listed above) have been

calculated for all of c's children. Therefore, finding the lengths of all the streams is linear time with respect to the number of nodes in the tree, but finding the length of a single stream is linear time with respect to the number of nodes in the subtree rooted at the client that started the stream. In the algorithm below, the set of children for a leaf node is empty. We denote the set of children of node n as $C(n)$. Since T is a tree, each node has a single parent. The parent of node n is $P(n)$.

Stream Calculation Algorithm

Input: An ordered standard merge tree T whose client pairs have empty elimination sets.

Output: Stream lengths for all nodes in T.

Procedure:

1. Initialize $\ell(n)$ values for all nodes in T to be 0. We assume $c(n)$ client pairs are part of the input tree T. Initialize $LJ(n)$ and $PJ(n)$ to null for all nodes n in T. Set the $LJ(n)$ pairs to equal $c(n)$ for all leaf nodes n in T.

2. We assume all the children of the node n for which we are setting values already have all node values set. If not, perform the following operations on the children of n recursively.

3. Find the length of a premature start, if there is one and add it to $\ell(n)$.

$$\ell(n) += \max\{0, t(c) - \min_{n' \in C(n)} \{t(LJ(n'))\}\} \tag{3.2}$$

4. Calculate the value $LJ(n)$ with the following procedure. If n is a leaf node, then $LJ(n)$ is equal to $c(n)$. Otherwise, find the largest time t of all $PJ(n')$ pairs where n' is a child of n. If t is larger than $t(c)$, then $LJ(n)$ is set to the pair with this maximal t value. If not, then $LJ(n)$ is set to $c(n)$.

5. Find the length of the stream from the client start time $t(c)$ to its left-most junction by the following:

$$\ell(n) += \max\{0, t(LJ(n)) - t(c)\} \tag{3.3}$$

6. For non-root nodes calculate the parent junction $PJ(n)$ given $LJ(n)$ and $c(P(n))$.

$$t(PJ(n)) = [t(LJ(n)) - f(LJ(n))] - [t(c(P(n))) - f(c(P(n)))] + t(LJ(n)) \quad (3.4)$$

$$f(PJ(n)) = 2([t(LJ(n)) - f(LJ(n))] - [t(c(P(n))) - f(c(P(n)))]) + f(LJ(n)) \quad (3.5)$$

7. For non-root nodes calculate the length of the stream from the left-most junction to the merger to its parent:

$$\ell(n) += t(PJ(n)) - t(LJ(n)) \quad (3.6)$$

After applying the above series of steps to every node in T, starting with the leaf nodes, we have the value $\ell(n)$ for every node, which is the length of the stream for the corresponding client.

Figure 3.17 shows the calculation of stream lengths for a standard ordered merge tree. On the right is the standard ordered merge tree along with the node values and on the left is the rectilinear merge tree produced from the standard ordered merge tree. The length of stream a is 3, which is the length of the vertical line below a in the rectilinear merge tree. The length of stream b is 8, which includes the total length of the horizontal line going through client b plus the vertical line of length 2 below client c in the rectilinear merge tree. The length of the stream for client c is 0, since no new stream is initiated. The length of the stream for client d is 3, which is the horizontal line of length 2 plus the vertical line of length 1 to the root stream. The algorithm does, indeed, calculate the correct stream lengths.

If instead we use the calculation for stream lengths proposed in [38] using Equation 3.1, we get the following values for stream lengths for the arrivals depicted in Figure 3.17. The lengths of clients a, c, and d are calculated correctly since those streams do not have premature starts. The length of stream b, however, is calculated to be 6 units long according to Equation 3.1 when it should really be 8 units long to account for the premature start. As a second example. consider the rectilinear merge tree shown in the bottom right of Figure 3.13. The stream calculation algorithm above calculates the stream lengths for clients a, b, c, and e correctly. We ignore client d since it lies on the horizontal stream for client a.

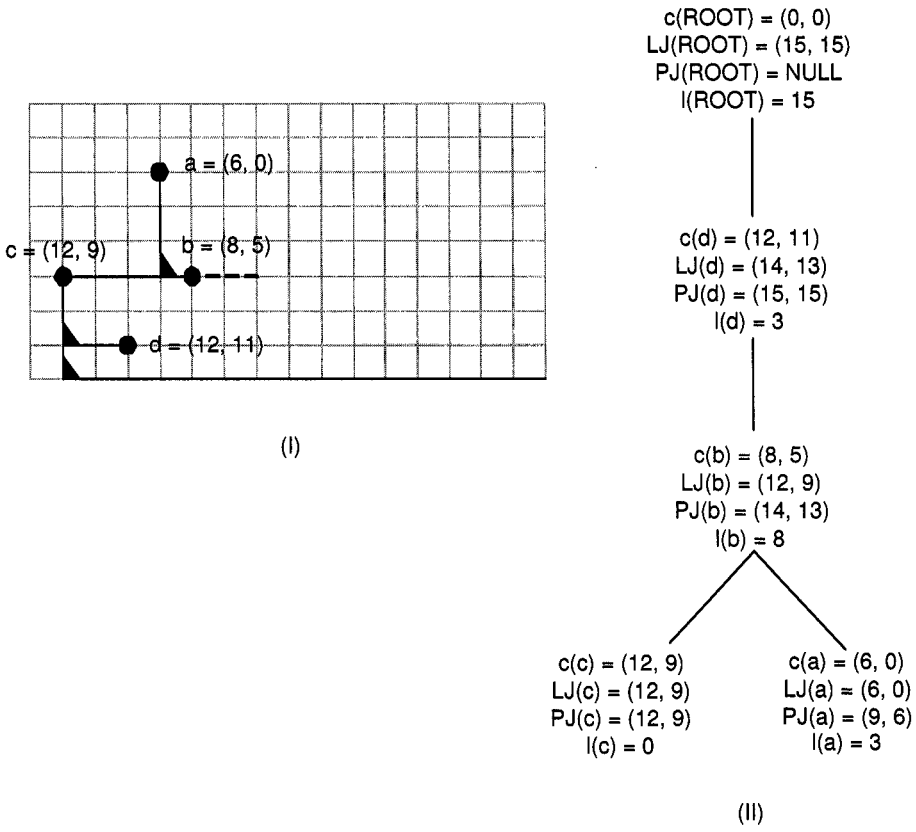


Figure 3.17: On the left (I), a rectilinear merge tree for four client arrivals as indicated. On the right is the standard ordered merge tree representing the merge schedule in (I) along with the node values for the stream calculation algorithm. The algorithm performs the node calculations from leaf nodes up to the root of the tree.

Equation 3.1 calculates stream lengths correctly for all clients except for e, which consists completely of a premature start. Equation 3.1 calculates the length of stream e to be -2 units long. We have shown how to calculate stream lengths correctly for general merge patterns that could require introducing premature starts.

3.3.7 Local Optimizations

Are there subsets of clients that are always connected optimally with a single connection structure regardless of other clients in the system? If so, then we can reduce the number of

clients necessary for the ordered standard merge trees. As will be shown in Chapter 4, we can always connect a horizontal string of unit spaced clients with a single horizontal line in an optimal rectilinear merge tree. Let $C' = \{c_0, c_1, \ldots, c_{k-1}\}$ be K unit spaced clients along a horizontal line, we can eliminate the $K-2$ clients between the endpoints c_0 and c_{k-1} from the client set used to generate the ordered standard merge trees. Also all the ordered standard merge trees must have c_{k-1} as the parent of c_0. This local optimization reduces the total number of ordered standard merge trees that must be created to find the optimal rectilinear merge tree.

The second case, as will be shown in the Chapter 4, is side-by-side columns of unit spaced client arrivals. We can always connect these clients optimally via the zipper connection. We cannot eliminate the clients from the system, but we can ensure that all ordered standard merge trees have the clients in the double columns in a fixed arrangement as a chain of nodes in the tree. By restricting the tree structure we reduce the number of ordered standard merge trees that need to be created for the entire client set.

3.4 Conclusions

This chapter provided a model for the time-shifting problem which we use in subsequent chapters. We have introduced a new model for media-on-demand systems where clients access any part of the previously or currently broadcasted stream. We showed how this time-shifting model can be captured by a streams on a rectilinear grid. Merge schedules for stream merging events are made transparent in the rectilinear merge tree. We also defined properties of rectilinear merge trees and defined transformations such that all optimal rectilinear merge trees obey the structure of normalized rectilinear merge trees. A client ordering can then be used to generate merge schedules and corresponding rectilinear merge trees in order to find the optimal tree given a set of client arrivals.

The number of possible rectilinear merge trees for n clients is $n!$, so searching through the space of possible trees is prohibitive for large values of n. In the next chapter, we show that finding the optimal rectilinear merge tree for the time-shifting problem is NP-hard. Chapter 5 presents online algorithms for the time-shifting problem, making use of the

rectilinear grid introduced in this chapter.

Chapter 4

OPTIMAL SCHEDULE FOR MEDIA-ON-DEMAND WITH TIME-SHIFTING

In Chapter 3 we introduced the time-shifting model and captured streams and merge events in the form of rectilinear merge trees. Our goal is to minimize the total server bandwidth necessary to distribute the media broadcast to a set of clients. In this chapter we focus on the offline problem of determining the optimal rectilinear merge tree for stream merging in the time-shifting case given a set of client arrivals C. Is it possible to extend the optimal offline algorithm for the original media-on-demand case to the time-shifting case? The offline algorithm for the original media-on-demand case uses dynamic programming to find the optimal solution [8]. The clients are ordered by increasing arrival time and partitioned into two independent sets such that the first client of the second set is the last to merge directly to the root stream in an optimal solution. Recursively, this client becomes the root stream for a smaller subproblem.

We showed in Chapter 3 that the time-shifting client set C produces a client set C_O such that the possible merge targets for a client c are listed before c in C_O. The ordering in C_O does not guarantee a partitioning into two independent sets where the first arrival in the second set is the last to merge to the root stream in an optimal solution. Let C include $a = (5, 0)$, $b = (5, 2)$, and $c = (13, 9)$. The client ordering C_0 is $\{b, c, a\}$. The optimal rectilinear merge tree is shown in Figure 4.1. Intuitively, one can see that having all clients merge directly to the root costs 1 segment more since b is 3 units above the root stream. The cost for a to merge to c is more than for a to merge to the root stream. Similarly, the cost for c to merge to b is more than for c to merge to the root stream. We cannot partition the set, since c is the last to merge directly to the root stream and a does not merge to c in the optimal solution.

In this chapter we show that finding the optimal offline merge schedule for time-shifting

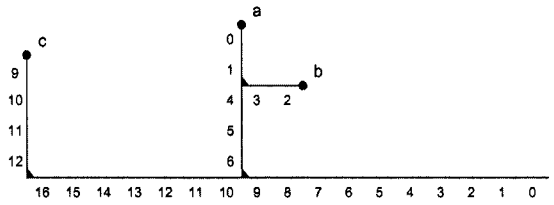


Figure 4.1: Shows the optimal rectilinear merge tree for clients $a = (5, 0)$, $b = (5, 2)$, and $c = (13, 9)$.

is NP-hard. This might be surprising since the optimal offline stream merging solution for the original media-on-demand case has a polynomial-time solution [8]. The chapter is organized as follows: we first describe the Rectilinear Steiner Arborescence Problem and explain its similarities and differences to the time-shifting problem and then show the stages of the reduction from planar 3SAT to a time-shifting instance.

4.1 Proof Outline

Our goal is to find the rectilinear merge tree that has the least total server bandwidth cost. This problem is closely related to the Rectilinear Steiner Arborescence Problem [61, 10, 24, 50, 65]. The Rectilinear Steiner Arborescence (RSA) Problem operates on a rectilinear grid with points in the first quadrant, or points such that the x and y components are both greater than or equal to 0. The problem is to find the tree rooted at the origin that has the least total edge length cost, connects to all points on the grid, and uses lines such that all paths leading from the origin to the points have line segments that point up and/or to the right.

The optimal time-shifting media-on-demand problem corresponds to the Rectilinear Steiner Arborescence Problem in that all lines must go to the right and/or up from the root node (the origin) to client arrivals in the rectilinear merge tree. In the time-shifting problem, we showed that the path must include lines going down and to the left from each client to the root stream. We can define the "origin" of the rectilinear merge tree as the end of the root stream, where the left-most client connects to the root stream. The time-shifting

problem differs from the RSA problem in that rectilinear merge trees must satisfy the elbow property, and they could have multiple junctions at a single location, crossovers, and premature starts. Therefore, the total server bandwidth for an optimal rectilinear merge tree could exceed the total length of the Rectilinear Steiner Arborescence connecting client arrivals. The last difference between a RSA and a rectilinear merge tree is that the rectilinear merge tree could have crossovers, which does not correspond to a planar RSA.

Shi and Su proved that the decision version of the RSA Problem is NP-complete by reducing planar 3SAT to the RSA Problem [65]. First, they transform the planar 3SAT instance to an instance I with additional vertices that can be embedded into a rectilinear grid such that vertices are adjacent if and only if they are located 1 unit apart. They develop a collection of tiles (configurations of points) corresponding to variables, or operations, negation operations, and clauses from the 3SAT instance. Each tile is constructed such that the set of points can be minimally connected in the RSA in at most two ways. These connections represent truth values of variables in the original planar 3SAT instance and interact to mimic or and negation operations. When the tiles are placed on the plane according to I's configuration, the original instance is satisfiable if and only if there is a Rectilinear Steiner Arborescence of length at most K where K is equal to the length of the minimum possible connection to points in all tiles.

The decision version of the optimal time-shifting with stream merging problem is: Given a set of client arrivals C and a positive integer K, is there a feasible stream merging schedule such that the total server bandwidth is at most K? We have shown above how to map a stream merging schedule to a rectilinear merge tree, so we reduce planar 3SAT to finding the optimal rectilinear merge tree for the set of client arrivals. The planar 3SAT problem is: Given a set of variables $V = \{v_1, v_2, \ldots, v_n\}$, a set of clauses $C = \{c_1, c_2, \ldots, c_m\}$, a set of edges $E = \{(v_i, c_j) : v_i \in c_j \text{ or } \neg v_i \in c_j\}$ and the bipartite graph $G = (V \cup C, E)$ is planar, is there an assignment to the variables such that all clauses are satisfied?

We use the same strategy as Shi and Su in [65] to prove that the decision version of the optimal stream merging with time-shifting problem is NP-complete. However, each of their tiles must be modified for the rectilinear merge tree problem. We build tiles to represent variables, the or operator, the negation operator, and the clauses in the original planar

3SAT instance. Each tile is built to ensure the following *reduction properties* of the optimal rectilinear tree: (i) each junction satisfies the elbow property, (ii) there are no premature starts, (iii) each grid position has at most one junction, and (iv) there are no crossovers. Finding the optimal rectilinear tree that satisfies the reduction properties is exactly like finding the minimum Steiner Arborescence for the set of client arrivals. Throughout the reduction, we will use this example planar 3SAT instance: $c_1 = (v_1 \vee v_2 \vee \overline{v_3})$ and $c_2 = (\overline{v_1} \vee \overline{v_4})$.

4.2 Degree-3 Planar Graph Conversion

First, we convert the planar 3SAT instance into a graph where each vertex has at most three incident edges. Let the original planar 3SAT instance be $G = (V, E)$ where V consists of two types of vertices – those representing variables in the instance (v_i) and those representing clauses in the instance (c_j). Let $G' = (V', E')$ be the transformed instance where each vertex in V' has three or fewer incident edges. The set V' will have three types of vertices - those representing variables, those representing negation, and those representing clauses. In the original graph G, a variable vertex v_i could have degree up to m, where m is the number of clauses, if the variable is present in each clause. To reduce the degree of variable vertices, we create a chain of $d(v_i)$ (the degree of v_i) vertices in G' that represents the single vertex v_i in G. We name this chain $u_i^1, u_i^2, \ldots, u_i^{d(v_i)}$. We add edges connecting successive vertices in the chain, so E' contains $(u_i^1, u_i^2), (u_i^2, u_i^3), \ldots, (u_i^{d(v_i)-1}, u_i^{d(v_i)})$.

Now we create the connections between variable vertices, negation vertices, and clause vertices in V'. Consider clause c_j in the original planar 3SAT instance. Clause c_j will fall into one of the four cases:

1. Assume c_j has just two variables and appears as $c_j = (v_i \vee \overline{v_k})$. Find all clauses that contain v_i or $\overline{v_i}$ and let clause c_j be the rth clause (in increasing subscript order) that contains v_i or $\overline{v_i}$. Similarly, find the sth clause that contains v_k or $\overline{v_i}$. Then, add clause vertex c_j to V' and add edges (u_i^r, c_j) and (u_k^s, c_j) to the edge set E'.

2. Assume c_j has three variables and appears as $c_j = (v_i \vee \overline{v_k} \vee \overline{v_\ell})$. Find the clauses that

contain v_i or $\overline{v_i}$ and let c_j be the rth such clause. Find the clauses that contain v_k or $\overline{v_i}$ and let c_j be the sth such clause. Find the clauses that contain v_ℓ or $\overline{v_\ell}$ and let c_j be the tth such clause. Add to V' an OR vertex o_j and clause vertex c_j. Add edges (u_i^r, o_j), (u_k^s, o_j), (u_ℓ^t, c_j), and (o_j, c_j) to E'.

3. Assume c_j has just two variables, but is not in the form of (1) above. If the first variable v_i is negated, add a NOT vertex between the variable vertex and insert edges from the vertex to the NOT vertex and between the NOT vertex at the clause. For example, if c_j is the rth such clause that contains v_i or $\overline{v_i}$, then we add the vertex n_i^r to V' and add edges (v_i^r, n_i^r) and (n_i^r, c_j). Similarly, if v_k is not negated, we perform a similar insertion of a NOT vertex.

4. Assume c_j has three variables, but is not in the form of (2) above. Then we insert NOT vertices wherever the variables do not match the form as in (3) above.

Now G' is still planar, but each vertex has at most degree 3. Each variable vertex u_i^j in V' has at most three edges. It might be connected to a chain of variable vertices representing v_i (at most two edges) and it is connected to either a clause vertex, a NOT vertex, or an OR vertex. Each clause vertex is connected to exactly two other vertices. It is connected to one of the following sets: (a) two NOT vertices, (b) one NOT vertex and one variable vertex, (c) two variable vertices, (d) one OR vertex and one variable vertex, or (e) one OR vertex and one NOT vertex. Each NOT vertex is connected to exactly two vertices: a clause vertex and a variable vertex. Each OR vertex is connected to exactly three vertices, one of which is always a clause vertex. The other two can be combinations of variable vertices and NOT vertices. In total, we have four types of vertices: variable vertices, NOT vertices, OR vertices, and clause vertices.

Figure 4.2 shows the transformation from a planar 3SAT graph G to a planar graph G' where each vertex has degree at most three. Since v_1 is used in two clauses, we represent v_1 as u_1^1 and u_1^2 in part (b). An OR vertex is used for clause 1, since clause 1 has 3 variables. Also, NOT vertices are introduced to create clauses in the right form.

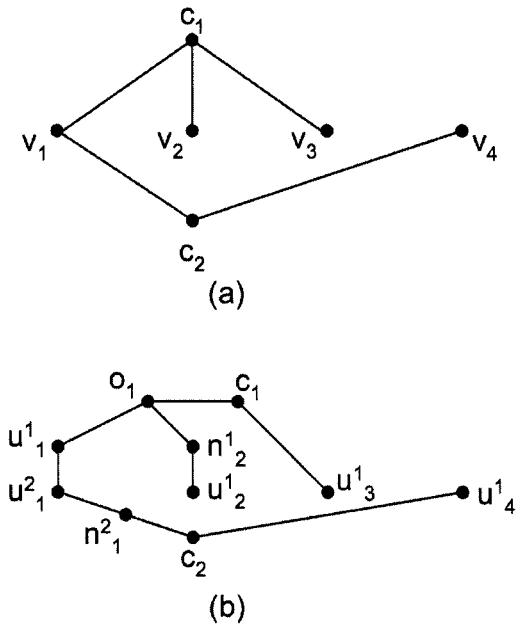


Figure 4.2: Part (a) shows the original planar 3SAT instance G where $c_1 = (v_1 \vee v_2 \vee \overline{v_3})$ and $c_2 = (\overline{v_1} \vee \overline{v_4})$. Part (b) shows the transformed graph G' where each vertex has degree at most 3. Clause vertices are represented by c_j, variable vertices are represented by u_i^r, OR vertices are represented by o_j, and NOT vertices are represented by n_i^r.

4.3 Rectilinear Grid Embedding

Graph G' is a degree-3 planar graph. Since each vertex has degree at most three, we can embed this graph into a rectilinear grid. We call the rectilinear graph R. G' has at most $3m$ variable vertices (the total degree of all the variables in the original graph G is $3m$ if each clause has 3 variables), m clause vertices, m OR vertices, and $3m$ NOT vertices (if each variable must be negated to appear in the right form). Therefore, the size of V' is at most $8m$. We can embed these vertices in a rectilinear grid of size $O(m^2)$.

In the embedding, we require that two vertices are incident in R if and only if they are one unit apart in the rectilinear embedding of R. We add auxiliary vertices, denoted as a, to the graph to satisfy this property. Also, each OR vertex will occupy two horizontally

adjacent vertices with an edge to a variable, auxiliary, or NOT vertex from the left, an edge to a variable, auxiliary, or NOT vertex from below, and an edge to a clause vertex to the right. Each clause vertex will be positioned such that the edge from a NOT vertex, a variable vertex, an auxiliary vertex, or an OR vertex will connect from the left and the edge from a NOT vertex, an auxiliary vertex, or a variable vertex will connect below. Each auxiliary vertex will be of degree 2, connecting any kind of vertex with any other kind of vertex. The original vertices of G' will be connected via the auxiliary vertices in R. Figure 4.3 shows the rectilinear embedding for the instance shown in Figure 4.2.

Once we have embedded R into the grid, we need to represent the following vertices in the rectilinear merge tree:

- variables
- clauses
- OR operations
- NOT operations
- auxiliary vertices for the embedding

4.4 Quadruped Components

Shi and Su use quadrupeds as basic building blocks for tiles [65]. We adapt their quadruped to satisfy the reduction properties. The quadruped represents parity information from the planar 3SAT instance. Figure 4.4 shows this basic structure, which consists of two types of points. The q points (unfilled points) represent client arrivals in the rectilinear grid. The b points (filled diamonds) may be client arrivals, may be junctions without a client arrival, or may not exist at all. Either b_1 and b_3 must exist or b_2 and b_4 must exist. There are no other client arrivals other than those indicated by circles in Figure 4.4. The distances are indicated by α, β, δ_1, and δ_2, which are greater than 0. In addition, $\beta + \delta_1 > \alpha$, so that q_3 is below q_4. This also ensures that the horizontal string of points to the right of q_4 is to the

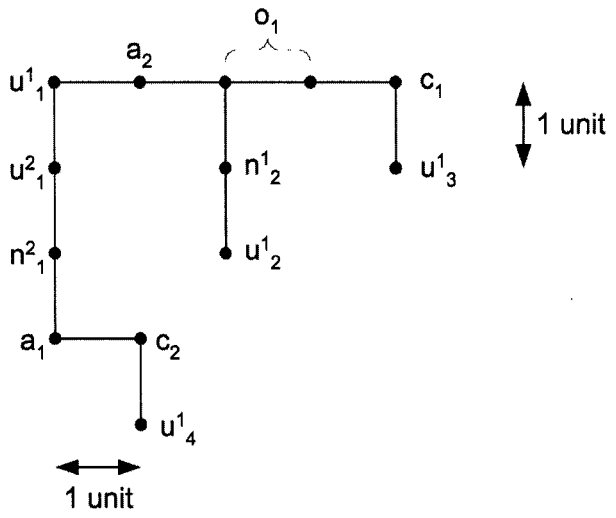


Figure 4.3: Shows the rectilinear embedding R of G' shown in Part (b) of Figure 4.2. Each vertex lies on an intersection point of the grid. Vertices are incident if and only if they are 1 unit apart. The vertices a_1 and a_2 are auxiliary vertices, added to create the rectilinear embedding.

left of the vertical line below q_2. Also, $\delta_1 + \delta_2 < \beta$ and $\delta_1 + \delta_2 < \alpha$. There are $\alpha - \delta_1 - \delta_2$ unit spaced client arrivals to the right of q_4. Similarly, there are $\beta - \delta_2$ unit spaced clients to the right of q_3. Here, both solid and dashed lines indicate streams, so dashed lines no longer represent premature starts. Notice that either the solid set of lines or the dashed set of lines can be used to connect to all four q points.

Definition 4.4.1 *A* minimum connection *of a quadruped is a set of rectilinear merge trees with roots at filled diamonds that connect to all client arrivals such that the total cost of the set of trees is minimal.*

We will now show that the quadruped obeys the necessary properties to ensure that the total length of the minimum connections is exactly equal to the server bandwidth. First, we show that just two minimum connections are possible (shown by solid and dashed lines.) Second, we show that the quadruped obeys the elbow property (without premature starts) at all junctions in a minimal connection. Next, we show that each grid location has at most

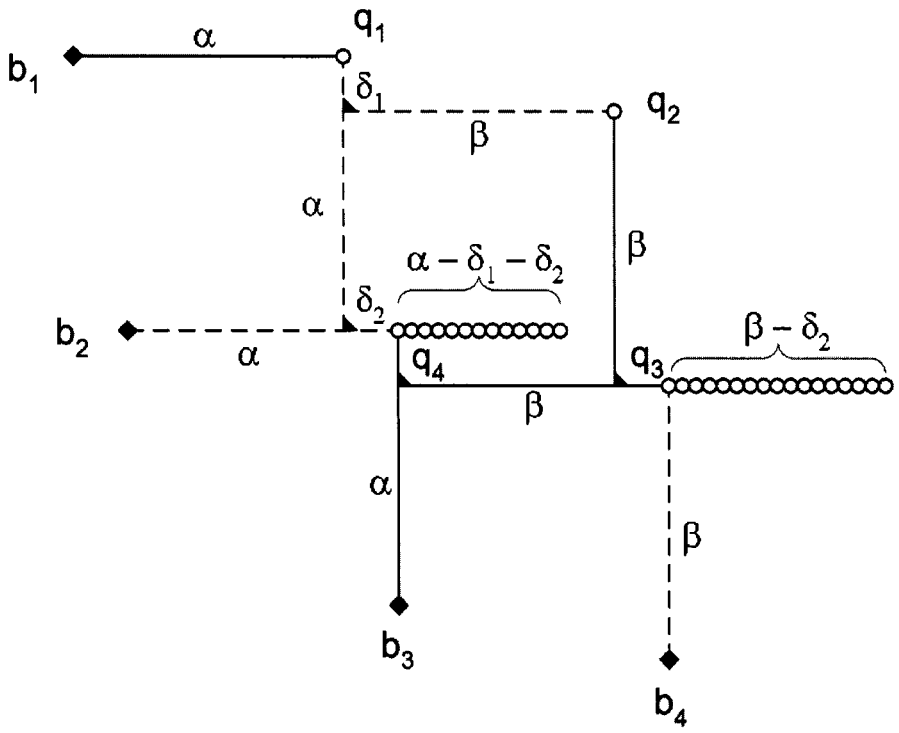


Figure 4.4: A quadruped: The basic structure for all the tiles. A minimum connection to the q points (unfilled circles) from a subset of the b points (filled diamonds) uses either the solid lines or the dashed lines and connects the horizontal series of client arrivals to the right of q_4 and q_3 with horizontal lines.

one junction. Finally, we will show that there are no crossovers in the minimum connection.

Lemma 4.4.1 *Let there be a horizontal series of K unit spaced clients. Then an optimal rectilinear merge tree containing these clients* either *connects the K clients with a horizontal line* or *the tree can be transformed into a tree with the same cost using a horizontal line to the connect the K clients.*

Proof: Assume we have $K = \{c_0, c_1, \ldots, c_{k-1}\}$ unit spaced clients arrivals on a single horizontal line and the minimum connection does not connect these via a single horizontal stream. Then for some client c_i where $i > 0$, the client has a vertical line leading down. Since this vertical line must be at least one unit long, we could instead connect c_i to c_{i-1} via a horizontal edge. This transformation is guaranteed to preserve or reduce the total server bandwidth cost of the minimal connection. Thus, we can assume the two series of horizontal client arrivals to the right of q_3 and q_4 in Figure 4.4 will always be connected via single horizontal streams. ∎

Lemma 4.4.2 *For the quadruped, there are only two minimum connections, one shown as dashed lines and one shown as solid lines in Figure 4.4. The total edge length of the minimum connection is* $3(\alpha + \beta) - 2\delta_2 - \delta_1$.

Proof: We show that a minimum connection must have total server bandwidth that is at least $3(\alpha + \beta) - 2\delta_2 - \delta_1$. From above, we can assume the two series of horizontal client arrivals are connected via horizontal streams of length $\alpha - \delta_1 - \delta_2$ and $\beta - \delta_2$. Now we need to connect client arrivals q_1, q_2, q_3 and q_4. To connect q_4, we need a path of at least α in length. To connect q_1, we also need a path of at least α even if we use the path to q_4. We need a path of at least β to connect q_3, even if we use the path to q_4. A path between q_2 and q_4 has length $\alpha - \delta_1 + \beta - \delta_2$ which is greater than β since $\alpha > \delta_1 + \delta_2$. Thus, the path to q_2 must be of length at least β. The lower bound for the minimum connection is $\alpha - \delta_1 - \delta_2 + \beta - \delta_2 + 2\alpha + 2\beta$ which equals $3(\alpha + \beta) - 2\delta_2 - \delta_1$. Every connection must have total length at least as much as the lower bound.

We show we can achieve the lower bound by using the solid edges or the dashed edges in Figure 4.4. It is clear from the figure that the solid edges or the dashed edges total $2\alpha + 2\beta$.

We can connect the horizontal series of unit spaced client arrivals with streams of length $\alpha - \delta_1 - \delta_2$ and $\beta - \delta_2$. Adding this to $2\alpha + 2\beta$, we have $3(\alpha + \beta) - 2\delta_2 - \delta_1$. Therefore, the solid and dashed lines form two separate minimum connections. ∎

Lemma 4.4.3 *Every junction in minimum connections of the quadruped in Figure 4.4 (consisting of dashed lines only or solid lines only) satisfies the elbow property for rectilinear merge trees. Furthermore, no premature starts are necessary to satisfy the elbow property.*

Proof: There are four such junctions. The junction formed by the dashed lines connecting q_1 and q_2 satisfies the elbow property since $\delta_2 + \delta_1 < \beta$, which means that δ_1 is definitely less than β.

Now consider the dashed junction between the vertical line from q_1 and the horizontal line from q_4. From Lemma 4.4.1, we know that a minimal connection can be transformed to one that connects the string of unit spaced points to the right of q_4 with a horizontal line. Thus, the length of the stream coming into the junction from the right is $\alpha - \delta_1$ units long which is equal to the length of the vertical segment connecting the junction δ_1 units below q_1 to the junction δ_2 units to the left of q_4.

Consider the junction under q_4 at the horizontal level of q_3, shown by solid lines in Figure 4.4. The vertical segment is δ_1 units long and the horizontal segment is at least β long. Thus, the horizontal segment is at least as long as the vertical segment.

Consider the junction under q_2 and to the left of q_3. Lemma 4.4.1 lets us assume we connect the horizontal string of points with a single horizontal line. Thus, the line coming in from the right is of length β and the vertical line is of length β, so the elbow property is satisfied. Every junction satisfies the elbow property in the quadruped. ∎

Lemma 4.4.4 *Every grid location for the minimum connections of the quadruped contains at most one junction.*

Proof: There are four junctions in the quadruped, two for the solid line minimum connection and two for the dashed line minimum connection. In both minimum connections, these junctions are not co-located on the grid. ∎

Lemma 4.4.5 *The minimum connections of the quadruped do not contain crossovers.*

Proof: Figure 4.4 shows the two minimum connections, one as solid lines and one as dashed lines. As can be seen from the figure, there are no crossovers in the formation of the minimum connections. ∎

Theorem 4.4.6 *The length of the minimum connections of the quadruped is exactly equal to the server bandwidth necessary for the stream merging schedule indicated by the minimum connections.*

Proof: Because the quadruped satisfies the elbow property, there is no need for premature starts, and there are no additional lines for the stream merging schedule. All grid points have at most one junction, so vertical segments count exactly once toward the server bandwidth. There are no crossovers, so the minimum connection is a planar tree. Thus, the length of the minimum connections is exactly equal to the server bandwidth necessary for the merge schedule indicated by the lines in the minimum connection. ∎

4.5 Tiles

The proof proceeds in the same manner as in [65]. We create basic block tiles for variable vertices and auxiliary vertices in the planar 3SAT grid embedding. We create vertical not and horizontal not tiles to represent negation, OR tiles to represent the or operator, and clause tiles to represent clauses.

Each tile consists of overlapping quadrupeds. The choice of using the solid or the dashed minimum connection of one quadruped forces the rest of the connected quadrupeds to use the same choice for the overall minimum connection. Additionally, each tile has dimensions 96 by 96, except for the OR tile which has a width of 192.

Because the minimum connection is a set of rectilinear trees and we need to create a single rectilinear tree, we connect the "roots" of rectilinear trees in the minimum connection with strings of horizontal and vertical client arrivals in each tile. Similarly, once all the tiles are placed, we connect the roots of the trees in each tile to a client arrival positioned on a tile to its left and/or below. We showed before that a series of horizontally unit spaced

client arrivals can be connected via a single horizontal stream in a minimum connection. Connecting a vertical series of unit spaced clients is slightly more complex. Because we need to satisfy the elbow lemma for each junction, we instead place a double column of client arrivals side-by-side when connecting roots of trees in the vertical direction. If we have side-by-side double columns of unit spaced client arrivals in the vertical direction, then we can assume they are connected via the zipper connection in any minimum connection, as shown in Figure 4.5.

Lemma 4.5.1 The Zipper Lemma: *If the client set has 2 horizontally aligned columns of K unit spaced clients, then any optimal rectilinear merge tree containing these clients* either *connects the left set with a single vertical line and each client in the right set horizontally to the client to the left* or *the tree can be transformed to utilize this* zipper *connection. See Figure 4.5 for an illustration of the connection.*

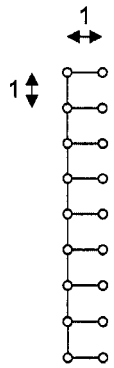


Figure 4.5: Illustrates the zipper connection for two columns of 9 horizontally aligned client arrivals.

Proof: Suppose we have an optimal rectilinear merge tree with $2K$ clients positioned as the lemma indicates and it does not already have a zipper connecting these clients. Then one of the following cases must happen:

Case 1: One or more of the clients (except for the bottom client) in the left vertically aligned set is not connected with a downward edge, but rather a horizontal line to the left. If this line is more than one unit long, the original rectilinear merge tree was not optimal.

Thus, the line must be one unit long and instead can be directed to the client one unit to the south. We can always create a junction to the south one unit away, since the client to the south has a client to its right. We do not increase the total cost by switching a unit length horizontal line to a unit length vertical line.

Case 2: One or more of the clients in the right vertically aligned set is connected to the client below instead of the client to its left. Because it is connected below, the junction must have a horizontal line coming from the right. If the horizontal line into the junction below is not long enough, we must use a premature start, causing an increase in the server bandwidth. Since the client is connected with a unit length line to the client below, we can change the connection to use a unit length line to the client to its left.

Thus, any optimal rectilinear merge tree with these clients can be transformed into an optimal tree where two columns of vertically aligned clients are connected with the zipper pattern. ∎

4.5.1 The Basic Block

The basic block tile, shown in Figure 4.6, is formed from overlapping quadrupeds. The reduction properties are still satisfied for the minimum connections of the basic block, so the length of the minimum connection is exactly equal to the total server bandwidth for the merge schedule associated with the minimum connection. Since the choice of the minimum connection for one quadruped forces the minimum connection for the others, there are two minimum connections for the basic block (one shown by solid lines and one shown by dashed lines). The differences between our basic block tile and the basic block in [65] are the following: 1. We remove the black points along the bottom if there is another basic block tile below, 2. We remove the grey points along the bottom if there is a vertical NOT tile below, 3. We remove the grey points on the left if there is a tile to the left, 4. We remove the grey points on the right if there is a tile other than a horizontal NOT tile on the right.

Unit spaced points along a horizontal line are connected with a horizontal line and the double columns of vertical points are minimally connected with the zipper pattern. The

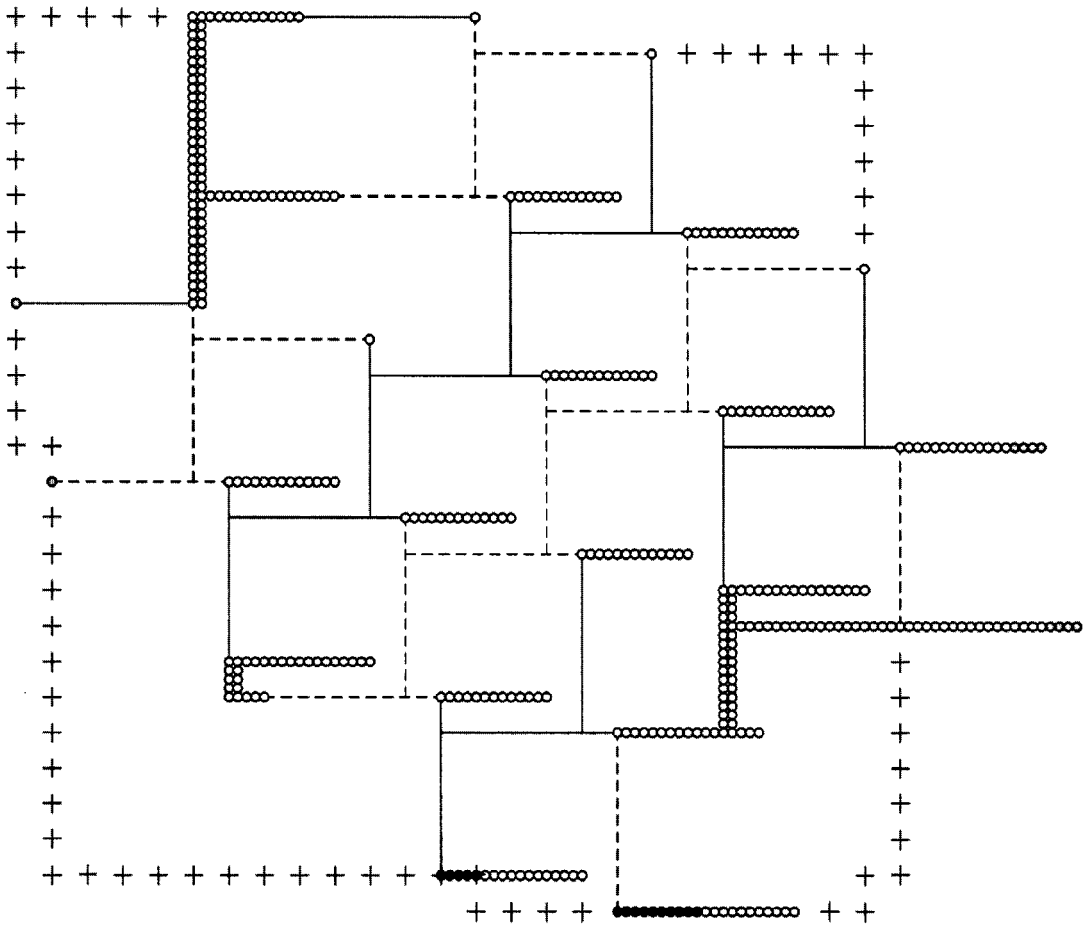


Figure 4.6: A basic block tile, where $\alpha = \beta = 20$ and $\delta_1 = \delta_2 = 4$.

crosses indicate the perimeter of the basic block tile. Notice that there are points to the right that are not confined to the boundary of the basic block tile. If there is no tile to the right of the basic block, these points do not interfere with any other points in that space on the grid. If there is a tile to the right (other than a horizontal NOT), the grey points are removed. The other tiles, as will be shown below, have a similar structure on the left side of the tile as the basic block. Consider two basic blocks side-by-side with the grey points on the right side of the left block removed. Similarly, the grey points on the left side of the right block are removed. When the two blocks are side-by-side, the string of points from the

left block are to the left of any point of the right tile, so the points outside the basic block tile do not interfere with points from other tiles. See Figure 4.7 to see how side-by-side basic blocks interlock. We show in the next section how a basic block connects side-by-side to a horizontal NOT tile.

Definition 4.5.1 *The* parity *of a basic block tile is 1 if the minimum connection uses a horizontal edge to connect to the top string of horizontal points escaping the right edge, and 0 otherwise.*

In Figure 4.6 the parity of the basic block tile is 1 if the solid lines are used and the parity is 0 if the dashed lines are used.

Lemma 4.5.2 *Consider two horizontally adjacent basic block tiles. In order to create a minimum connection across both tiles, the tiles must have the same parity.*

Proof: Because there is a block to the right of the left basic block tile, we remove the grey points extending to the right of the left tile. We also remove the two grey points on the left edge of the right basic block tile. For a minimum connection, the solid line on the right edge of the left tile must join a solid line on the right tile or the dashed line on the right edge of the left tile must join a dashed edge on the right tile. If the parity of the left tile is 1, then the parity of the right tile must be 1 in a minimum connection. Similarly, they could both have parity 0 for a minimum connection. See Figure 4.7. ∎

Lemma 4.5.3 *Consider two vertically adjacent basic block tiles. In order to create a minimum connection using both tiles, the tiles must have the same parity.*

Proof: Because we place one basic block tile on top of another basic block tile, the black points of the top tile are removed. To create a minimum connection, the solid lines must meet or the dashed lines must meet. Thus, the two tiles must have the same parity in a minimum connection. ∎

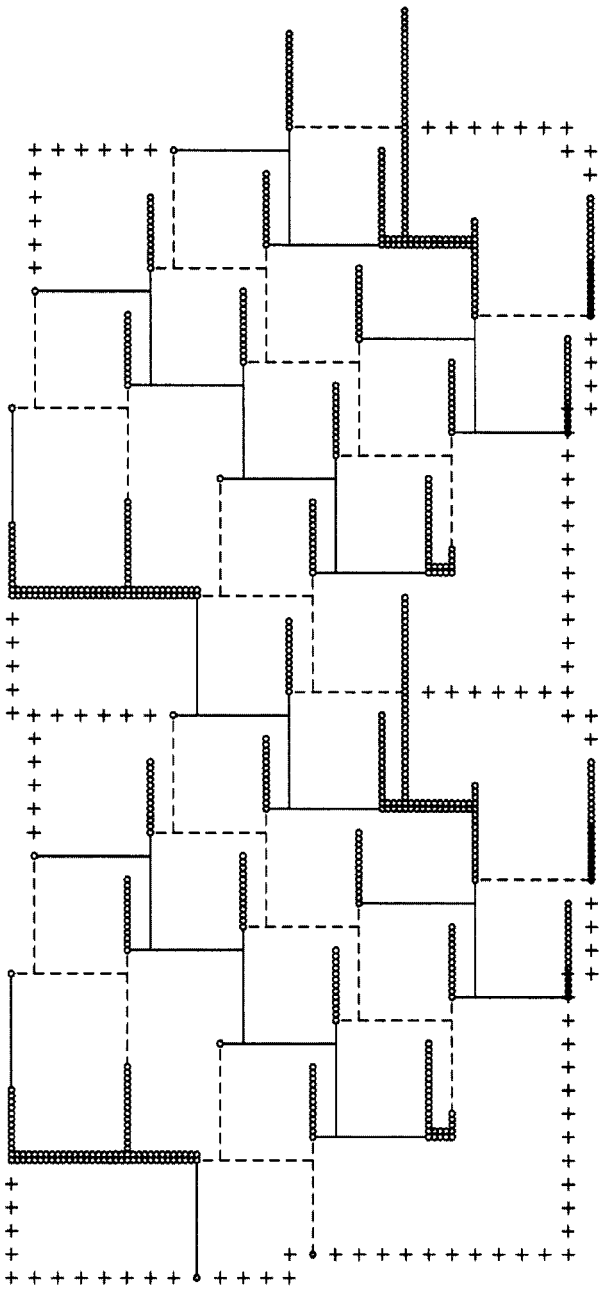


Figure 4.7: Two basic block tiles positioned side-by-side (Figure is rotated 90 degrees to the left)

4.5.2 The Negation Tiles

To represent negation, we create tiles that change the parity from one neighbor to the other. We design two tiles for negation – one that will change the parity in the horizontal direction and one that will change the parity in the vertical direction. Figure 4.8 shows the horizontal NOT tile. Notice that it has connections to the left and right. Solid lines indicate one way to form a minimum connection while dashed lines indicate another. A horizontal NOT tile has parity 1 if the dashed lines are used (the top string of points to the right of the tile is connected by a horizontal line), and 0 otherwise. The horizontal NOT tile is similar to the one in [65], except that all client arrivals are shifted four units upward.

Notice that the string of client arrivals extending to the right of the horizontal NOT tile always exist and there is always a tile placed to the right of a horizontal NOT tile. Notice also that part of the connections to the string of clients on the right are outside the tile. Just like side-by-side basic block tiles, these connections do not interfere with the points of the tile to the right. See Figure 4.9 for the placement of a basic block tile to the right of a horizontal NOT tile. Also, since $\alpha = 25$ in a horizontal NOT tile, when a basic block tile is to the left of a horizontal NOT tile, the string of horizontal points on the right edge of the basic block tile do not interfere with the points on the NOT tile. See Figure 4.9 for this illustration.

Lemma 4.5.4 *Consider a basic block placed to the left of a horizontal NOT tile. Then the minimum connection must have two different parities for the tiles. If a basic block is placed to the right of a horizontal NOT tile, then the two tiles must have the same parity.*

Proof: Consider a basic block tile that is placed to the left of a horizontal NOT tile. To create the minimum connection, the solid edge on the left side of the horizontal NOT tile must connect to the solid vertical edge on the right side of the basic block tile. Alternatively, the dashed edge on the horizontal NOT tile could connect to the dashed edge of the basic block tile. Since the solid edges of basic block tiles constitute parity 1, and using the solid horizontal edge on the left side of the horizontal NOT tile creates parity 0, the two tiles have different parity. Likewise, if the basic block tile on the left has parity 0, the horizontal NOT tile has parity 1.

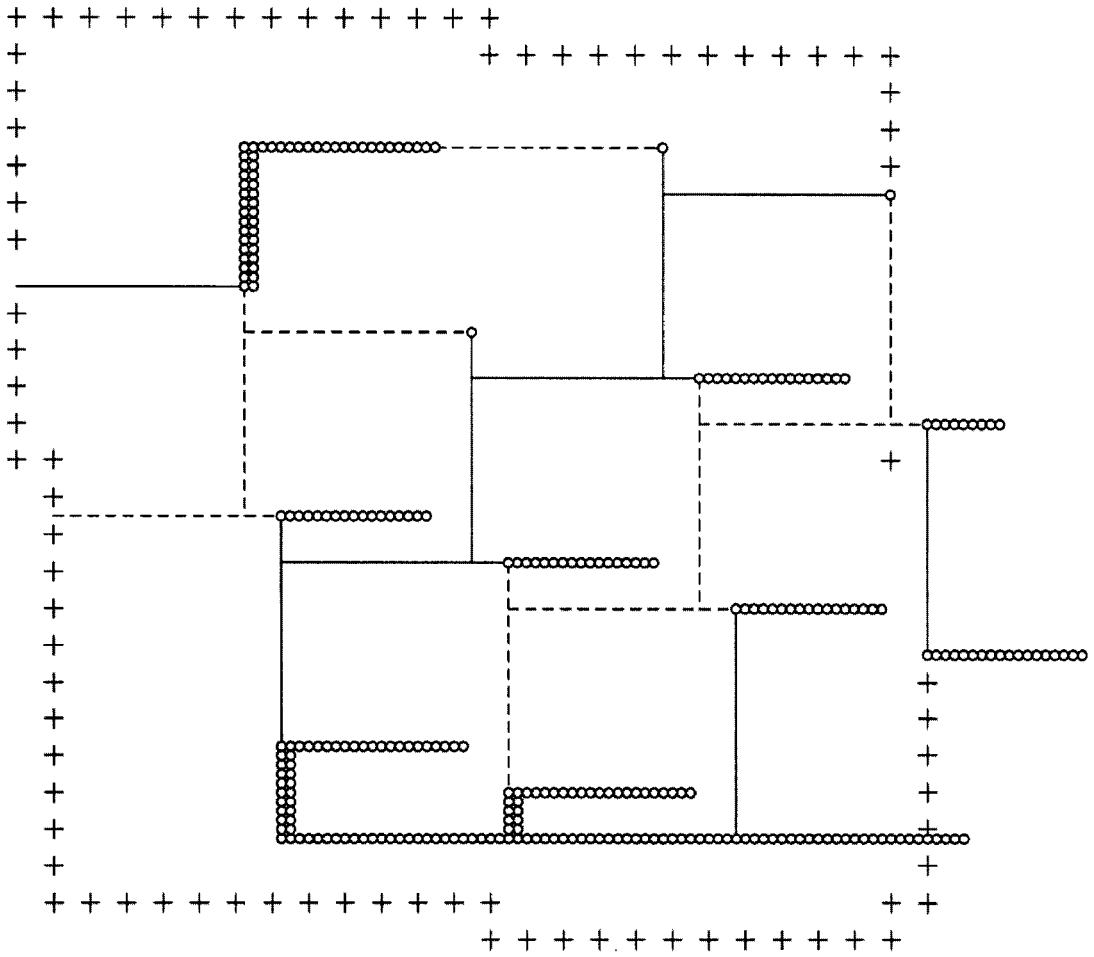


Figure 4.8: A horizontal NOT tile, where $\alpha = \beta = 25, \delta_1 = 5$, and $\delta_2 = 4$.

Now consider a basic block tile placed to the right of a horizontal NOT tile. To form a minimum connection, the dashed edge on the right edge of the horizontal NOT tile must join a solid edge of the basic block tile. Alternatively, the solid edge of the horizontal NOT tile could join a dashed line on the left side of the basic block. The dashed edges in a NOT tile constitute parity 1 and the solid edges of a basic block tile form parity 1, so the parities must be the same.

Figure 4.9 shows how the basic blocks connect to the horizontal NOT tile. The solid lines show the basic block to the left has parity 1 while the basic block to the right has parity 0. The solid lines indicate the horizontal NOT tile has parity 0. ∎

The vertical NOT tile is similar in function to the horizontal NOT tile, except that it connects to tiles above and below. Vertical NOT tiles have parity 1 if the top middle point is connected by a horizontal edge (solid lines) and parity 0 otherwise. Figure 4.10 shows the vertical NOT tile.

Lemma 4.5.5 *Consider a basic block tile below a vertical NOT tile. Then the minimum connection must have two different parities for the tiles. If a basic block is placed above a vertical NOT tile, then the two tiles have the same parity.*

Proof: This is shown by a similar argument as was shown for Lemma 4.5.4. ∎

4.5.3 The Clause Tile

We represent clauses with a different tile. From the reduction to the rectilinear version of the planar 3SAT instance, the clause vertices always have a vertex to the left and a vertex below. For the clause to be true, either the vertex on the left must be true or the vertex below must be false. Thus, our tiles will form connections to the left and below. The clause tiles will form a minimum connection when either the tile on the left has parity 1 or the tile below has parity 0. Figure 4.11 shows the clause tile. The grey points on the bottom are removed from the clause tile if there is a vertical NOT tile below. The solid horizontal line in the center is 24 units long.

Lemma 4.5.6 *To connect points $q0$ to $q5$ in a clause tile with a minimum length of 108, either the tile to the left has parity 1 or the tile below has parity 0.*

Proof: The points $q0$, $q1$, $q4$, and $q5$ can be connected with a total length of 80, using either the solid lines or the dashed lines. Assume $q1$ is connected with a solid vertical line. Then $q3$ can be connected via a horizontal edge of length 24 and $q2$ can be connected via a vertical edge of length 4. Assume $q4$ is connected with a horizontal solid line. Then $q2$ can be connected with the darker dashed line in Figure 4.11 with length 24 and $q3$ can be

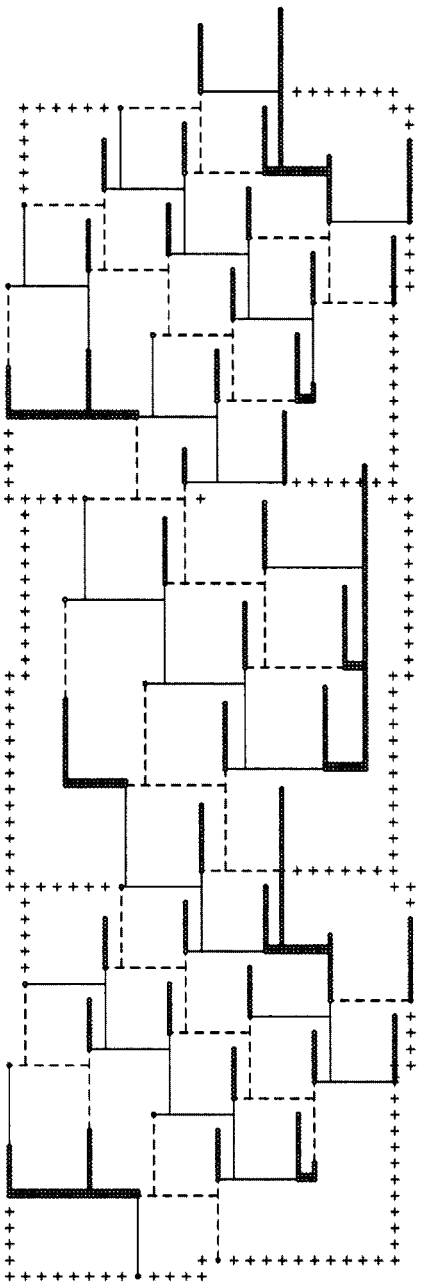


Figure 4.9: A horizontal NOT tile placed side-by-side with two basic block tiles (Figure rotated 90 degrees to the left). The basic block tiles have opposite parity in a minimum connection.

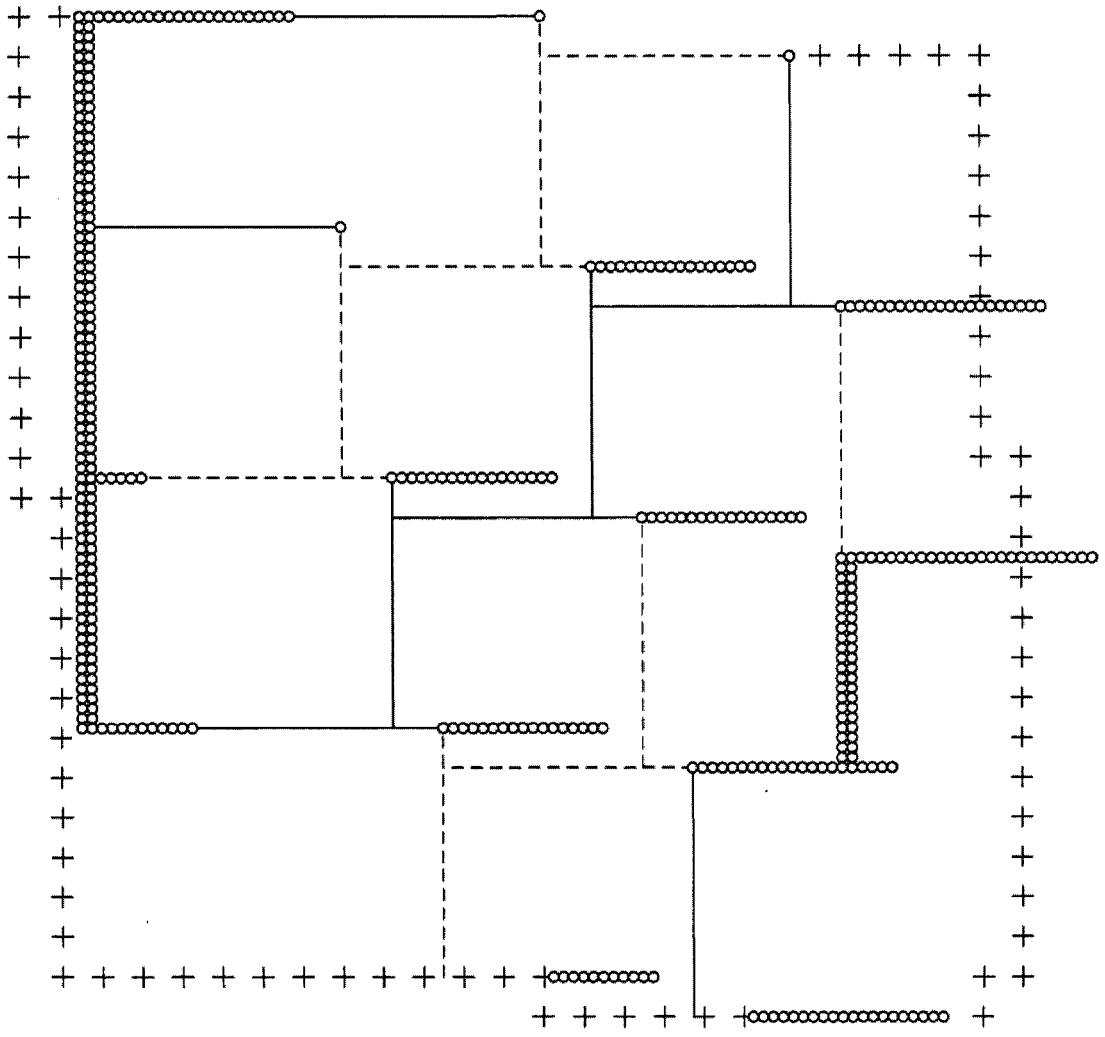


Figure 4.10: A vertical NOT tile, where $\alpha = \beta = 25, \delta_1 = 4$, and $\delta_2 = 5$.

connected horizontally with an edge of length 4. The solid edges total 108 in length. If $q1$ is connected via a horizontal edge (dashed line) and $q4$ is connected via a vertical edge (dashed line), then the total connection is 120. For $q1$ to be connected via a vertical solid edge, it must be connected to a tile on the left with parity 1. For $q4$ to be connected via a horizontal solid edge, the vertical solid edge to $q5$ must be used and the parity of the tile below must be 0. Either the tile to the left must have parity 1 or the tile below must have

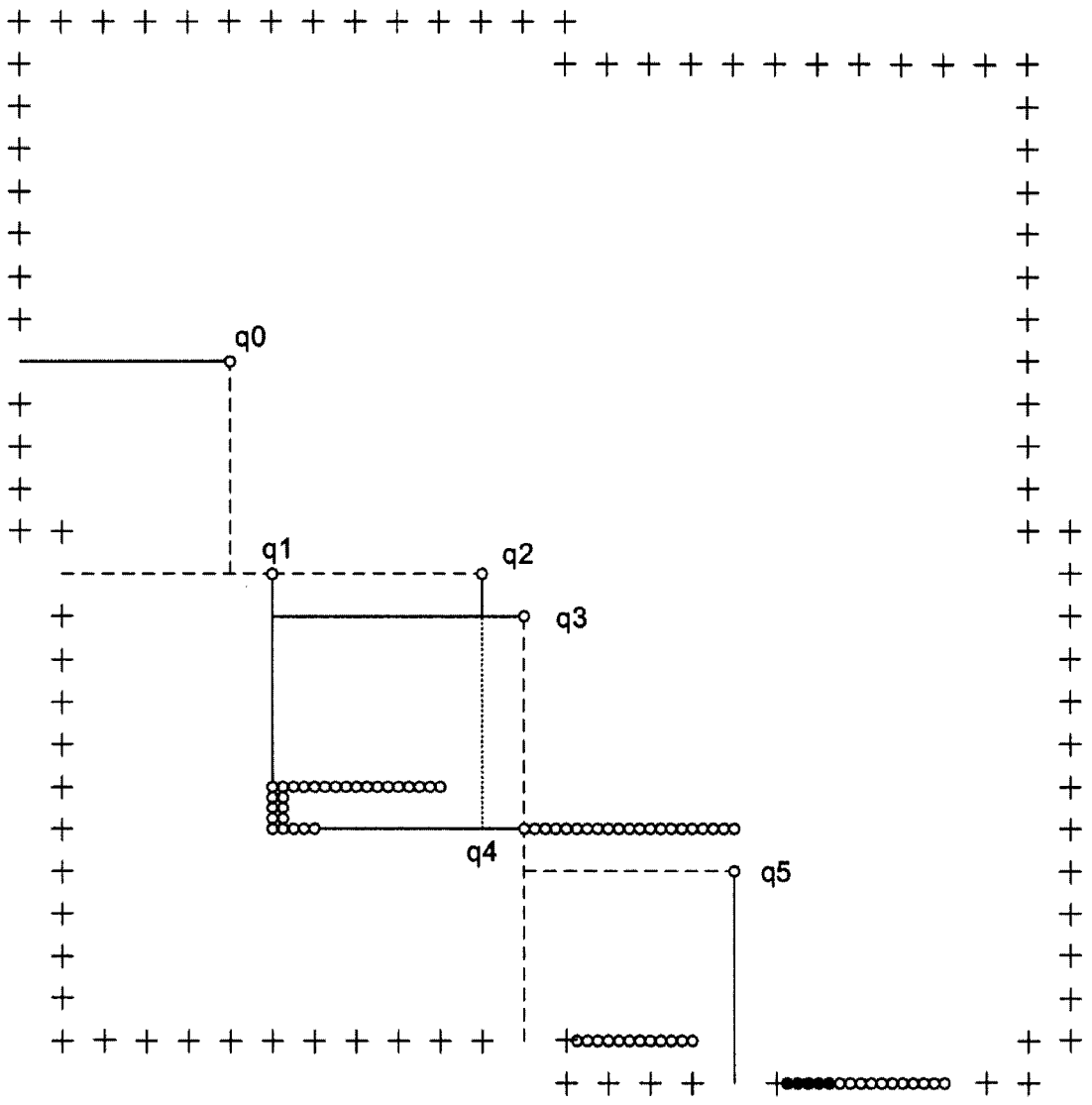


Figure 4.11: A Clause tile, where $\alpha = \beta = 20, \delta_1 = 4$, and $\delta_2 = 4$.

parity 0 for the clause tile to have the minimum connection of 108. ∎

4.5.4 The OR Tile

We also need to represent the or operation from the original planar 3SAT instance as a tile. The OR tile represents the boolean operation "or" on two tiles and the submission of the result to a clause tile. The OR tile differs from the others in that it has double the size in width, so it has height 96 and width 192. The left side of the OR tile acts like the clause tile, while the right side allows the OR tile to have double width. Figure 4.12 shows the OR tile. The grey points are removed from the bottom edge of the tile if it is connected to a vertical NOT tile from below. The parity of the OR tile is 1 if the top string of points extending to the right of the tile is connected by a horizontal line (solid edges) and 0 otherwise. Note that a minimum connection using the dashed lines can always be formed, but we are simply concerned if the OR tile can have a minimum connection with parity 1. A clause tile is always to the right of an OR tile, so the clause tile will achieve its minimum connection if the OR tile on its left has parity 1 or the tile below has parity 0.

Lemma 4.5.7 *If the parity of the minimum connection of an OR tile is 1, then either the tile to its left has parity 1 or the tile below has parity 0. Conversely, if the tile to the left has parity 1 or the tile below has parity 0, a minimum connection with parity 1 can be formed.*

Proof: Suppose the minimum connection of an OR tile is 1. Then we know the solid edges must be used in Figure 4.12. Like the clause tile, to use the solid lines, we must have a tile to the left with parity 1 or a tile below with parity 0 (using the heavy dashed line instead of the 20 units on the left side of the line with length 36).

Now assume the tile to the left has parity 1. Then we must connect to that tile using the solid horizontal line on the left of the OR tile. This forces the minimum connection to use the solid lines, and thus have parity 1. If the tile below has parity 0, then we can use the heavy dashed line and the other solid lines (except for the first 20 units of the solid line of length 36) to form the minimum connection. This creates an OR tile with parity 1. ∎

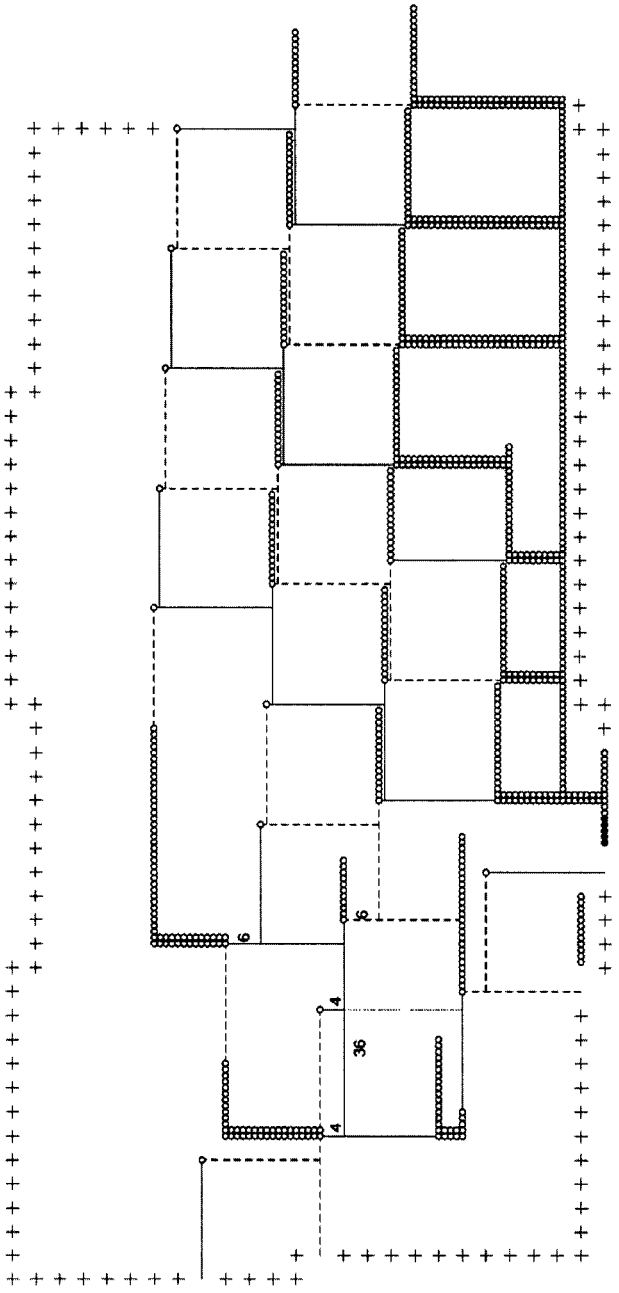


Figure 4.12: An OR tile, where $\alpha = \beta = 20, \delta_1 = 1$, and $\delta_2 = 4$ (unless noted). (Figure rotated by 90 degrees to the left) The width is 192 and the height is 96.

4.6 *Stream Merging with Time-Shifting Instance Generation*

We have defined the tiles to generate a rectilinear merge tree instance from the embedded rectilinear grid instance. We reduce from planar 3SAT to the embedded grid instance, just as in [65]. For each vertex in the embedded grid instance, we replace it with a tile. One unit in length in the vertex embedding becomes 96 units of length in the tiled rectilinear embedding. Variable and auxiliary vertices are replaced with basic block tiles. Each not vertex that is connected to vertices on the left and right is replaced by a horizontal NOT tile. Each not vertex that is connected to vertices above and below is replaced by a vertical NOT tile. Each pair of or vertices is replaced by a single OR tile and each clause vertex is replaced by a clause tile. Because the embedded instance is such that adjacent vertices are exactly one unit apart and non-adjacent vertices are more than one unit apart, tiles are connected only if the original vertices were connected.

In the time-shifting problem, the client arrivals are always positioned on or below the zero line. Thus, our tiles must be within this boundary, too. Let T_l be the left-most tile in the embedding, T_r be the right-most tile, T_t be the top-most tile, and T_b be the bottom-most tile. We'll denote the horizontal position of T_l as 0 and the vertical position of T_b as 0. Define H to be the horizontal position of T_r and define V to be the vertical position of T_t. Find the x-axis crossing for the line that goes through (H, V) with slope $-\frac{1}{2}$. Let s be this x-axis crossing. In the rectilinear grid containing the tiles, add a unit spaced string of horizontal client arrivals 4 units below T_b and 4 units to the left of T_l for a string of $96 * (s + 1) + 5$ client arrivals. The "end" of the live stream will be positioned at the origin of the grid. Find the top-most tile in the left-most column of the grid. Add a double column of unit spaced vertical clients from the origin to the top of the top-most tile in the grid. Now all client arrivals are within the boundary of the triangle created by the root stream, the vertical double column of client arrivals on the left edge, and the zero line. Figure 4.13 shows how tiles, the root stream, and the double column of vertical points are placed on the grid.

We now need to connect the tiles to each other, to the root stream, and possibly the double column of vertical client arrivals above the origin. We will connect the tiles with

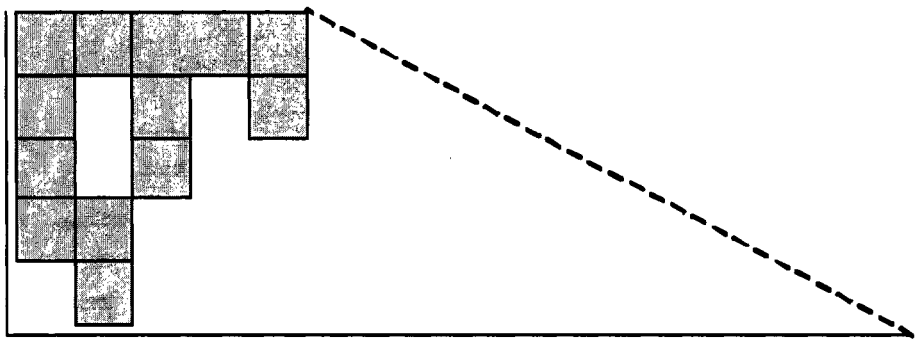


Figure 4.13: Illustrates how tiles are placed within the boundary for client arrivals in the stream merging with time-shifting problem. The root stream (shown as the horizontal line) is placed 4 units below the bottom tiles. The double column of vertical clients (shown as the vertical line) is placed 4 units to the left of the left-most tiles. The dashed line indicates the zero line that serves as the top boundary for client arrivals.

strings of unit spaced horizontal clients and double column unit spaced vertical clients. Find the clients in each tile that do have any client arrivals to the left or below in the same tile. We will call these clients tile roots, since they form the roots of a set of trees for that tile. For example, if a basic block tile is connected to a tile below, then the three tile roots that need to be connected are the two grey points on the left edge of the tile in Figure 4.6 and the southwest point of the cluster of points in a "C" shape in the southwest part of the tile. For each tile, find the tile roots and connect them vertically via double columns going down and horizontally to the left via single rows of clients to any client arrival to the left and/or below without crossing a path consisting of solid or dashed edges.

Figure 4.14 shows a complete instance. The grey lines represent strings of client arrivals connecting the tiles, the root stream, and the left-most double column of client arrivals. The black lines represent the minimum connection and the thicker black lines represent strings of client arrivals in the original tiles. The bottom grey string of client arrivals continues to the right so that the client arrivals are in the appropriate boundary. The planar 3SAT instance represented in Figure 4.14 is $(v_1 \vee v_2 \vee \overline{v_3}) \wedge (\overline{v_1} \vee \overline{v_4})$, and is a direct tile substitution for the graph shown in Figure 4.3.

Theorem 4.6.1 *The decision version for the optimal time-shifting stream merging problem*

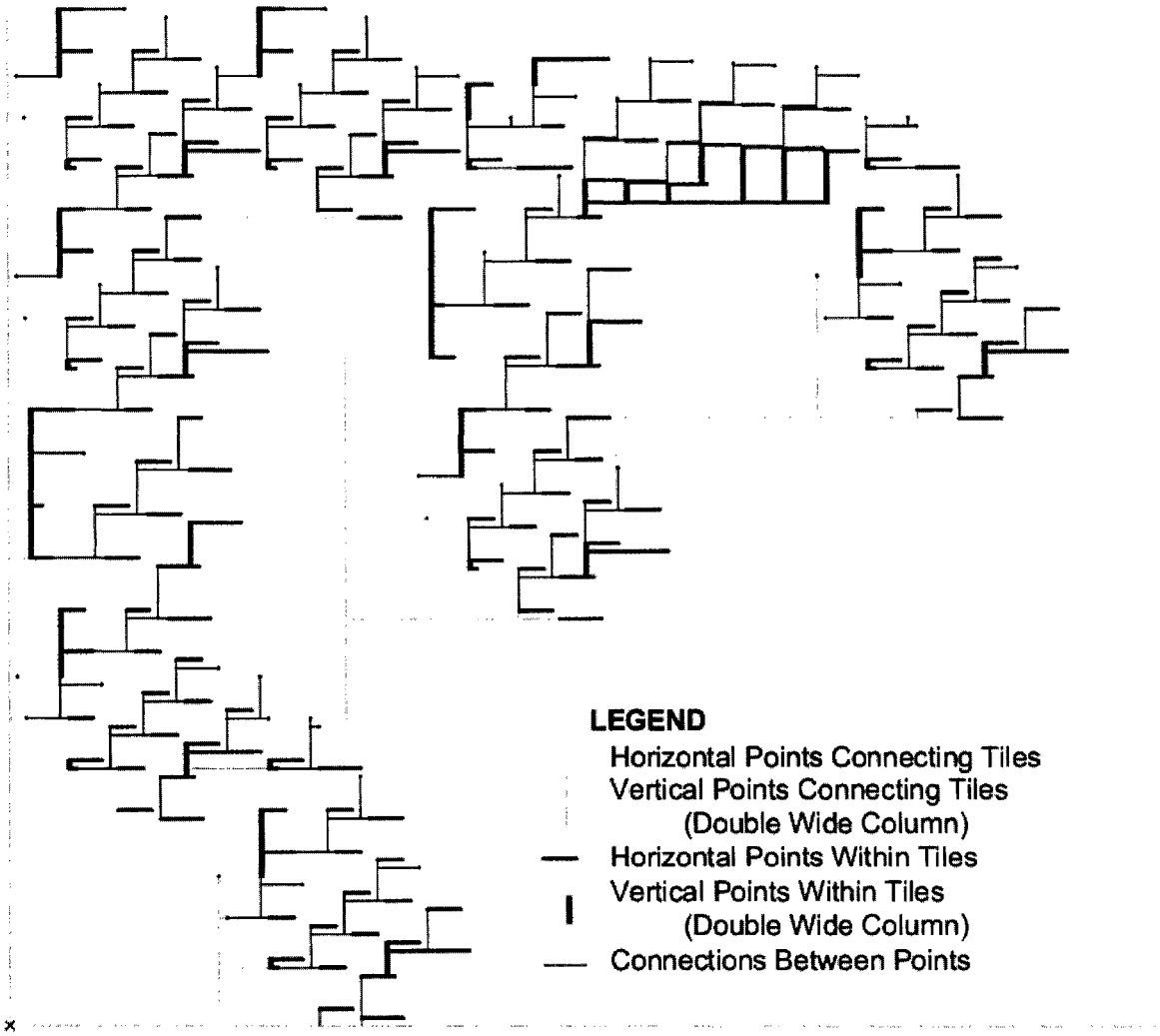


Figure 4.14: A time-shifting rectilinear merge tree instance and its minimum connection formed from the planar 3SAT instance $c_1 = (v_1 \vee v_2 \vee \overline{v_3})$ and $c_2 = (\overline{v_1} \vee \overline{v_4})$. The satisfying assignment represented here is $v_1 = true$, $v_2 = true$, $v_3 = false$, $v_4 = false$.

is NP-complete.

Proof: We have shown above how to transform an instance of planar 3SAT to an instance of the stream merging with time-shifting problem. Let L be the length of the minimum connection for all the strings of client arrivals, both horizontal and vertical, that are not

included in the original tiles. Add to L the sum of the minimum connection (solid or dashed lines) for the basic block tiles, the horizontal NOT tiles, the vertical NOT tiles, and the OR tiles. Then, all that is left is the sum of the minimum connections for the clause tiles. We claim the set of client arrivals has a rectilinear merge tree with total bandwidth of $L+108m$ (where m is the number of clause tiles) if and only if the planar 3SAT instance is satisfiable.

First, let us assume the planar 3SAT instance is satisfiable. Then, there is a satisfying assignment for each variable. For each variable's truth value, we make a minimum connection in its corresponding basic block to have parity 1 if the variable is true and parity 0 if the variable is false. These assignments will force the minimum connections of adjacent basic block, horizontal NOT, and vertical NOT tiles. All clauses are satisfied in the planar 3SAT instance, so we can make minimum connections of 108 for each clause tile and force adjacent OR tiles to create minimum connections with the correct parity. Every clause tile has length 108, so the minimum connection has cost $L+108m$ which means the rectilinear merge tree has server bandwidth of $L+108m$.

Assume that the set of client arrivals has a rectilinear tree with server bandwidth of $L+108m$. Each tile must have connections to all of its client arrivals, so each tile must use either the solid lines or the dashed lines for the connection. Because L is the length of all strings of clients, the connections in the basic block tiles, the NOT tiles, and the OR tiles, the connections in the clause tiles must sum to $108m$. There are m clause tiles and each has a minimum connection of 108, so each clause tile must be connected minimally with 108. Thus, for each clause tile, either the tile to the left has parity 1 or the tile below has parity 0. If the tile to the left is an OR tile and it has parity 1, then its tile to the left has parity 1 or its tile below has parity 0. So, for the OR tile to have parity 1, one of the arguments to the or operator must be true. If the tile below the clause tile has parity 0, then that variable (or negation of the variable if there is a vertical NOT tile there) is true. So, one of the variables in the clause must be true for the clause tile to have a minimum connection of 108. Hence, there is a satisfying assignment for the original planar 3SAT instance.

We have a reduction from planar 3SAT to the optimal time-shifting problem. The reduction is polynomial time, since the number of vertices in the rectilinear embedding of a planar 3SAT instance is polynomial in the number of vertices and clauses in the original

instance. The tiles are a direct substitution for the vertices in the rectilinear embedding.

The optimal time-shifting problem is in NP, since given rectilinear merge tree and a positive integer K, we can verify if the total bandwidth cost for the rectilinear merge tree is less than or equal to K. Therefore, the decision version for the optimal time-shifting problem is NP-complete. As a result, finding the optimal time-shifting merge schedule is NP-hard. ∎

4.6.1 Restricted Versions

Because an online algorithm would never start a stream before a client arrival, it is useful to find the optimal offline solution for time-shifting without premature starts. The proof above does not make use of premature starts, so the decision version of the optimal offline time-shifting problem without premature starts is still NP-complete.

Theorem 4.6.2 *The decision version of the optimal offline time-shifting problem without premature starts is NP-complete.*

Similarly, the proof did not include crossover streams. In other words, each time horizontal and vertical streams intersect, they meet at junctions. Therefore, the decision version for finding the optimal solution without crossovers is NP-complete.

Theorem 4.6.3 *The decision version of the optimal offline time-shifting problem without crossovers is NP-complete.*

4.7 Conclusions

We have shown how to reduce the planar 3SAT problem to the optimal offline time-shifting problem for a set of client arrivals. The proof required defining the time-shifting problem as finding the minimal connections among clients positioned in a rectilinear grid. Using the rectilinear grid allowed us to use a proof technique inspired by the one used to show the Rectilinear Steiner Arborescence problem is NP-complete. We had to modify the component tiles so that all vertical segments meet at junctions with sufficiently long horizontal segments. These modifications to the tiles did not interfere with the placement of adjacent tiles. We

showed how to map the tiles into the boundaries of the rectilinear embedding of clients for stream merging with time-shifting. Using this embedding, the minimal connection length is exactly equal to the server bandwidth.

Even though it is unlikely that we can find an efficient algorithm to determine the optimal server bandwidth cost in the time-shifting framework, we can still devise online algorithms, since the system must make decisions when clients arrive. Developing and experimentally testing online algorithms for time-shifting is the topic of the next chapter.

Chapter 5

MEDIA-ON-DEMAND WITH TIME-SHIFTING ALGORITHMS

In the previous chapter we showed that finding the optimal offline algorithm for stream merging with time-shifting is NP-hard. Even though it is probable that we will not find an efficient solution for calculating the optimal bandwidth, the problem of scheduling streams in an on-demand, online manner can be addressed. In this chapter we introduce and compare approaches for delivering online streams to clients in the time-shifting case. We also consider the adaptation of the proposed algorithm for time-shifting for the standard media-on-demand model.

Before presenting online algorithms for the time-shifting case, let us review the properties and definitions of the model. Recall that each client has a arrival time t and a requested first segment f. We indicate clients by pairs (t, f). There is a single, continuous, live broadcast, which we assume starts at time 0. This live broadcast does not have an ending time. We use both the standard merge figure diagrams (where streams are represented by diagonal lines) and the rectilinear merge diagrams which were introduced in Chapter 3 We assume the live media is divided into segments of equal playback duration. Under this assumption, the values for (t, f) pairs are non-negative integers. We assume the *receive-two* model, where clients can receive two streams simultaneously. In addition, we assume the server can multicast streams to clients.

5.1 Merge Once To Live Stream

A first and simplest approach to support online stream merging for time-shifting is to deliver streams to each client that fill the time between their requested first segment f and the current segment being broadcasted by the live stream. For example, assuming the live broadcast starts at time 0 with segment 0, consider the necessary stream for a client arriving at time 5 requesting segment 1. The server could distribute a stream with segments

1 through 4 to the client while the client also receives the live stream. At time 9, the client merges to the live stream and the dedicated stream to the client can terminate. These actions for client (5,1) are shown in Figure 5.1.

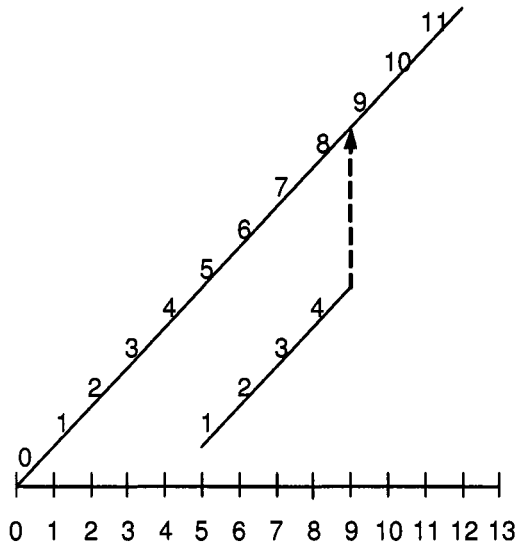


Figure 5.1: Example of the client (5,1) merging directly to the live stream at time 9. The clients receives segments one through four from its dedicated stream while receiving segments 5 through 8 from the live stream.

We call this approach *merge once* since each client simply merges once directly to the live broadcast (the root stream). We use the merge once algorithm as a source for comparing other online algorithms for the time-shifting case. We do not have an efficient way to compute the optimal offline solution for the time-shifting case, so we look at the improvement of online algorithm solutions over the merge once solution.

5.2 Dyadic Algorithm for Time-Shifting

Goshi describes an online algorithm for the time-shifting case which is a modification of the dyadic algorithm for the original media-on-demand model [38]. His version of the dyadic algorithm differs from the standard dyadic algorithm in two ways: 1) the start times for dyadic intervals are not client arrival times but instead artificial arrival times given by $t - f$,

and 2) the algorithm uses a list to store intervals instead of a stack. In this manner, client A can merge to client B if the artificial arrival time for A is less than or equal to the artificial arrival time for B. Because artificial arrival times are not monotonically nondecreasing as clients arrive to the system, the algorithm uses a list to keep track of all the intervals. Recall that in the original dyadic algorithm, the intervals are stored in a stack. It is less efficient than the original dyadic algorithm because the algorithm may need to look through an entire list of intervals without removing them.

The dyadic algorithm is as follows: [38]
Let L be a list of dyadic intervals $[t_a, t_r)$. At time t of a client request, find the artificial arrival time $a = t - f$. Search from the front of L to find an interval $[t_a, t_r)$ such that $t_a \leq a < t_r$ and the stream represented by the interval $[t_a, t_r)$ has not yet terminated. While looking for such an interval, remove all intervals whose streams terminated at or before time t.

After searching through L, there are two possibilities:

1. There is no such interval $[t_a, t_r)$. Then the client arriving at time t will merge directly to the live stream and the interval $[a, \infty)$ is placed at the back of L.

2. Let s be the corresponding root stream for $[t_a, t_r)$. Create the interval $[a, r)$ by finding r:
   $r = t_a + (t_r - t_a)\max\{\alpha^{-k+1} : \alpha^{-k}(t_r - t_a) < \alpha - t_a\}$
   The arrival at t merges to the stream s and the interval $[a, r)$ is inserted immediately in front of $[t_a, t_r)$ in L.

The above description does not specify the order in which to process all the clients arriving at time t. Recall that in the time-shifting case, multiple clients can arrive at time t, each requesting a different first segment f. We make the assumption that clients arriving at the same time will be processed according to increasing requested first segments.

The dyadic algorithm for time-shifting also uses ∞ as the endpoint of an interval. In the implementation of the algorithm, we can assume the smallest unit as one segment and calculate powers of α such that $\alpha^i \leq a < \alpha^{i+1}$ and use α^{i+1} as the right-hand value instead

of ∞ for the interval. Additionally, we assume α is equal to 2 unless otherwise specified in this chapter.

5.2.1 Dyadic Algorithm Example

Here we describe how the dyadic algorithm for time-shifting creates merge schedules. Let us assume the live broadcast starts at time 0 with segment 0 and $\alpha = 2$. Consider the case of three arrivals: $a = (6, 0)$, $b = (9, 0)$, and $c = (10, 3)$. Since there are no clients in the system when client a arrives, we create the dyadic interval $[6, 8)$ for the arrival and a merges directly to the live stream. Client b arrives at time 9 and another interval $[9, 16)$ is created and put on the end of the list. At this point, the list L has two intervals and both a and b merge to the live stream. At time 10, c arrives. The artificial arrival time for c is 7 and 7 falls into the interval $[6, 8)$. Thus, client c merges to a and a new interval $[7, 7)$ is put in L in front of $[6, 8)$. In summary, a merges directly to the root stream, b merges directly to the root stream, and c first merges to a before merging to the root stream.

5.2.2 Deficiencies

One drawback of the dyadic algorithm for the time-shifting case is that a stream may be extended indefinitely under certain circumstances. Since the live stream is continuously broadcasted, it might be more advantageous to have clients merge to the live stream rather than continually extending another stream s as more clients merge to s. We showcase this deficiency with an example.

Extending Stream Example: Consider the arrivals $a = (4, 0)$, $b = (6, 0)$, $c = (8, 2)$, $d = (10, 4)$, and $e = (12, 6)$. Assume that $\alpha = 2$ in the dyadic algorithm. When a arrives, the interval $[4, 8)$ is created with a stream ending time of 8. At time 6, b arrives. When searching through the list L, we find the interval $[4, 8)$ whose stream has not yet terminated. Thus, b merges to a and we add the interval $[6, 6)$ to L. At time 8, $c = (8, 2)$ arrives to the system. At this point, $L = \{[6, 6), [4, 8)\}$ and the ending times for the intervals are 8 and 12. The artificial arrival time for c is 6 and the stream for interval $[6, 6)$ ends at 8, so c merges to a and the interval $[6, 6)$ is added to L. The pattern continues for arrivals d and e. The

complete merge schedule is illustrated in Figure 5.2. Notice that the stream started at a's arrival continues to be extended as clients b through e merge to the stream for a. Instead of following the schedule created by the dyadic algorithm, it would be more advantageous to have a merge directly to the root stream and have a single stream that transmits data to b through e and have this single stream merge to the root stream.

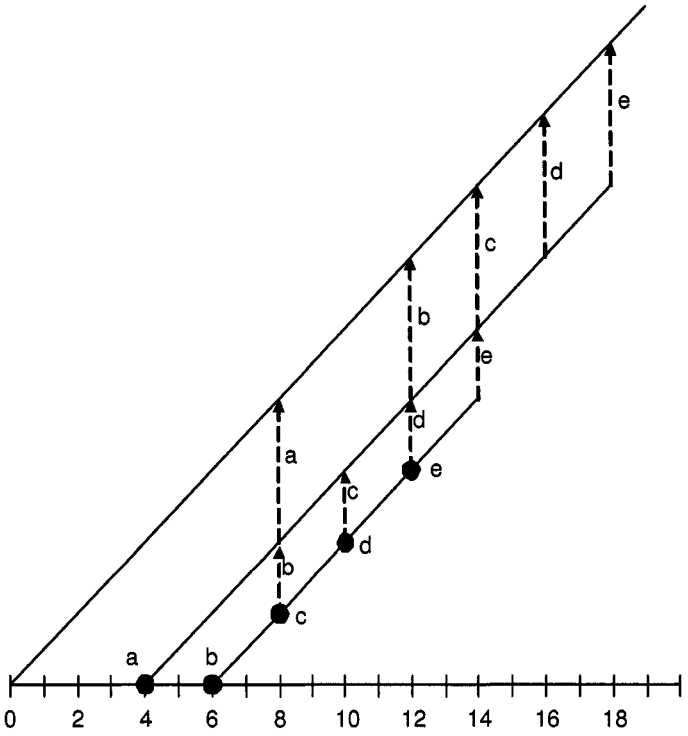


Figure 5.2: Example of the dyadic algorithm where the stream started for client $a = (4, 0)$ gets extended for clients $b = (6, 0)$, $c = (8, 2)$, $d = (10, 4)$, and $e = (12, 6)$ to merge to a's stream. Filled circles indicate client arrivals and the dotted lines illustrate clients merging to other streams.

A second drawback of the dyadic variation for the time-shifting case is that the entire merge schedule for a client is determined at the client's arrival time to the system. When a client arrives, it knows its entire merge sequence and each merger happens to a stream that already exists in the system. Recall from our earlier discussion that the time-shifting case differs from the standard media-on-demand case because it might be advantageous for

clients to merge with clients who have not yet arrived to the system. The dyadic algorithm does not take into account this possibility of merging with future client arrivals. We show an example below where the dyadic algorithm behaves exactly the same as the merge once approach.

No Client Mergers Example: Assume the arrival set is $a = (10, 0)$, $b = (11, 3)$, $c = (12, 6)$, and $d = (13, 9)$. Using the dyadic algorithm with $\alpha = 2$, when client a arrives to the system, no other streams exist so a merges directly to the root stream and $L = \{[10, 16)\}$. When b arrives, its artificial arrival time is 9 which does not fit into an existing interval, so b merges directly to the root stream. L is now $\{[9, 16), [10, 16)\}$. This pattern continues when c arrives and d arrives, so each arrival merges directly to the root stream. In this case, the dyadic algorithm produces a solution that is equal to the merge once solution. Notice that a could make use of b's stream, but the dyadic algorithm does not schedule mergers to streams that have not yet commenced. Figure 5.3 shows the merge schedule for this example.

5.3 Rectilinear Algorithm

We propose an online algorithm for the time-shifting case that takes advantage of the rectilinear merge figures presented in the previous chapters. Hence, we call this approach the *rectilinear* algorithm for the time-shifting case. Recall that we can embed client arrivals on a grid and streams are represented by vertical and horizontal lines. We use this grid to construct a greedy event-driven algorithm where clients only know their most immediate merger. Clients' merge targets could also change due to new arrivals to the system. In this fashion, clients have the opportunity to merge to clients not yet in the system. A client's entire merge schedule is dynamically created until it merges to the live stream.

The algorithm is based on grid points in the rectilinear grid. Each stream has an associated grid point and direction (vertical, horizontal, or not yet assigned), and keeps track of the grid location for the merger. When looking at the rectilinear grid, each diagonal line (northwest corner to southeast corner) represents the system at a fixed time t. The algorithm greedily schedules mergers based on active streams (active grid points) in the

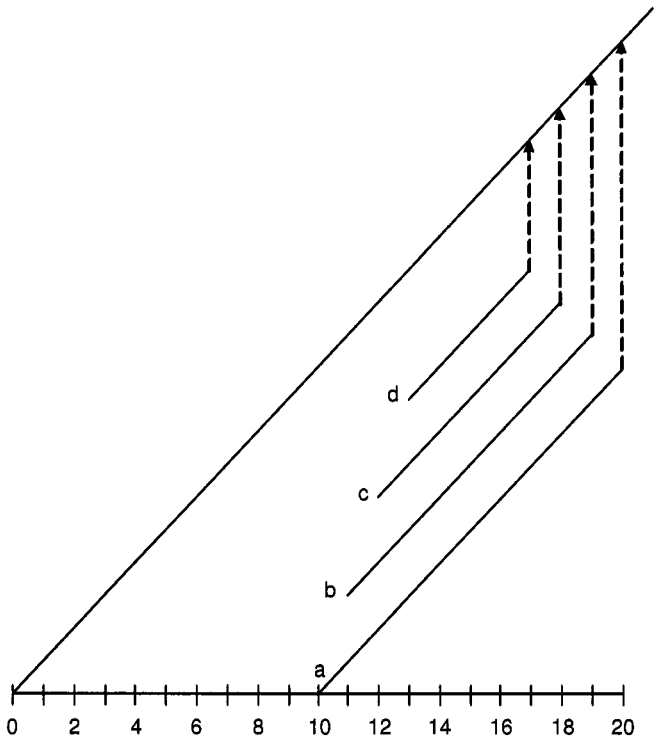


Figure 5.3: Example of the dyadic algorithm where the scheduled streams equal the streams for the merge once solution. The clients are $a = (10, 0)$, $b = (11, 3)$, $c = (12, 6)$, and $d = (13, 9)$. Notice that each stream merges to the root stream because the artificial arrival times decrease for each arrival to the system.

system at every time increment. Rectilinear (Manhattan) distances between grid points are used in the scheduling. We schedule the closest pair of grid points to merge first and continue until every grid point has been scheduled for a particular point in time. The algorithm makes a sweep across these diagonal lines representing a single time point, starting from the right and continuing to the left until all clients merge to the root stream.

The rectilinear algorithm operates in two distinct phases. At each time t, the first phase processes mergers that happen at time t. At a merge point, at least one stream is terminated. The second phase processes all the streams for the next time increment. At the end of the second phase, all grid points lie on the diagonal line for $t + 1$ and have directions set as vertical or horizontal. Because decisions about mergers are made at each time point t,

we are guaranteed that each stream can merge to any stream below itself on the rectilinear grid.

Each stream is represented by a grid point G on the rectilinear grid. Each grid point has the following properties:

- $L(G)$ – the current location on the grid. The location has two parts, a time and a segment. These are represented as $L_t(G)$ and $L_s(G)$.

- $D(G)$ – the current direction of the stream (horizontal (H), vertical (V), or not assigned (N))

- $ML(G)$ – the grid point location $(ML_t(G), ML_s(G))$ at which the stream will merge to another stream. This is assigned only when $D(G)$ is H or V.

- $T(G)$ – the grid point G' to which G is set to merge. This is defined only when $D(G)$ is V.

The algorithm proceeds as follows. At each time t where time increases by unit intervals, we perform the following two phases in order. Phase one processes the mergers that happen at time t. Phase two sets the directions and updates the streams for the next time marker $t+1$. Grid points G_i are stored in an ordered list L, where L is sorted by increasing $L_s(G_i)$ values.

Phase One: Process Mergers

1. Find all grid points G in L such that $ML_t(G)$ equals t. If $D(G)$ is V, remove G from L. If $D(G)$ is H, then assign $D(G)$ to N.

Phase Two: Set Directions

1. For each grid point G in L where $D(G)$ is V and the length of the stream already distributed is at least as long as the stream extension to reach its merge target, set $L_t(G)$ to $L_t(G)+1$ and $L_s(G)$ to $L_s(G)+1$.

Table 5.1: Shows the distance between grid points for the rectilinear algorithm. The distances are based on the stream directions of the grid points.

$D(G_i)$	$D(G_{i+1})$	Distance
N	N	$2(L_s(G_{i+1}) - L_s(G_i))$
N	H	$2(L_s(G_{i+1}) - L_s(G_i)) - (ML_s(G_{i+1}) - L_s(G_i))$
N	V	$2(L_s(G_{i+1}) - L_s(G_i))$
H	N, H, or V	∞
V	H	∞
V	N	$2(L_s(G_{i+1}) - L_s(G_i))$
V	V	∞

2. Let L_T be the subset of the list L such that all grid points G in L_T have $L_t(G)$ equal to t. Calculate the distance between successive grid points G_i and G_{i+1} using Table 5.1. For the last grid point G_f in L_T, the distance is the distance to the root stream, which is $L_t(G_f) - L_s(G_f)$. Find the grid point G_i with the smallest distance. (If there are ties, choose the latest such stream listed in L_t.) Update G_i by setting $D(G_i)$ to V and incrementing $L_t(G_i)$ and $L_s(G_i)$ by one. If G_i is not G_f, let G_{i+1} be the stream immediately after G_i in L_T. Set $D(G_{i+1})$ to H and calculate and set the values $ML_t(G_i)$, $ML_t(G_{i+1})$, $ML_s(G_i)$ and $ML_s(G_{i+1})$ to the grid location where the streams will merge. Finally, we remove G_i and G_{i+1}, if G_{i+1} exists, from L_T and put them in L. We repeat this step until there are no grid points G with $D(G)$ equal to N in L_T.

3. For any remaining grid point G in L_T, increment $L_t(G)$ and $L_s(G)$ by one and put it in L.

4. Set t to $t+1$ and repeat phases one and two after adding arrivals that come at time $t+1$ to L.

5.3.1 Rectilinear Algorithm Details

Keeping L Sorted

In the algorithm, we assume the list L of grid points is sorted, first by time and for those with the same time, by increasing segment. In the implementation, we keep L sorted at all times. When new arrivals come to the system at time t, we simply insert them into positions to keep L sorted. The list L_t is also kept in sorted order at all times in the implementation. By keeping L sorted at all times, we do not need to sort the list at the beginning of each time interval.

Setting Directions and Merge Targets

During the algorithm execution, a grid point's direction $D(G)$ could be reset. Let G be a grid point such that $D(G)$ is N. Because grid points with no direction do not have merge targets, we do not need to do anything other than set the merge location for these points.

Once a grid point's direction is set to H, its direction does not get reset until it reaches its merge location and its direction is set to N. During algorithm execution, a grid point with direction H could have more than one vertical grid point eventually merging to it. In this case, we keep track of the merge location furthest in the future (furthest to the left on the rectilinear grid).

Keeping track of grid points with directions set to V requires more bookkeeping. A grid point with direction V could be reset as V (with a different merge target) or as H. When a grid point traveling in the vertical direction changes direction or changes its merge target, we must update its old merge target with the information that the vertical stream will no longer merge to it. Therefore, we record the number of streams that are currently merging to grid points with directions set to H. Once the number of streams currently merging to a grid point with direction set to H goes to 0, the grid point's direction is set to N.

5.3.2 Rectilinear Algorithm Example

We show how the rectilinear algorithm works with an example. Consider the following arrivals to the system: $a = (11, 0)$, $b = (11, 3)$, $c = (11, 4)$, $d = (11, 9)$, $e = (12, 3)$, and

$f = (12, 6)$. We show how the algorithm proceeds with a series of figures. In each figure, the grid point directions are represented by shapes (circles for N, squares for H, and triangles for V). Also, when a grid point is assigned as vertical, we keep track of where the vertical and horizontal stream would be in the grid, just in case we update its direction before it reaches the merge location. Dashed lines show the trails of vertical grid points in both their horizontal and vertical directions. Streams that are permanent are shown as solid lines.

In the example shown through the sequence of Figures 5.4 through 5.11, we see that the rectilinear algorithm in fact allows streams to merge with streams not yet in the system. The stream started at grid point $(11, 3)$ eventually merges to the stream started at $(12, 6)$. The dyadic algorithm for the time-shifting case does not schedule any stream started at time 11 to merge with a stream started at time 12.

The stream merge schedule for a single client can be gleaned from the figure. Let us look at client $(11, 3)$ in Figure 5.12. When client $(11, 3)$ arrives to the system, the server distributes segment 3 to the client via its own stream. Meanwhile, $(11, 3)$ also listens to the stream being sent to $(11, 4)$, so it gets segment 4 from that stream. At time 12, $(11, 3)$ merges to the stream for $(11, 4)$. After the merger, client $(11, 3)$ now gets segment 5 via the vertical stream and receives segment 6 from the stream started for client $(12, 6)$. At time 13, the streams merge and client $(11, 3)$ starts listening to the root stream. It gets segments 7 through 12 from the vertical stream and segments 13 through 18 from the root stream. At time 19, client $(11, 3)$ merges to the root stream and continues to receive segments from the root stream.

5.3.3 Variations

Committed Vertical Grid Points

We improve and vary the proposed rectilinear algorithm by extending the algorithm in two dimensions. First, in the algorithm description, vertical grid points whose stream lengths are at least as long as the length to the merger are moved to the next time unit without being part of the greedy selection process. This is to ensure that the grid points make progress toward the root stream and settle on a merge target. This tradeoff of when to

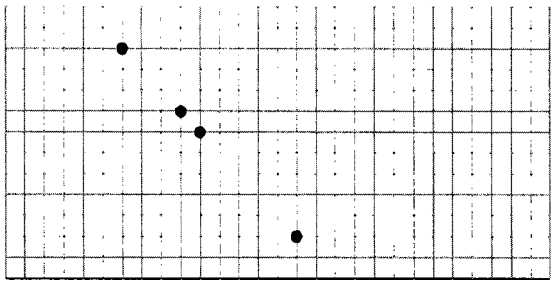


Figure 5.4: The arrivals $a = (11, 0)$, $b = (11, 3)$, $c = (11, 4)$, and $d = (11, 9)$ are shown on the rectilinear grid. Each arrival does not a direction specified, so we calculate the distances between the pairs. The distances between pairs are as follows: 6, 2, 10, 2. Notice that the last distance is 2, since d is only two units away from the root stream, which is represented by the solid line. We have a tie for the minimum distance and we choose the grid point listed last, so d is chosen for scheduling first.

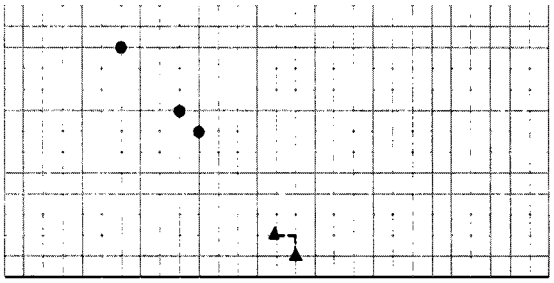


Figure 5.5: The stream represented by $d = (11, 9)$ is set to vertical. We keep track of where the stream would be on the grid as a vertical and horizontal stream, since it could be set to a different direction later in time. The triangle represents that the stream is vertical.

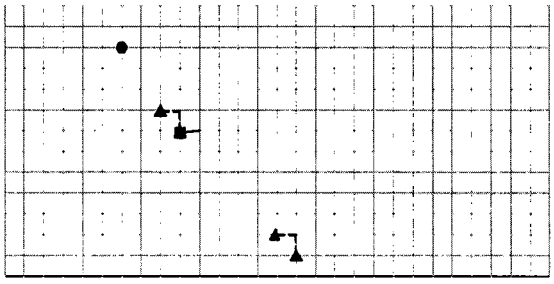


Figure 5.6: We have $a = (11, 0)$, $b = (11, 3)$, and $c = (11, 4)$ to schedule. The distances (in order) is as follows: 6, 2, 7. Thus, b and c are chosen to merge and b is set to vertical and c is set to horizontal. Their merge location is at $(12, 5)$. The square indicates a horizontal stream and the triangles indicate vertical streams.

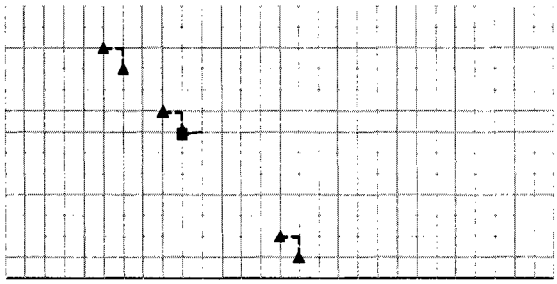


Figure 5.7: At this point, all but a have been scheduled at time 11. Thus, a is the only stream left on the list and its direction is set to vertical to merge to the root stream. At this point, all streams have been updated for time 12.

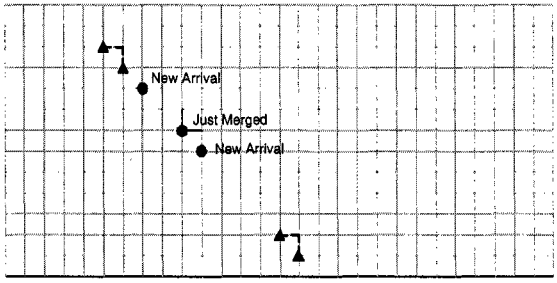


Figure 5.8: Two new arrivals $e = (12, 3)$ and $f = (12, 6)$ arrive to the system. Also, there is a merger of two streams at $(12, 5)$, so the direction is set to not yet assigned.

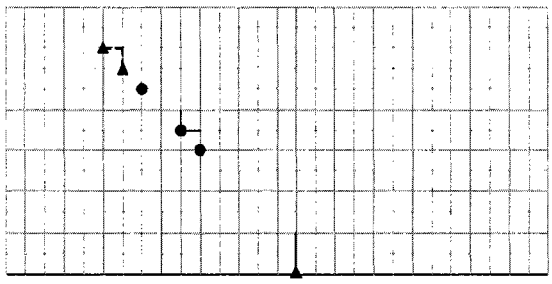


Figure 5.9: The distances between grid points is as follows: 4, 4, 2, 8, and 1. Thus, the stream closest to the root stream is chosen first to schedule. It simply continues in the vertical direction.

establish a permanent merge target for a vertical grid point is a parameter that we can tune. We ran experiments to determine the optimal parameter for making vertical grid points permanent.

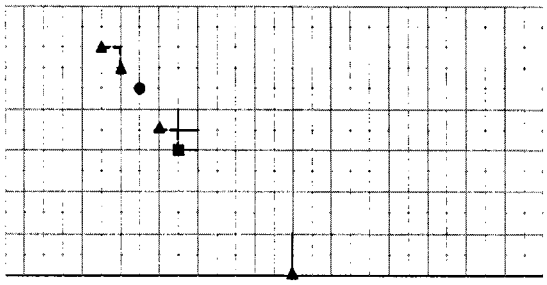


Figure 5.10: The following grid points need to be scheduled: $(12,1)$, $(12,3)$, $(12,5)$, and $(12,6)$. The distances are 4, 4, 2, and 6. Thus, $(12,5)$ and $(12,6)$ are chosen to merge and their directions and merge locations are updated accordingly.

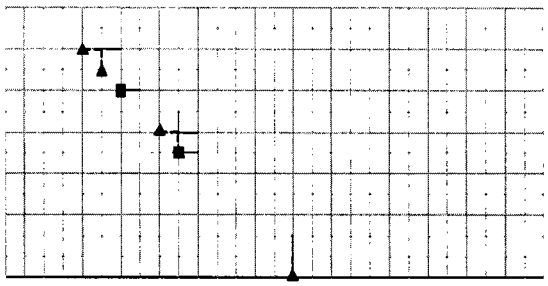


Figure 5.11: The remaining grid points are $(12,1)$ and $(12,3)$ that need to be scheduled. The distances are 4 and 9. Thus, the two grid points are scheduled to merge and their directions and merge locations are updated accordingly.

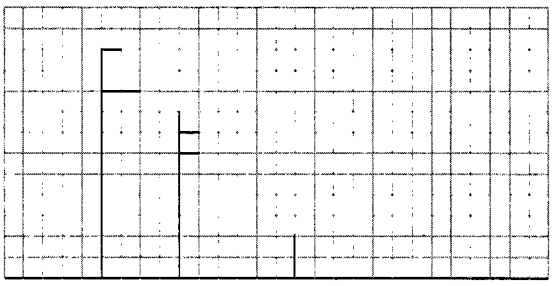


Figure 5.12: This figure shows the resulting set of streams generated by the rectilinear algorithm for the six arrivals. Notice that the rectilinear algorithm allows streams starting at time 11 to eventually merge to streams starting at time 12.

We adjust the factor which we call the vertical grid point committed factor. This factor indicates at which point a vertical grid point will continue toward its merge target without a direction reassignment in the rectilinear algorithm. We dictate that a vertical pair is

committed if the length of the stream invested is greater than or equal to the committed factor multiplied by the distance to its merge point. The distance invested DI can be calculated based on the grid point G as follows:

$$DI(G) = 2L_t(G) - L_s(G) - 2ML_t(G) + ML_s(G) \tag{5.1}$$

We can also calculate how much longer the vertical stream needs to be in order to merge to its merge target. We call this the distance to target DT. We keep track of the merge location for grid point G, so this calculation is straightforward:

$$DT(G) = ML_t(G) - L_t(G) \tag{5.2}$$

Now that we have the two calculations, we can modify the rectilinear algorithm in the beginning of phase two. Instead of updating vertical grid points where the stream already distributed (DI) is as least as long as the stream extension (DT) to the merge target, we can use a committed factor f other than one. We can restate part one of phase two as:

1. For each grid point G in L where $D(G)$ is V and $DI(G) \geq f * DT(G)$, set $L_t(G)$ to $L_t(G) + 1$ and $L_s(G)$ to $L_s(G) + 1$.

Figure 5.13 shows the total bandwidth for running the rectilinear algorithm with committed factors ranging from 0 to 1. A committed factor of 0 means that a vertical grid point can always be reset; in other words, the vertical grid point never becomes committed. A committed factor of 1 is the result of running the rectilinear algorithm as proposed originally. In the experiment, the mean interarrival rate was .1 seconds (approximately 10 arrivals per second) with arrivals following a Poisson process over one-hour period. The figure shows that the best committed factor is .6 for this sequence of arrivals.

The results are similar for different interarrival rates. Figure 5.14 shows the total bandwidth results for interarrival rates of 1 second and 1.9 seconds. The trend is similar to the interarrival rate of .1, so we will use .6 as the committed factor throughout the rest of this chapter.

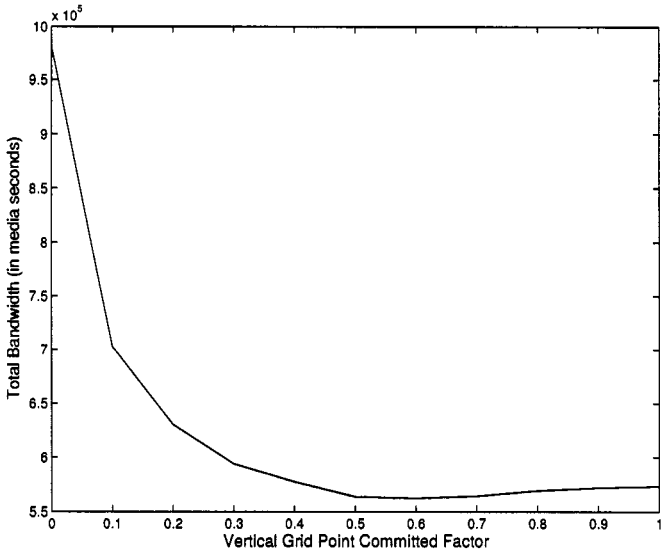


Figure 5.13: This figure shows committed factors from 0 to 1 for the rectilinear algorithm. The mean interarrival rate was .1 and the length of the experiment was 1 hour. The figure shows the committed factor producing the least total bandwidth is .6.

Re-using Horizontal Grid Points

The second way in which we can improve the algorithm is by keeping grid points with directions set to H on the list L_t for use in making greedy merge target decisions. A horizontal stream on the rectilinear grid can support several vertical streams, so instead of moving the horizontal grid point to L, we leave it on L_t for other grid points to use. We refer to this variation as the *keep horizontals rectilinear algorithm.*

We ran the rectilinear algorithm with a vertical committed factor of .6 for both the original rectilinear algorithm and the modification where we keep the horizontal pairs on the list after greedily selecting them for merging. The total bandwidth and average bandwidth results for the comparison are shown in Figure 5.15 and Figure 5.16, respectively. Each figure shows the bandwidth results for varying interarrival rates from .1 to 1.9. The variation of re-using horizontal grid points produces similar results to the original version. We will continue to use the modification of re-using horizontal grid point throughout the remainder of the chapter, since there is a slight benefit for the maximum bandwidth under high loads

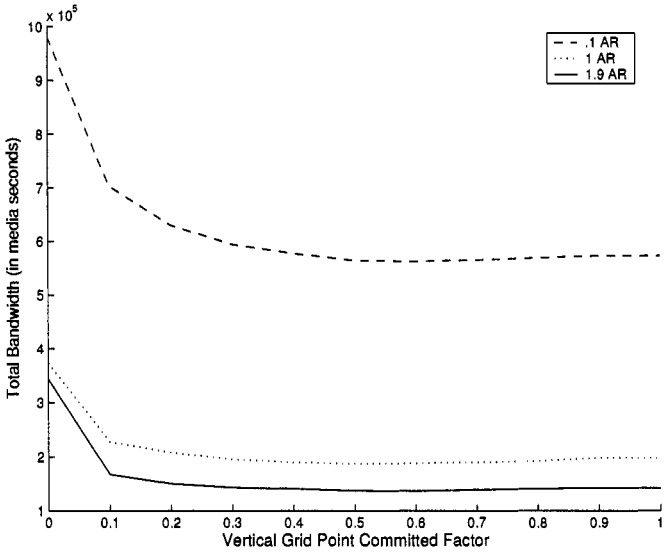


Figure 5.14: This figure shows committed factors from 0 to 1 for the rectilinear algorithm. The mean interarrival rates were .1, 1, and 1.9, and the length of the experiment was 1 hour. The figure shows the committed factor producing the least total bandwidth is close to .6 for all three interarrival rates.

(small interarrival rates).

In the modification of the rectilinear algorithm, we keep the grid points assigned to the horizontal direction on L_T in step two of phase two. Instead of incrementing the time and segment of the horizontal pair and placing it on L, we do not increment the time and segment and keep it on L_T. At the end of the greedy selection process for time t, the horizontal pairs assigned during the greedy selection will be updated and put on L.

5.3.4 Complexity

At each time t, let us consider the complexity of the rectilinear algorithm in terms of the number of concurrent streams and new clients in the system. We denote the number of concurrent streams and new clients as C. In the implementation, we can keep all the clients and streams in sorted order. Because each update to the next time interval simply increases the location pair values by 1, the grid points remain in sorted order for the next time interval. Thus, we can assume that the existing streams at time t are in sorted order. We can place

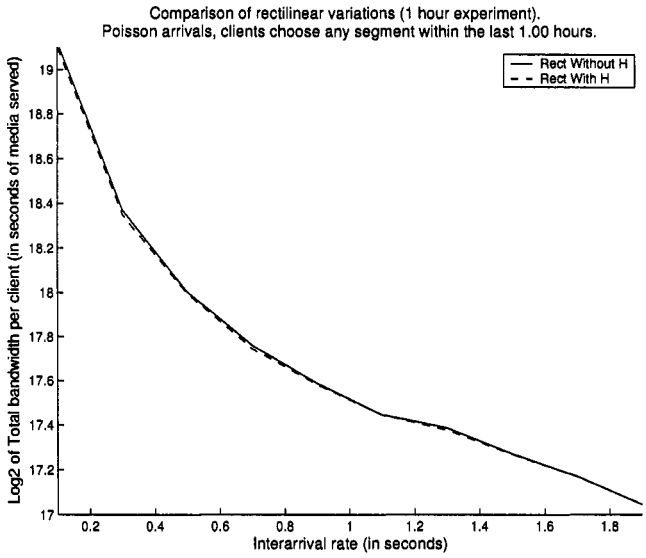


Figure 5.15: Shows the total bandwidth for the original rectilinear algorithm and the modification where horizontal grid points are kept on the list for other greedy selections.

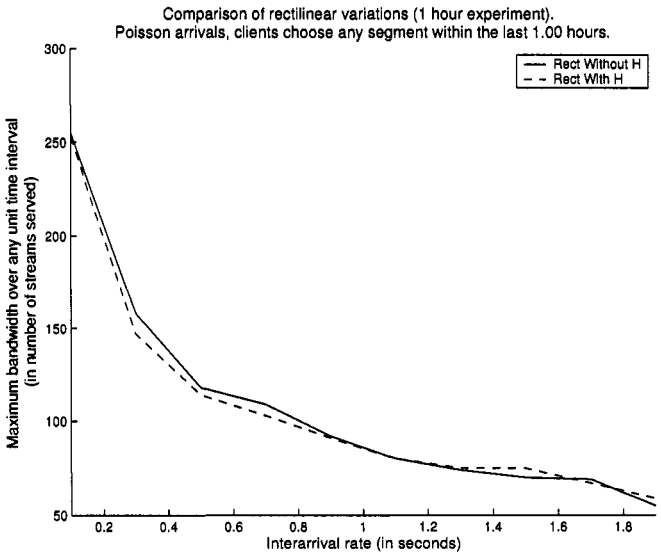


Figure 5.16: Shows the maximum bandwidth for the original rectilinear algorithm and the modification where horizontal grid points are kept on the list for other greedy selections.

each new arrival into the list of sorted grid points in $O(C)$ time by doing a linear search to find the insertion point. (If L is an indexed list, then each insertion requires moving $O(C)$ elements to the right. If L is a linked list, then finding the insertion point takes $O(C)$ time and the insertion is constant time.) Assuming the number of new arrivals is N and the number of existing streams is $C - N$, the cost for placing all N arrivals in L is $O(NC)$. Finding the distance between successive streams takes time $O(C)$ time, since there are C streams and new clients in the list for time t. Each merger (selection of closest pair) is a constant operation, since we just need to update the information about the stream. Thus, at each time interval t the complexity of the rectilinear algorithm is $O(NC + C) = O(NC)$. If N is much smaller than C, then the algorithm runs in linear time proportional to the number of concurrent streams at each time point.

The original dyadic algorithm is linear time with respect to the total number of arrivals, since there is at most one new arrival at each time interval and finding the stream to which the client should merge requires popping intervals from the stack. Each of these intervals gets pushed on the stack once and gets popped at most once. Each interval gets created once due to a client arrival, so the maximum number of push/pop operations is $2N$ where N is the number of client arrivals. The dyadic variation for time-shifting is no longer linear time with respect to the number of arrivals, since the intervals do not necessarily get removed each time they are scanned. If there are I existing intervals in the dyadic list at time t and there are N new arrivals at time t, then finding the merge target for each arrival could take up to $O(N + I)$ operations, since the dyadic list grows by one with a new arrival. The total running time for the dyadic algorithm at each time point is $O(N(N + I))$.

5.3.5 Proof of Correctness

Here we prove that the rectilinear algorithm (without the variations) does indeed produce feasible merge schedules for clients. By feasible, we mean that the client receives each media segment in time to playback the media segment and all media segments necessary to "catch up" to the root stream are delivered to the clients. We use the lemmas introduced in Chapter 3 to show that the rectilinear algorithm produces feasible schedules for a finite

set of client arrivals.

Lemma 5.3.1 *For a finite set of client arrivals, the rectilinear algorithm creates paths (as followed in the direction of increasing time) consisting of downward vertical segments and leftward horizontal segments.*

Proof: Let us assume there is a directed path p (in the direction of increasing time) that has an upward vertical segment. Then, at some point the grid point G_v along the path is set as vertical with a target that is above the grid point in the grid. Since the list of grid points is sorted by increasing segment, a vertical grid point can only be assigned to merge to a grid point that has a larger segment value (which is lower and to the right on the grid). Thus, all vertical segments must be directed downward in a path. Now let us assume that the path has a segment directed to the right. At some point, the grid point G_h will have to be set as horizontal along this path. Thus, the vertical grid point that eventually merges to G_h must be to the right of G_h. This is impossible, since the list of grid points is always in order by increasing segment and vertical grid points merge to grid points later in the list. In both cases, we have a contradiction. Therefore, the rectilinear algorithm creates paths (in the direction of increasing time) that consist of downward and leftward segments. ∎

Lemma 5.3.2 *For a finite set of client arrivals, the rectilinear algorithm creates a path between every client arrival (as placed on the rectilinear grid) and the root stream.*

Proof: Assume there is a client arrival c that is not connected via a path created by the rectilinear algorithm to the root stream. First, we know that all paths consist of downward and leftward segments. Let us assume that the path that goes through c on the rectilinear grid terminates before reaching the root stream. Then at some point during the execution of the algorithm, a grid point is removed that represents this termination point. Grid point are only removed when they lie on the root stream (a merger to the root stream) and when a vertical grid point merges to its horizontal merge target (a merger at the merge location). Since we are assuming the path terminates prior to the root stream, it must be the case that the vertical grid point merged to the horizontal grid point and discontinued. But, according

to the algorithm, the horizontal grid point remains in the system which would continue the path. Therefore, a path cannot terminate before reaching the root stream.

Now let us assume the algorithm creates a path from a client c that continues indefinitely. Because the paths are made up of leftward and downward segments, the path must have an infinite number of leftward segments and fewer downward segments than the vertical distance from c to the root stream. To have an infinite number of leftward segments, there must be a corresponding infinite number of grid points assigned to the horizontal direction. For a grid point to be assigned as horizontal, there must be at least one other grid point to its left assigned as vertical. At some point, the horizontal grid point will merge to its leftmost vertical grid point and the new grid point will be assigned the direction N. Since we are assuming a finite set of client arrivals, at some point the last grid point assigned horizontally will meet its leftmost merge target and become assigned to the N direction. The algorithm will then assign the direction to V to merge to the root stream. Thus, the rectilinear algorithm creates a path from each client arrival to the root stream. ∎

Lemma 5.3.3 *The rectilinear algorithm creates schedules that satisfy the elbow property. That is, at all junctions, the segments coming from the right must be at least as long as the segments coming in from above.*

Proof: We prove this lemma by contradiction. Assume the rectilinear algorithm creates a junction j (merge location) where the segment coming in from the right is shorter than the segment coming in from above. At some point during the execution of the algorithm, a grid point G_v was assigned to merge to grid point G_h at location j. For the vertical segment to be longer than the horizontal segment, G_v must be assigned to merge at j at time t while G_h must be assigned after time t. This is impossible, since the algorithm does all merge assignments for pairs at the same time. ∎

Lemma 5.3.4 *Each junction or client arrival has exactly one line going down or to the left.*

Proof: Again, we prove this by contraction. Assume there is a junction j that has two lines leaving it, one to the left and one going downward. At the junction j, the rectilinear

algorithm will either assign the grid point to N as the direction or keep the direction as H if there are other grid points eventually merging to it. If the algorithm assigns the direction to N, the algorithm will assign the direction to V or H by the end of the time unit, so there is exactly one line leaving the junction. If the algorithm keeps the grid point's direction as H, then there is a single horizontal line leaving the junction. In all cases, the rectilinear algorithm creates paths such that each junction has exactly one line going down or to the left. ∎

Theorem 5.3.5 *The rectilinear algorithm produces feasible merge schedules that are in the form of tree.*

Proof: From the preceding lemmas, it is clear that merge schedules produced by the rectilinear algorithm are trees, with leaves as client arrivals embedded on the rectilinear grid. Every client arrival eventually merges to the root stream. When the path is vertical, a client is listening to two streams simultaneously and gathers the necessary segments before merging to the horizontal grid point. Note that the algorithm can change a vertical grid point's merge target, so the client may receive multiple copies of the same data. The schedule is still feasible and the client will receive the necessary data before playback. ∎

5.3.6 Buffers and Early Client Exits

When clients arrive to the system, they must have the buffer space to store $t - f$ segments, where t is their arrival time and f is their requested first segment, in order to eventually merge to the live stream. If the client has little to no buffer space, then the stream sent to the client is always horizontal in the rectilinear grid since the client cannot buffer segments from a second stream. If a client has a buffer of size b segments, then it can merge to a stream at a grid point no further than $2b$ away in rectilinear distance. If a client with buffer space b does have a merge target that is $2b$ away in distance, then at the time of the merger, the client cannot handle any more segments in the buffer so it simply plays out the segments in its buffer. When it has emptied its buffer, it can then re-enter the system as client $(t + b, f + b)$. The rectilinear algorithm assumes clients have sufficient buffer space

to eventually merge to the live stream. Adapting the rectilinear algorithm for the case of clients with buffer limits is left for future work.

Clients can exit the system early. This is reasonable in the time-shifting case, since the client "tunes in" for a set amount of media time before leaving the system. The rectilinear algorithm supports this operation easily since we simply remove a stream if all the clients receiving that stream have exited the system. This feature of allowing client departures was not implemented, but should be a straightforward modification to remove clients from the system.

5.4 Comparison Between Dyadic and Rectilinear

Let us compare the dyadic and rectilinear algorithms on example sets of client arrivals. The dyadic algorithm produces the schedule discussed in Section 5.2.1 and shown in Figure 5.2 for the client arrivals $a = (4, 0)$, $b = (6, 0)$, $c = (8, 2)$, $d = (10, 4)$, and $e = (12, 6)$. The total number of segments distributed (other than the live stream) is 22. The rectilinear algorithm produces the schedule as shown in Figure 5.17, which distributes a total of 16 segments. In fact, we can extend the example, such that for each new arrival in the form of $(2x, 2x - 6), x \in \{7, 8, 9 \ldots\}$, the dyadic algorithm increases the total cost by 4 while the rectilinear algorithm increases the total cost by 2.

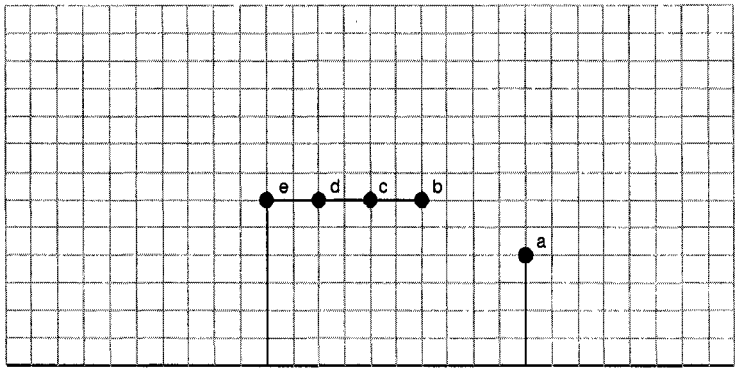


Figure 5.17: Shows the schedule produced by the rectilinear algorithm for client arrivals $a = (4, 0)$, $b = (6, 0)$, $c = (8, 2)$, $d = (10, 4)$, and $e = (12, 6)$. The total bandwidth, excluding the live stream, is 16.

Next, consider how the dyadic and rectilinear algorithms schedule the arrivals $a = (10, 0)$, $b = (11, 3)$, $c = (12, 6)$. Figure 5.3 shows how the dyadic algorithm schedules these client arrivals. (Ignore the stream for client d.) The schedules is equivalent to the schedule produced by the merge once algorithm and distributes 24 segments for these three arrivals. The rectilinear algorithm, because it can reassign merge targets, produces the schedule shown in Figure 5.18. The rectilinear algorithm creates a schedule that uses 16 segments of media transmission other than the live stream.

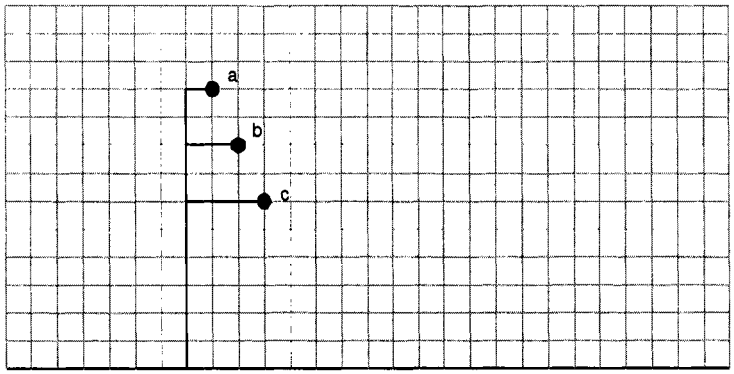


Figure 5.18: Shows the schedule produced by the rectilinear algorithm for client arrivals $a = (10, 0)$, $b = (11, 3)$, and $c = (12, 6)$. The total bandwidth, excluding the live stream, is 16.

In fact, we can extend the example with three arrivals to create a set of arrivals such that the dyadic algorithm produces a schedule that is any factor larger than the schedule the rectilinear algorithm creates. With three arrivals, we can create a situation where the dyadic schedule is about three times the total bandwidth cost of the rectilinear schedule. Take the arrivals a, b, and c in Figure 5.18 and move them vertically by 100 units. Then, we have the arrivals $(110, 0)$, $(111, 3)$, and $(112, 6)$ and the dyadic algorithm schedules each to merge to the root stream for a total cost of 324. The rectilinear algorithm creates the schedule shown in Figure 5.18, but the single vertical stream extends 100 more units down to the live stream. The rectilinear algorithm produces a schedule with a total cost of 126. The factor increase for the dyadic algorithm is about three. To create a factor increase close to N, just stack N arrivals such that the slope of the line connecting the arrivals is

−2. The rectilinear algorithm always merges the arrivals before creating a single stream to the live stream, while the dyadic algorithm dictates that each arrival merge directly to the live stream. The dyadic algorithm produces N long vertical lines, while the rectilinear algorithm creates a connected cluster of the arrivals and one long vertical line.

The dyadic and rectilinear algorithms differ in how they assign merge targets. The dyadic algorithm assigns merge targets based on intervals created when clients arrive to the system. When a client arrives to the system, the dyadic algorithm completely specifies its sequence of merge targets. On the other hand, the rectilinear algorithm only specifies its most immediate merge target, and this merge target could change before the merge time to the original target. The dyadic algorithm creates static schedules while the rectilinear algorithm creates dynamic schedules. Because the rectilinear algorithm can reassign stream directions, a client can receive multiple copies of the same movie segment. In Figure 5.18, client a receives segment 10 twice. When a arrives to the system, it is assigned to merge to the root stream, so it starts receiving segment 10 from the live stream broadcast while getting segment 0 from its dedicated stream. When b arrives to the system, client a instead starts listening to b's stream for segments 3 and 4. Eventually, client a will receive segment 10 again after merging to client c's stream. The client can either keep segment 10 in the buffer or discard segment 10 when a starts listening to b's stream.

5.4.1 Experimental Comparison

Simulation results show that the rectilinear algorithm saves server bandwidth over the dyadic algorithm and the merge once solution. The dyadic and rectilinear algorithms save a substantial amount over merge once in terms of server bandwidth. The first simulation uses a Poisson process to generate client request times and chooses uniformly between 0 and the request times to generate requested segments. The simulation lasted for two hours with the client interarrival rate varying from .05 seconds (about 20 arrivals per second) to 2 seconds. We calculated both the total bandwidth used and the maximum bandwidth used over any time period during the simulation. Figures 5.19 and 5.20 show the total and maximum bandwidth, respectively.

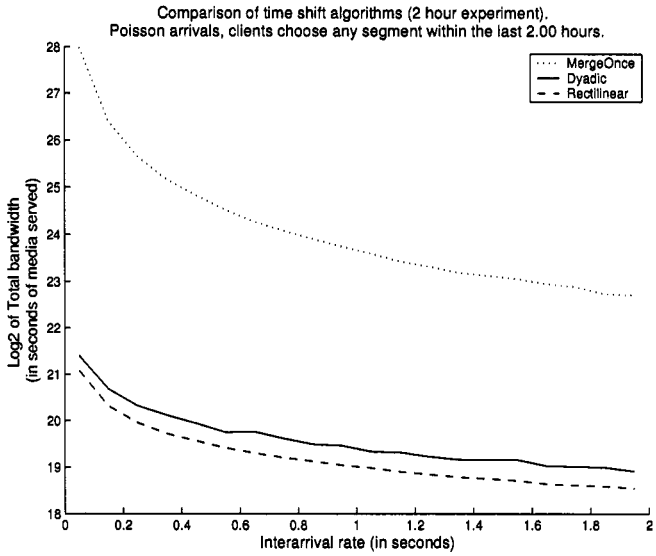


Figure 5.19: Simulation results for the merge once, dyadic, and rectilinear algorithms. The total bandwidth is measured in the number of media segments distributed to satisfy all client requests. Here we show the total bandwidth results for client interarrival rates from .05 seconds to 2 seconds. The rectilinear algorithm performs the best in terms of total server bandwidth.

We also simulated the situation where the clients have limited rewind capabilities, meaning that the first segment requested must be within a certain range of the currently distributed segment by the live stream. We ran a simulation for a duration of two hours and a maximum rewind capability of one hour. Again, clients arrived according to a Poisson process and first segments were randomly chosen between the current time and the current time minus one hour. The results for total and maximum bandwidth are in Figures 5.21 and 5.22. The results are similar to those presented earlier where clients can choose any segment during the experiment. Notice that the total bandwidth and maximum bandwidth costs are similar to the case where clients can choose any segment within the last two hours, as in Figures 5.19 and 5.20. Letting clients choose segments further back in time does not dramatically increase the bandwidth used by the server. It does require that the server store more data, since the history window is longer – 2 hours instead of 1 hour.

We also ran an experiment where the first segment requested always falls on the start

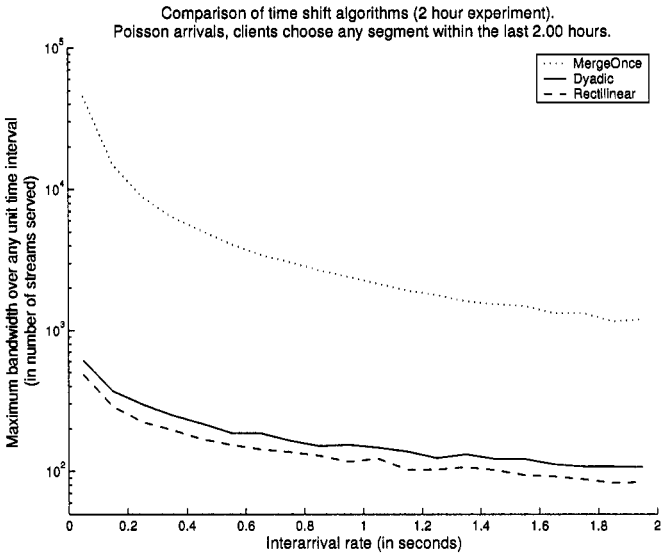


Figure 5.20: Simulation results for the merge once, dyadic, and rectilinear algorithms. The maximum bandwidth is the largest number of concurrent streams to satisfy clients in the system over all times in the simulation. Here we show the maximum bandwidth results for client interarrival rates from .05 seconds to 2 seconds. The rectilinear algorithm performs the best in terms of maximum server bandwidth.

of a 15-minute interval. The experiment lasted 2 hours and client could request as first intervals at the following minute markers: 0, 15, 30, 45, 60, 75, 90, and 105. In this case, the rectilinear and dyadic algorithms performed almost identically in terms of total server bandwidth and maximum bandwidth.

5.5 Rectilinear Algorithm for Media-on-Demand Without Time-Shifting

Recall that the standard media-on-demand model distributes media with a fixed length L, which is a measure of the length in the number of segments. We adapt the rectilinear algorithm presented in this chapter to work for the standard case by creating new rectilinear grids for full-length streams. Notice that all rectilinear pairs have 0 as the requested first segment in the standard model, since clients always receive the media object from beginning to end. We use the time-shifting rectilinear algorithm as a subroutine for the rectilinear algorithm for the standard model. We refer to the version for the standard model as

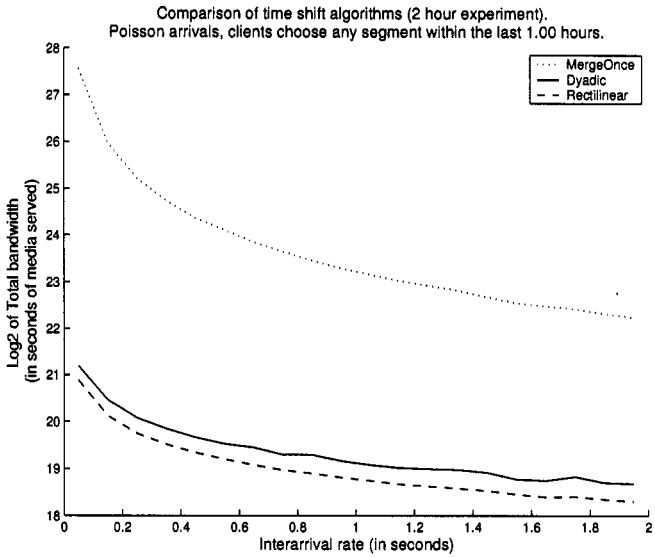


Figure 5.21: Simulation results for the merge once, dyadic, and rectilinear algorithms. The total bandwidth is measured in the number of media segments distributed to satisfy all client requests. Here we show the total bandwidth results for client interarrival rates from .05 seconds to 2 seconds. Clients can choose any segment within the last hour. The rectilinear algorithm performs the best in terms of total server bandwidth.

rectilinear standard.

We start a new rectilinear grid when a new client arrives at a time more than $L/2$ segments after the most recent root stream (full-length media file delivered). Because two full-length streams can co-exist in the system, there may be two grids in operation simultaneously.

Rectilinear Standard Algorithm

Let r_t be the start time of the most recently distributed full-length root stream. At time t, perform the following actions in order:

1. If an existing rectilinear grid has no remaining grid points in its list, then remove the associated rectilinear (for time-shifting) process.

2. If there is a new arrival at time t and $t \leq L/2 + r_t$, then create rectilinear pair $(t - r_t, 0)$ and insert it into the list of pairs for the rectilinear grid corresponding to the root at

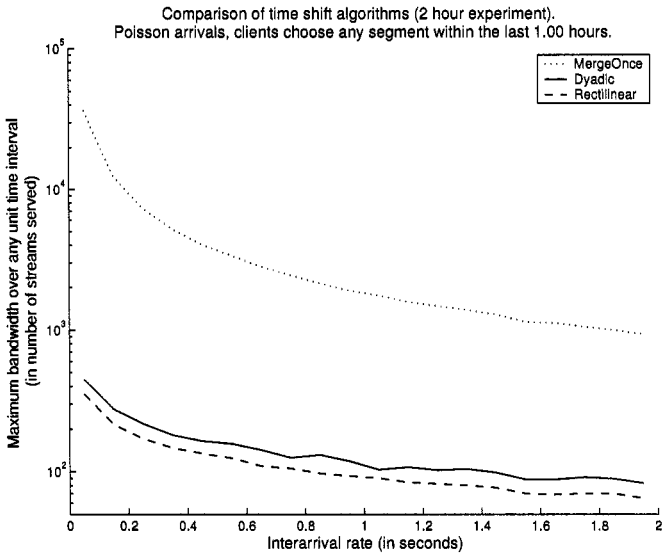


Figure 5.22: Simulation results for the merge once, dyadic, and rectilinear algorithms. The maximum bandwidth is the largest number of concurrent streams to satisfy clients in the system over all times in the simulation. Here we show the maximum bandwidth results for client interarrival rates from .05 seconds to 2 seconds. Clients can choose any segment within the last hour. The rectilinear algorithm performs the best in terms of maximum server bandwidth.

r_t.

3. If there is a new arrival at time t and $t > L/2 + r_t$, then set r_t to t and initialize a new grid and empty list for the rectilinear algorithm. Note: we do not put $(0, 0)$ on the list, since the rectilinear algorithm for time-shifting has an implied root stream to which all pairs on the list may merge.

4. Run the rectilinear algorithm for the concurrent rectilinear grids and set t to $t + 1$. (Note: There may be two concurrent rectilinear grids in progress.)

5.5.1 Rectilinear Standard Example

We show how the rectilinear algorithm for the standard model executes with an example. Assume the media length L is 12 with arrivals at times 0, 2, 4, 5, 7, 8, 10, 13, and 14. At

time 0, a new root stream is activated, represented by the horizontal line in Figure 5.23. The arrival at time 2 is scheduled to merge to the stream started at 0. The arrival at time 4 is scheduled to proceed vertically upon arrival, but at time 5 a new arrival is scheduled to merge to the arrival at time 4. Since there is no arrival at time 6, the arrivals 0, 2, 4, and 5 make up a single rectilinear grid and the algorithm proceeds until the root stream terminates at time 12. The arrival at time 7 initiates a rectilinear grid and root stream. At times 8 and 10, the arrivals are scheduled to proceed vertically to the root stream started at 7. At time 13, the arrival is scheduled vertically but is reset to horizontal for the arrival at time 14. The rectilinear algorithm proceeds for the grid consisting of arrivals 7, 8, 10, 13, and 14. The dashed line in Figure 5.23 shows the concurrent streams at time 8 in the system and highlights the fact that two rectilinear grids can be processed simultaneously.

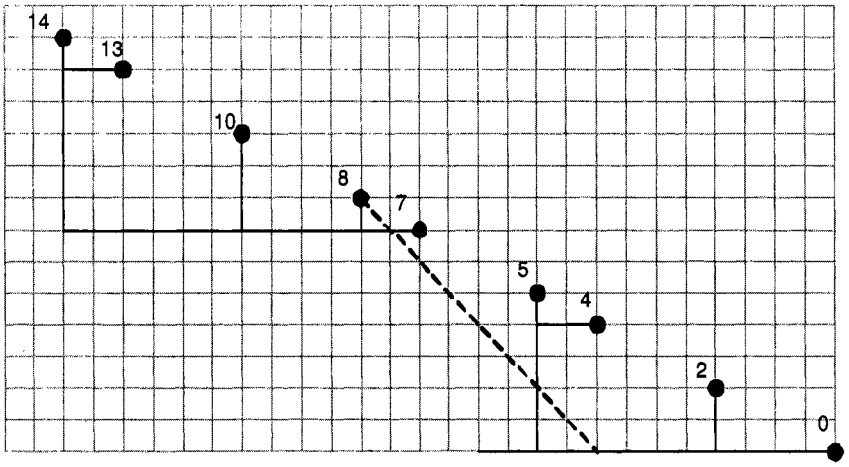


Figure 5.23: The resulting set of merge trees for the rectilinear algorithm adapted for the standard media-on-demand model with media length 12 for arrivals at times 0, 2, 4, 5, 7, 8, 10, 13, and 14. The dashed line shows the active streams in the system at time 8.

5.5.2 Proof of Correctness

It was proved earlier that the rectilinear algorithm for time-shifting creates feasible merge schedules in the time-shifting case. The only portion that needs to be proved for the standard model is that clients assigned to a root stream will eventually merge to the root

stream before or at the time the root stream terminates. Without loss of generality, assume the root stream starts at time 0 and the media has length L. From the definition of the algorithm, the last possible arrival that could be assigned to the root starting at 0 is at time $L/2$. On the rectilinear grid, the point $(L/2, 0)$ is L units to the left of 0 and $L/2$ units above 0. When processing the lists of grid points associated with the rectilinear algorithm for time-shifting, the point $(L/2, 0)$ can be assigned to merge to any grid point below it. Thus, the direction assigned to $(L/2, 0)$ will be vertical upon its introduction to the system. No other future grid points with times greater than $L/2$ will be assigned to the grid starting with a root at 0, so as time progresses, the grid point $(L/2, 0)$ can only be updated in terms of its merge target and not its direction. Its direction is vertical until it reaches the root stream at time L. Thus, the very last arrival assigned to merge to the root started at 0 will always be vertical in direction.

Consider an arrival a at time t such that a is not the final arrival assigned to merge to root 0. During the course of the rectilinear algorithm, the grid point associated with a's arrival may change directions. If it is set to the horizontal direction, we know that the very last arrival to merge to the root always proceeds vertically so the grid point cannot be left of the vertical line from the last arrival to merge to the root. Thus, the arrival a merges to the root stream at or before time L. Therefore, every arrival eventually merges to the root stream to which it was assigned in the rectilinear standard algorithm.

5.5.3 Experimental Comparison

We empirically compared the rectilinear standard algorithm to algorithms known to achieve total bandwidth usage that is close to optimal. We compared the rectilinear algorithm to the 2-Dyadic algorithm (where $\alpha = 2$) [21], the $\phi-$Dyadic algorithm (where α is the golden ratio) and the Earliest Reachable Merge Target algorithm [30]. The dyadic algorithm is familiar from earlier chapters of this dissertation. We describe the ERMT algorithm briefly here.

The ERMT algorithm is an event-driven algorithm that operates on sets of clients. A set could consist of a single client (new arrival to the system) or a group of recently merged

clients. A set is scheduled to merge to the closest stream to which the set can merge. A set can only merge to a stream if the stream will still be active at the time of the merger. Therefore, the algorithm relies on knowing the eventual merge times of client sets in the system. When streams merge in the system, the union of the sets of clients is scheduled to merge to another stream, if a "catchable" one exists.

The experimental setup for the comparison uses arrival rates as the variable parameter. Figure 5.24 shows the total bandwidth (in terms of the factor increase over optimal) for media with length 1000 and an experiment 50 times the media length. The results in Figure 5.24 were generated using constant rate arrivals, meaning the arrivals were uniformly spaced over time. The x-axis shows a variety of arrival rates, where the arrival rate is measured in the number of arrivals per media length. As one can see, the 2-Dyadic performed the worst over all arrival rates shown while the $\phi-$Dyadic almost always performed the best. The ERMT and rectilinear algorithms produced similar factor increases which were slightly worse than the $\phi-$Dyadic.

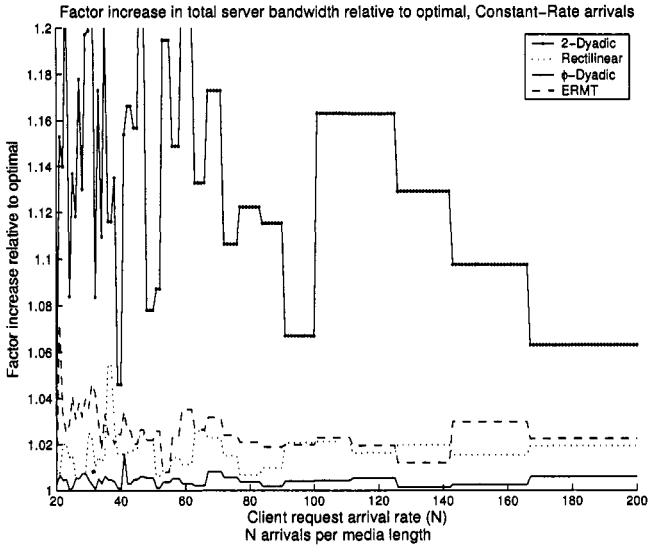


Figure 5.24: Simulation results for the 2-Dyadic, Rectilinear, ϕ-Dyadic, and ERMT algorithms for media of length 1000 and constant rate arrivals. The arrival rate is measured in the number of arrivals per media file length, so the arrival rate of 100 is 100 arrivals per 1000 media segments. The graph shows the factor increase over the optimal offline algorithm.

Figure 5.25 shows the factor increase relative to optimal for the four algorithms with client arrivals generated by a Poisson process. The mean arrival rate is shown on the x-axis as the mean number of arrivals generated per media length. Under the Poisson process, the algorithms produce more clearly delineated results. The ERMT performs the best while the rectilinear algorithm is second best for low client request intensities and continues an upward trend as client arrivals become more dense. This is most likely due to the fact that the rectilinear algorithm makes new greedy decisions at each time t during the simulation. When the number of concurrent grid points in the system becomes large, decisions made during the previous time tick are more likely to be "undone" at the current time tick, so decisions about merge targets force more updates.

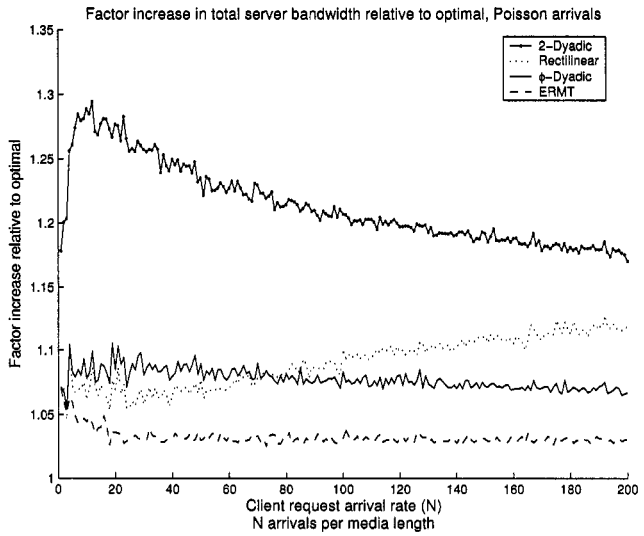


Figure 5.25: Simulation results for the 2-Dyadic, Rectilinear, ϕ-Dyadic, and ERMT algorithms for media of length 1000 and Poisson arrivals. The arrival rate is measured in the number of arrivals per media file length, so the arrival rate of 100 is 100 arrivals per 1000 media segments. The graph shows the factor increase over the optimal offline algorithm.

Figures 5.26 and 5.27 show results for experiments with similar setups, except they used media with length 100. The trend in the results is similar to the previous experiment, except that the total bandwidth results for the rectilinear algorithm do not increase as dramatically with dense arrivals.

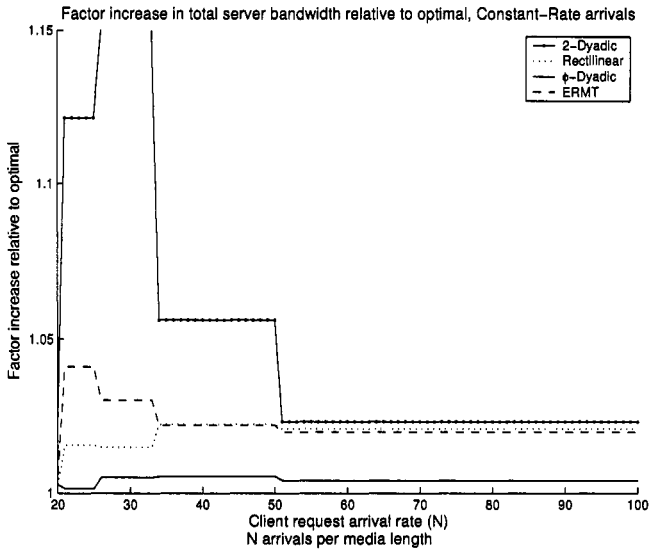


Figure 5.26: Simulation results for the 2-Dyadic, Rectilinear, ϕ-Dyadic, and ERMT algorithms for media of length 100 and constant rate arrivals. The arrival rate is measured in the number of arrivals per media file length, so the arrival rate of 100 is 100 arrivals per 100 media segments. The graph shows the factor increase over the optimal offline algorithm.

Finally, we show results in Figures 5.28 and 5.29 for media with length 10000 under the same experimental conditions. The trend in the results matches that of Figures 5.24 and 5.25.

5.5.4 Arbitrary Access

The time-shifting model supports arbitrary movement for standard media (random access to a movie position). In the random access model, clients have an arrival time and a requested first segment of the media. With media of length L, the set of possible requested first segments is $\{0, 1, 2 \ldots, L-1\}$. We assume the client receives the media from its requested first segment through the end. We modify the rectilinear grid to delineate the active stream area. Recall that in the time-shifting case there is a continuous, live broadcast; therefore, the bottom boundary on the rectilinear grid is this stream. We no longer have the continuous broadcast, so the boundary is dictated by the length of the media object. Figure 5.30 shows the boundary for the rectilinear grid in the standard model for random accesses. The dotted

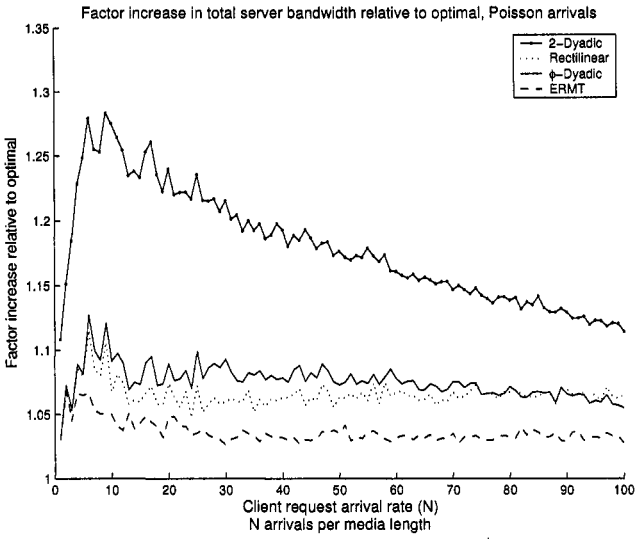


Figure 5.27: Simulation results for the 2-Dyadic, Rectilinear, ϕ-Dyadic, and ERMT algorithms for media of length 100 and Poisson arrivals. The arrival rate is measured in the number of arrivals per media file length, so the arrival rate of 100 is 100 arrivals per 100 media segments. The graph shows the factor increase over the optimal offline algorithm.

line represents the boundary for media of length 12. Notice that client arrivals can be below the horizontal line containing the grid point (0,0), represented as a square in the figure. As before, a single point in time is represented by a continuous diagonal line with slope -1. Since all streams must lie within the boundary, possible merge targets for client c must allow client c to meet its merge target within the boundary.

We can adapt the rectilinear algorithm to work with the standard model for arbitrary accesses. We make decisions for merging pairs closest together in terms of rectilinear distance, but if a pair cannot merge to any other pair in the system (because there are no other pairs or because the merger would produce streams outside the boundary), we set its direction to horizontal. Thus, the distribution of streams is determined dynamically and no full-length streams are issued unless a client requests the full media transmission.

The ability to have arbitrary access into a media file supports the VCR functions of fast forward and rewind. The literature refers to this model as interactive media-on-demand or true media-on-demand, since users can navigate to parts of the media as they wish

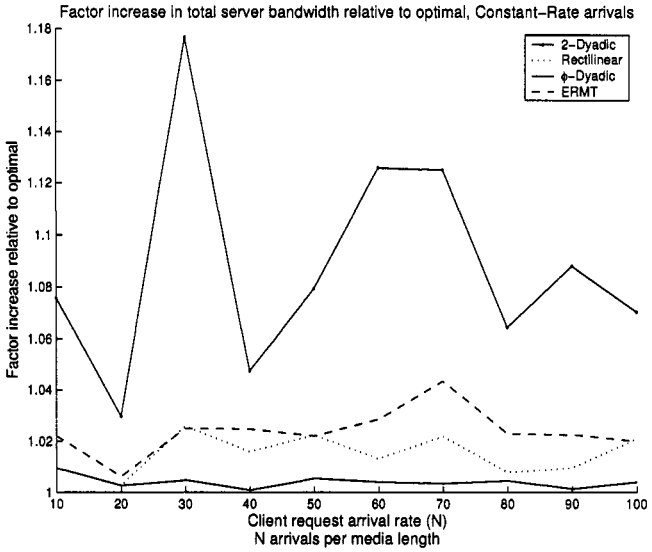


Figure 5.28: Simulation results for the 2-Dyadic, Rectilinear, ϕ-Dyadic, and ERMT algorithms for media of length 10000 and constant rate arrivals. The arrival rate is measured in the number of arrivals per media file length, so the arrival rate of 100 is 100 arrivals per 10000 media segments. The graph shows the factor increase over the optimal offline algorithm.

[53, 9, 20, 35, 70]. The existing proposals for allowing interactivity do this by allocating a separate dedicated channel to a client when it issues a command for interactivity. The rectilinear algorithm, on the other hand, continues to multicast the streams so other clients can make use of them.

Figure 5.30 illustrates the merge schedule produced by the rectilinear algorithm for the standard model with random accesses. We assume the media has length 12, which is the distance between the boundary lines above the grid location (0,0) [marked by a square in Figure 5.30]. We assume the set of client arrivals is $\{(0,5),(2,2),(2,4),(2,7),(3,0),(4,0)\}$. At time 0, the stream issued is horizontal, starting with segment 5. At time 2, the closest merger is between $(2,7)$ (new arrival) and $(2,7)$ from the horizontal stream. The next closest pair of grid points is $(2,2)$ and $(2,4)$, so they are scheduled to merge. At time 3, the arrival $(3,0)$ is assigned to merge with the horizontal stream at location $(3,5)$. At time 4, the stream started at $(2,2)$ merges to its target. The closest pair of points is $(4,0)$ (a

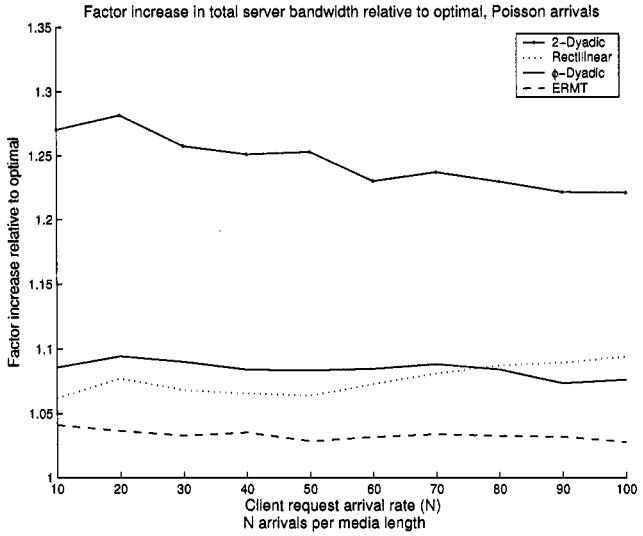


Figure 5.29: Simulation results for the 2-Dyadic, Rectilinear, ϕ-Dyadic, and ERMT algorithms for media of length 10000 and Poisson arrivals. The arrival rate is measured in the number of arrivals per media file length, so the arrival rate of 100 is 100 arrivals per 10000 media segments. The graph shows the factor increase over the optimal offline algorithm.

new arrival) and $(4,1)$ from the stream started at $(3,0)$. Therefore, the vertically assigned stream from $(3,0)$ gets reset as horizontal and $(4,0)$ is scheduled to merge to $(4,1)$. The grid location $(4,6)$ can merge to the stream started at location $(0,5)$, so this merger is scheduled. At time 5, the grid point $(5,2)$ cannot merge to the other streams in the system, so a horizontal stream is scheduled. There are no further actions except for the merger of the stream started at $(2,2)$ to the stream started at $(0,5)$. Figure 5.30 shows the final merge schedule.

Jin and Bestavros implemented an algorithm for the case of non-sequential access (clients can ask for any part of a media object) based on the closest target heuristic for stream merging [49]. They found that the required server bandwidth for clients receiving two streams at once is close to the required server bandwidth under the receive-all model. The rectilinear algorithm modification proposed here differs slightly from the closest target merging policy. In the rectilinear modification, a client's merge target is limited to the next item on the list, if it exists. The closest merge target for $(5,2)$ in Figure 5.30 is $(5,7)$ (from

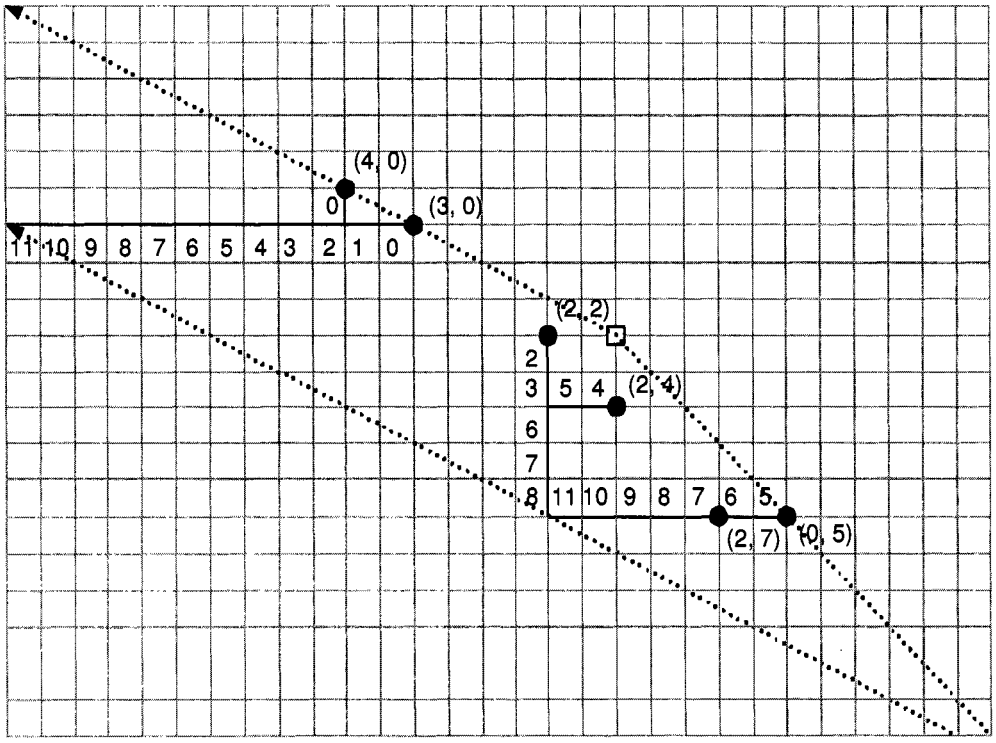


Figure 5.30: The merge schedule produced by the rectilinear algorithm for the standard media-on-demand model with arbitrary access. The client set is $\{(0,5), (2,2), (2,4), (2,7), (3,0), (4,0)\}$ and the media length is 12. All streams must be scheduled to finish inside the boundary marked by dotted lines.

the stream started at $(2,4)$). Thus, $(5,2)$ will merge with $(5,7)$, pulling $(5,7)$ from its merge target at $(5,10)$ (stream started at $(0,5)$). Therefore, $(2,2)$, $(2,4)$, $(3,0)$, and $(4,0)$ will be a single tree instead of what is shown in the figure. For this example, the closest target policy achieves a schedule that has total server bandwidth that is one segment better than the schedule shown in Figure 5.30.

The rectilinear modification supports clients receiving any portion of the media object. If the client exits the system before receiving the final media segment, we can discontinue its stream if no other clients need it. If clients access the media object with wrap-around, meaning the client accesses segment k first and continues sequentially to $L-1$ and then wants segments 0 through k, the rectilinear algorithm can support them. The client who

wants wrap-around access just becomes two client arrivals in the system (one client at (t, k) and one at $(t + L - k, 0)$). These two simulated clients are not in the system simultaneously since (t, k) must get the $L - k$ segments in $L - k$ time, so the first stream allocated finishes at time before or at $t + L - k$ and the second starts at time $t + L - k$.

Optimal Stream Merging For Arbitrary Access

We define the stream merging for arbitrary access problem formally. Let L be the media length. Assume a client c is a pair (t, f), where t is a time greater than or equal to 0 and f is in $\{0, 1, 2, \ldots, L-1\}$. We also assume that a client c wants to receive all segments between f and $N - 1$, inclusively, in time for playback. Given a client set C and a media length L, what is the minimum total server bandwidth necessary to satisfy all clients in C in the receive-2 stream merging model? If we can solve this problem efficiently, we can use the optimal offline solution for a client set C as a benchmark for comparing online algorithms that support arbitrary access.

We show that the optimization problem for stream merging for arbitrary access is NP-hard by showing that the decision version of the problem is NP-complete. The decision version is as follows: Given a client set C of arbitrary accesses, a media length L, and a positive integer K, is there a feasible merge schedule that produces total server bandwidth less than or equal to K? To show that the decision version is NP-complete, we create a reduction from the decision version of the optimal stream merging for time-shifting problem. Recall that we showed this problem is NP-complete in Chapter 4. The decision version of the optimal stream merging with time-shifting problem is: Given a client set C of time-shift pairs and a positive integer K, is there a feasible merge schedule that produces total server bandwidth less than or equal to K. Recall that the length of the live stream is not included in the total server bandwidth cost.

Theorem 5.5.1 *The decision version for optimal stream merging in the arbitrary access model for media of length L is NP-complete.*

Proof: The decision version of the arbitrary access problem is in NP, since once we have a stream merge schedule, we can simply add up the lengths of all the streams and check if

the total is less than or equal to K.

We reduce the time-shifting problem to the arbitrary access problem by altering the client set and creating a media length. Let C be a set of time-shift clients. Each client in C is a pair (t, f) where t is non-negative integer and f is greater than or equal to 0 and less than or equal to t. In the client set C, find the client with the largest time value t_{max}. This is simply a linear search through the client set C, so it is a polynomial time process. Let L be equal to $2t_{max}$. Let C' be the client set C. Add to C' the arrival $(0, 0)$ if it is not included in C'. C' is the client set we use as input to the solver for the decision version of the arbitrary access problem, so we need to verify that each arrival in C' is a valid arrival in the arbitrary access model with media length L. Clearly, each arrival has a valid time t, since every t value in the time-shifting model is non-negative. The f values are all less than or equal to t_{max}, so they are definitely less than L. Figure 5.31 shows a grey triangular area where all the time-shift arrivals are located. The dotted lines show the valid boundary for arrivals in the arbitrary access problem and the grey triangle is clearly within those bounds. The figure also shows the media length L and the black and grey area combined is the space where streams exist in the time-shift problem and the arbitrary access problem. We use the following as inputs to the arbitrary access problem: Given client set C' and media of length $L = 2t_{max}$, is there a feasible merge schedule with total server bandwidth cost less than or equal to $K + L$? (Note: We add L to the total cost since the live broadcast cost is not included in the total cost for the time-shift problem.) Therefore, if we can answer yes/no for the arbitrary access problem, we have the answer to the time-shift problem. Therefore, the decision version of the arbitrary access problem is NP-complete, making the optimization version NP-hard. ∎

The arbitrary access problem is a special case of the arbitrary range access problem. In arbitrary range access, clients request a range of media segments. For example, a client arrival could be indicated by the tuple (t_a, f_a, f_l) where t_a is the arrival time, f_a is the requested first segment, and f_l is the last segment wanted. For example, the client $(1, 5, 10)$ would arrive at time 1, requesting segments 5 through 10 of the media. We can reduce the arbitrary access problem to the arbitrary range access problem by modifying each client $c = (t, f)$ to be $c = (t, f, L)$ for the arbitrary range access problem. If we can answer

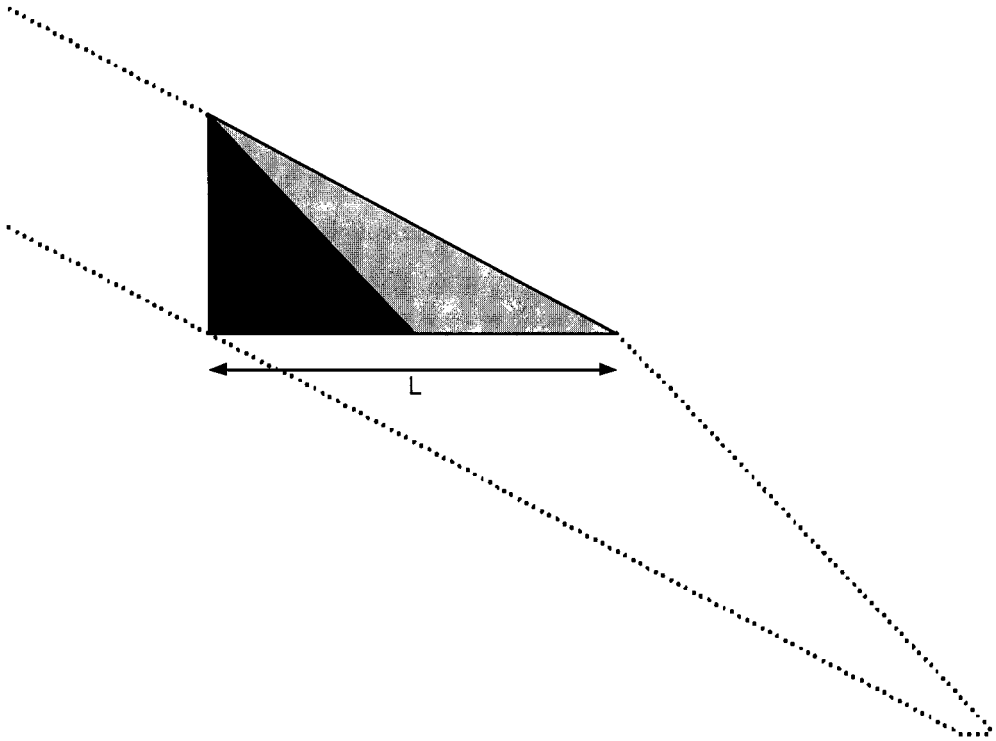


Figure 5.31: A view of the rectilinear grid for the reduction from the time-shifting problem to the arbitrary access problem. The grey area shows the area in which all clients in the time-shifting case are located. The dotted lines show the boundary for the arbitrary access case given a media length L. The grey and black areas together form the space of possible streams in both problems.

yes/no to the arbitrary range access problem, we can answer yes/no for the arbitrary access problem. Thus, the optimization version of the arbitrary range access problem is NP-hard.

5.6 Conclusions and Future Work

We have introduced a new algorithm designed specifically for the stream merging with time-shifting case. The rectilinear algorithm is based on a rectilinear grid, with streams proceeding horizontally (left) and vertically (down) until they reach the live, continuous stream, represented by a horizontal line. The rectilinear algorithm outperforms the proposed modification to the dyadic algorithm, since the rectilinear algorithm is more dynamic. The

rectilinear algorithm can also make use of streams not yet in the system as candidate merge targets during system evolution.

The rectilinear algorithm was adapted for use in the standard media-on-demand model and its performance is comparable to the dyadic algorithm and the ERMT algorithm previously shown to perform well in the literature. The rectilinear algorithm is also suitable for the case of arbitrary accesses, where clients receive parts f through $L-1$ of a media with length L. Through a reduction from the optimal stream merging with time-shifting problem, we showed that the arbitrary access problem is also NP-hard.

Future opportunities include exploring different heuristics for choosing pairs to merge in the rectilinear algorithm. The rectilinear continually chooses the closest pairs that have not been chosen to make merge target decisions. An improvement to the total bandwidth cost could come at the expense of a longer running time when considering multiple merge targets for each active pair in the system.

The modification to the rectilinear algorithm proposed for the standard media-on-demand case does not dynamically select the full-length streams. The modification proposed here uses a cutoff of $L/2$ for arrivals coming after the existing full-length (root) stream. A more dynamic selection of root streams (such as the proposal for the arbitrary access case) may improve the total server bandwidth measure. A more dynamic selection of root streams would be closer to the way the ERMT operates.

We also addressed the requirements for client buffer sizes for the time-shifting model. Future work includes devising online algorithms where clients have a requested first segment and a buffer size limit. We addressed earlier that clients with limited buffer space can exit the system and re-enter when their buffer has space. New algorithms would include decisions about when a client should re-enter the system, if it does not have sufficient buffer space to eventually merge to the live broadcast.

A study similar to [6] could be conducted for the time-shifting case, where the server has a fixed bandwidth capacity and clients are forced to wait if the server's bandwidth is fully consumed. In addition to looking at client waiting times, a study using multiple live streams could be conducted. The server could store and transmit multiple live broadcasts (such as different radio or television stations) and dynamically allocate server bandwidth

for each channel depending on the popularity of the channel. This is similar to a study for the standard media-on-demand model where the server stores multiple movies.

Chapter 6

NETWORK COST FOR MEDIA-ON-DEMAND SYSTEMS

In the previous chapters we assumed that our optimization goal was to minimize server bandwidth to deliver media files to clients. In order for the media file to be transmitted, the client must be connected to the server via a network. A different optimization goal is to minimize the amount of network traffic to deliver media to a set of clients. Under this model, we assume the available server bandwidth is sufficient to satisfy client requests, but we instead want to minimize the amount of data traveling through the network. We also assume the original media-on-demand model where clients want access to a media file from its beginning to its end, clients can buffer up to half the media file, and clients can receive two streams simultaneously. Like before, we assume clients will hierarchically merge with other clients in the system.

In this chapter we introduce the model for minimizing network cost, demonstrate the calculation for total network cost, and present an online algorithm for the network cost model.

6.1 Network Cost Model

The network cost of delivery incorporates the network topology and costs along links in the network. We assume clients can be positioned at any node in the network and the network edges can have any cost greater or equal to 0. For simplicity, we assume the costs along the edges of the network are integer-valued. We assume an overlay network, so the nodes and edges may not represent physical network connections. The costs in our model may represent the number of physical links between nodes, the latency in terms of time to get a packet from one node to another, or the cost of using the physical links between the nodes. The costs might change dynamically over time, but we assume static costs for our algorithms. Because a single server houses the media data, we assume the server is a node

in the overlay network and the overlay network is constructed in the form a tree, with the server as the root of the tree. (The overlay network could be constructed as clients arrive to the system by finding the best link or set of links for each new client arrival to the existing overlay network.) In this section we formally introduce the calculation for the total network cost and provide an example to show that the optimal merge pattern in terms of network cost may not be the optimal pattern in terms of server bandwidth.

6.1.1 Total Network Cost

We quantify the total network cost by calculating the number of media segments distributed over each link in the network. Since each link has an associated cost, the network cost per link is the number of media segments distributed along that link multiplied by the cost of the link. We assume that a merge schedule (set of merge trees) determines the stream merging pattern, the lengths of streams distributed, and to which clients a stream is sent. We also assume that clients will merge to its parent in the merge schedule as soon as possible.

Let DT be the network distribution tree that contains the server S and the set of clients C. Let E be the complete set of edges in DT. Let e be a single edge in E and let $c(e)$ be the cost of the edge e. Let $n(e)$ be the number of media segments distributed along e for the entire client set C. The total network cost given DT and the client merge schedule is the following:

$$\sum_{e \in E} c(e)n(e) \tag{6.1}$$

We simulate the merge schedule to find the number of media segments traveling across each link in the network. Since we already have the set of merge trees specifying the merge schedule, we use them to calculate the total network cost. We call the set of merge trees for the client set C the merge forest. For each node in the merge forest, we calculate the length of the stream started for that client along with the set of clients listening to each segment of that stream. Because later clients can extend streams already in the system, we perform the calculation in the order of client arrivals. The network cost model differs slightly from the original media-on-demand model, since now we can have clients arriving at the same

time, but they are no longer treated as a single client from the perspective of the server.

The following algorithm calculates the total network cost given a merge forest MF for a client set C. We assume C is a sorted list of clients according to increasing arrival time. In the case of clients arriving at the same time, they are ordered by increasing distance from their root node in MF. Let L be the number of segments in the media file. Let $\ell(c)$ be the length of the stream started for the client arrival c. For each stream, we keep track of the clients receiving each segment of the stream. Let $R(c, s)$ be a set of clients receiving media segment s from the stream started at c's arrival. Let $t(c)$ be the arrival time of c, and let $p(c)$ be the parent node of c in MF. The algorithm proceeds in two stages. The first stage calculates the lengths of streams along with the client set listening to those streams. The second stage uses the stream length information and the distribution tree DT to calculate the network cost.

1. Calculate Stream Lengths of Receiving Clients: Process each client c in C in order:

   - If c is a root node in MF, $\ell(c) = L$ and add c to each set $R(c, i)$ where $0 \leq i < L$.
   - Else, $\ell(c) = t(c) - t(p(c))$ and add c to each set $R(c, i)$ where $0 \leq i < \ell(c)$. For each ancestor a in MF for c, perform the following actions, starting with $p(c)$ and ending with its ancestor at a root node.
     - Calculate s:
       * If a has a grandchild g on the path to c, $s = t(g) + \ell(g)$.
       * Else, $s = t(c)$
     - If a is a root node, $\ell(a)$ does not change and add c to each set $R(a, i)$ where $s - t(a) \leq i < L$.
     - Else, the stream for a is extended such that $\ell(a) = 2t(c) - t(a) - p(a)$. Add c to each set $R(a, i)$ where $s - t(a) \leq i < \ell(a)$.

2. Calculate Network Cost: For each stream s and set of receiving clients found in stage 1, perform the following:

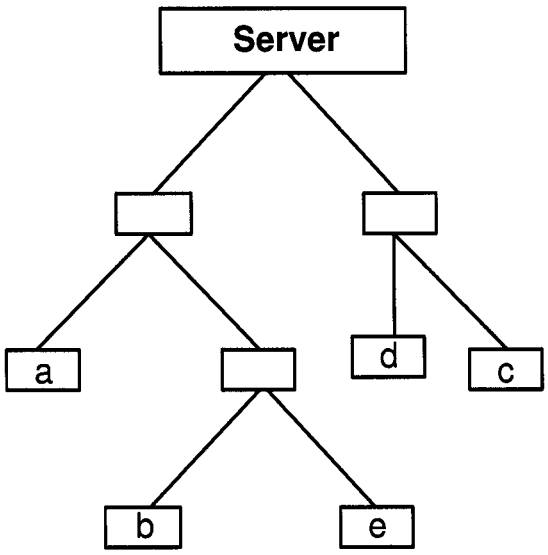


Figure 6.1: Distribution tree for client arrivals $\{a = 0,\ b = 2,\ c = 4,\ d = 5,$ and $e = 6\}$. Each link in the distribution tree has cost 1.

- For each segment in s, find the links in the distribution tree traversed for the set of receiving clients. Total the costs of all links traversed for the segment and add this cost to the network cost.

Figure 6.1 shows a distribution tree for a set of clients. Each link has a cost of 1. The client arrivals include the following: $\{a = 0, b = 2, c = 4, d = 5, e = 6\}$. The media length is 16 and the merge schedule, illustrated as a merge tree, is shown in Figure 6.2. The streams are indicated to the right of each node in Figure 6.2 and the set of receiving clients are listed below each stream segment. For example, $R(a, 4) = \{a, b, c\}$ since those three clients are receiving segment 4 from the stream initiated for client a. This stream information is the product of running phase one of the network cost algorithm.

Figure 6.3 shows the existing streams in the system at time 5. There are three concurrent streams in the system, each distributing a media segment. The directed paths show which links are traversed for each of the streams.

To calculate the total network cost for the schedule shown in Figure 6.2, we simply find the network cost for each of the streams. We refer to streams by the client that initiated

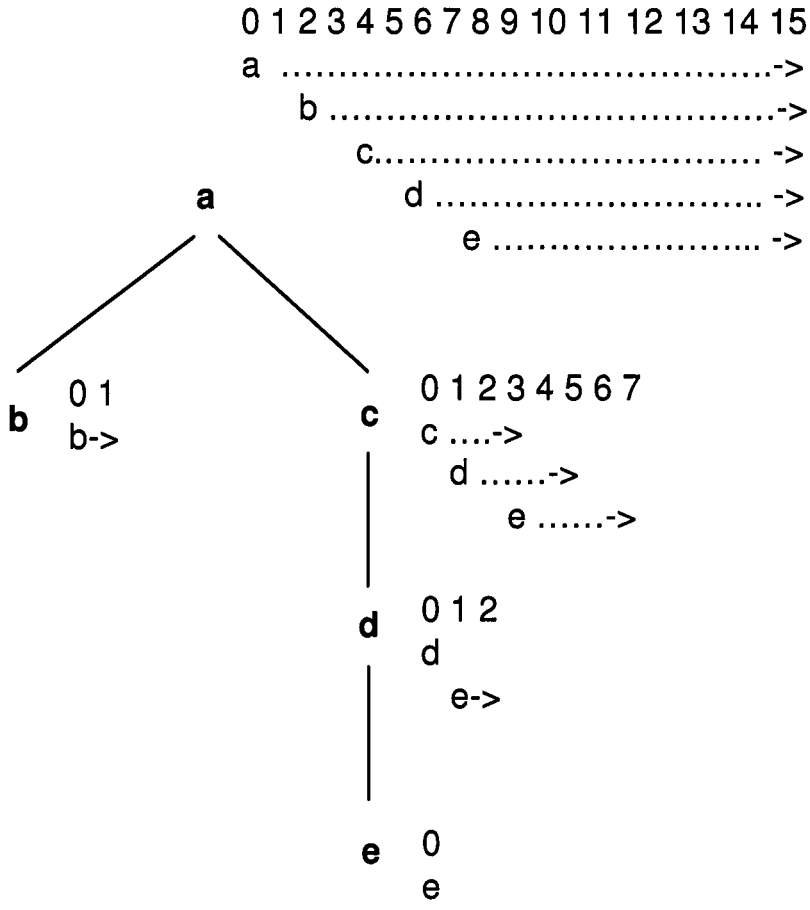


Figure 6.2: Merge tree for client arrivals $\{a = 0, b = 2, c = 4, d = 5, \text{and } e = 6\}$. Each node has its associated stream, along with the receivers for each segment along the stream. This is an example of the resulting calculation from phase one of the total network cost algorithm.

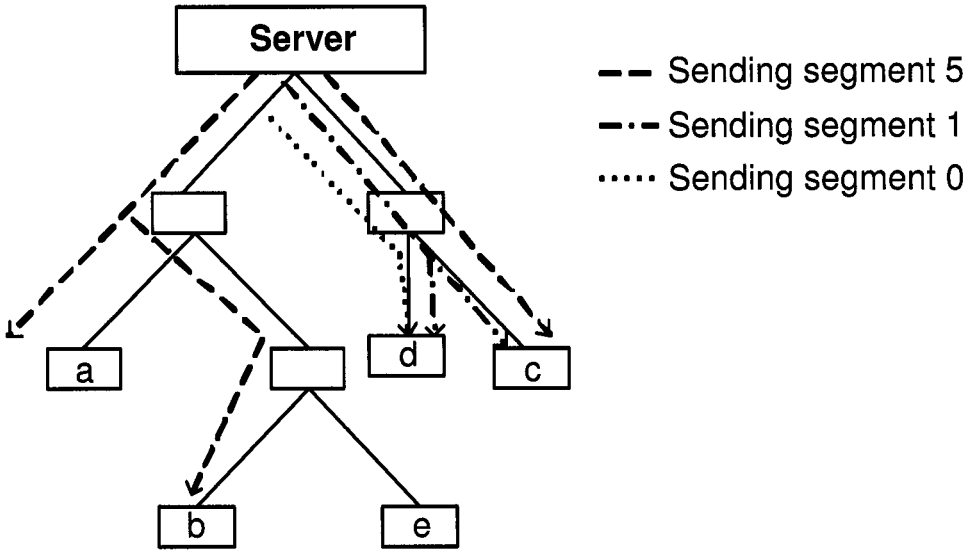


Figure 6.3: Distribution tree for clients at time 5. There are three concurrent streams in the system at time 5 and the directed paths indicate the links traversed by each stream.

them. Stream a has length 16. Segments 0 and 1 are sent to a only, with each segment costing two in the distribution network. Segments 2 and 3 are sent to a and b, each with a network cost of 4. Segments 4 and 5 are sent to a, b, and c, each with a network cost of 6. Segments 6 and 7 are sent to a, b, c, and d, each with a network cost of 7. The last 8 media segments are sent to all five clients, each with a network cost of 8. The total network cost for stream a is 102. The cost for stream b is 6. The cost for stream c is 30. The cost for stream d is 8 and the cost for stream e is 3. Thus, the entire network cost for all streams in the network is 149.

6.1.2 Optimal Network Cost

Calculating the optimal merge schedule in terms of server bandwidth for the original media-on-demand model can be done efficiently [8]. One reason is that the optimal merge tree is always a preorder tree, which results in an efficient dynamic programming solution. A preorder tree obeys the property that if the tree is traversed in a preorder manner, then the clients are visited in their arrival order. A preorder traversal first processes the root

and the recursively applies the preorder traversal to the left subtree and then to the right subtree. The preorder property for optimal merge trees for the original media-on-demand model guarantees that we can split the set of clients into two sets: $\{c_1, c_2, \ldots c_{j-1}\}$ and $\{c_j, c_{j+1}, \ldots c_n\}$ for n client arrivals such that c_j is a child of the root and $\{c_{j+1}, c_{j+1}, \ldots c_n\}$ are all descendants of c_j in the merge tree. Breaking the client arrival set into two disjoint sets allows for a dynamic programming solution. To find the merge forest that minimizes total server bandwidth, we can efficiently find the clients to start new root streams [8]. Once these root clients are determined, we know that all clients arriving after a root client r and before root client $r+1$ will merge to r. This property guarantees that the optimal merge forest consists of trees such that each tree contains a consecutive sequence of clients, with respect to their arrival times. Therefore, clients will always merge to the most recently started root stream.

It is not the case that clients should always merge to the most recently started root stream when trying to optimize total network cost. Because of this fact, an efficient algorithm for finding the optimal merge schedule in terms of network cost is most likely NP-hard. (We leave this as future work.) We provide an example distribution network and set of client arrivals to show that the optimal schedule for network cost has a client merging to a root stream other than the most recently started one. Let the media length $L = 16$ and let the client set include the following arrivals: $\{a = 0, b = 6, c = 7, d = 13\}$. The distribution tree, shown in Figure 6.4, has a server with two outgoing links. Each of these outgoing links enters intermediate nodes. One intermediate node has an outgoing link to a and another outgoing link to c. The second intermediate node has an outgoing link to b and another outgoing link to d. Each client is two links from the server. Figure 6.5 shows the 12 feasible merge schedules for the four clients. Notice that merge schedule (VIII) has the best total network cost and c merges to a rather than merging to the root stream b. The figure also highlights the fact that the optimal schedule for total server bandwidth is not necessarily the same as the optimal schedule for total network cost and vice verse. Merge schedule (XI) achieves the optimal server bandwidth for the arrivals.

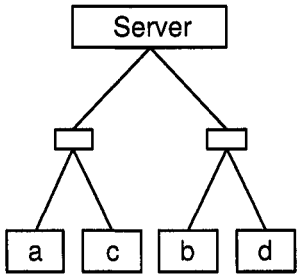


Figure 6.4: Distribution tree for clients $a = 0$, $b = 6$, $c = 7$, and $d = 13$.

6.2 Algorithms for Network Cost

We present online stream merging algorithms to minimize total network cost. The purpose of this section is not provide a comprehensive evaluation of the proposed algorithms; instead, we demonstrate how each of the algorithms makes merge decisions. We use a running example to showcase the resulting merge schedules produced for each algorithm. The distribution tree for our running example is shown in Figure 6.6. The arrival times are as follows: $a = 0$, $b = 3$, $c = 5$, $d = 6$, $e = 6$, $f = 7$, $g = 8$, $h = 9$, $i = 10$, and $j = 10$. Notice that both d and e arrive at time 6. The original media-on-demand algorithms would treat d and e as a single client, since they arrive concurrently. Our running example assumes a media length of 18.

The first proposed approach is simply to use a stream merging algorithm designed to minimize server bandwidth. Both the 2-Dyadic and ERMT algorithms have been shown to perform comparably well in practice [6]. We illustrate the resulting merge tree and total network cost when running 2-Dyadic on our example. We could apply the ERMT algorithm, but calculating the total network cost is more difficult for algorithms in which merge targets can change dynamically. The ERMT algorithm does not necessarily create static merge schedules, so the target clients can change throughout the evolution of the system. Calculating the total network cost would require more bookkeeping to keep track of merge target changes during the execution of the ERMT algorithm.

When we apply the original 2-Dyadic algorithm on the arrival sequence shown in Table

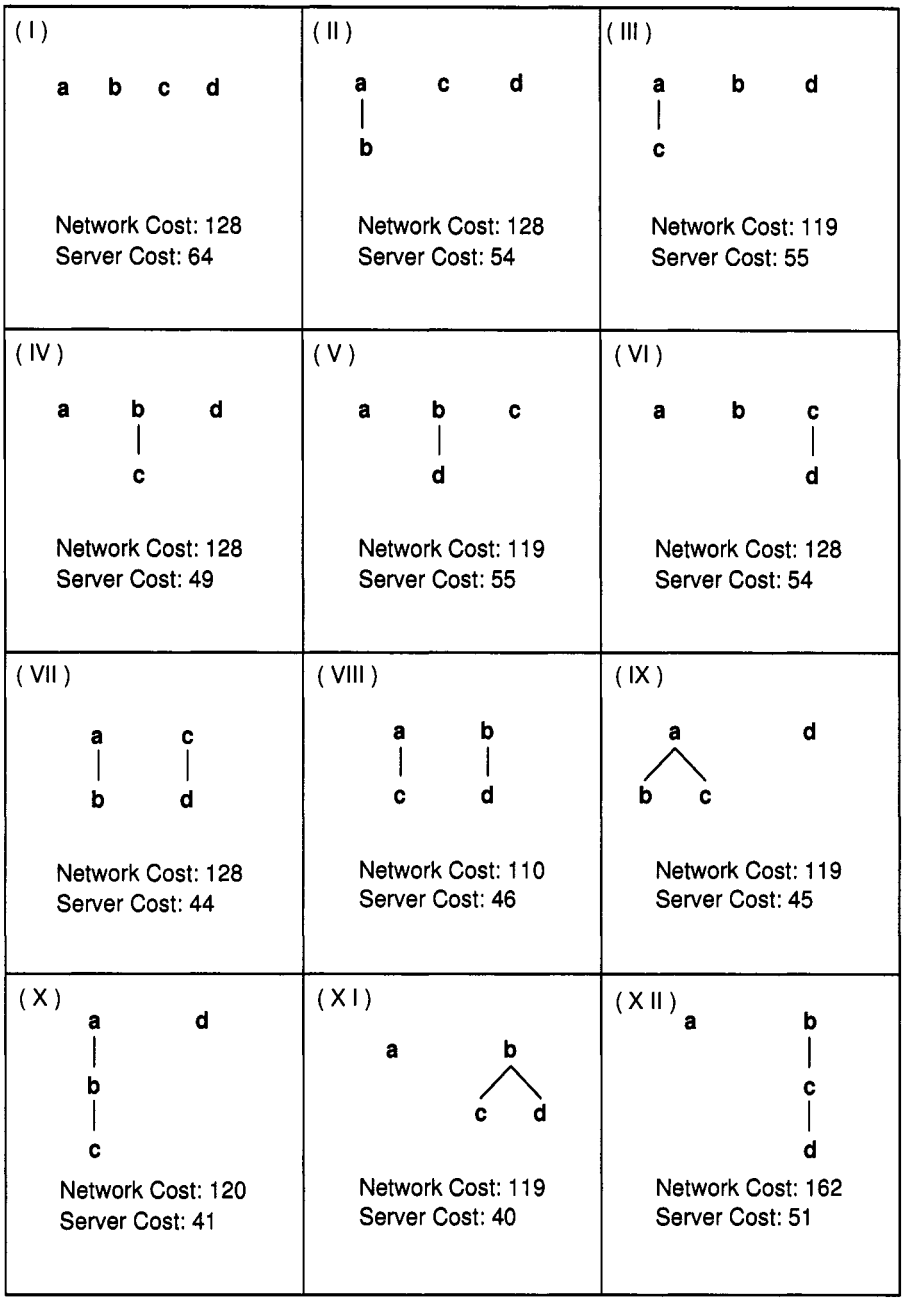


Figure 6.5: All possible merge forests for client arrivals $a = 0$, $b = 6$, $c = 7$, and $d = 13$ and media length 16.

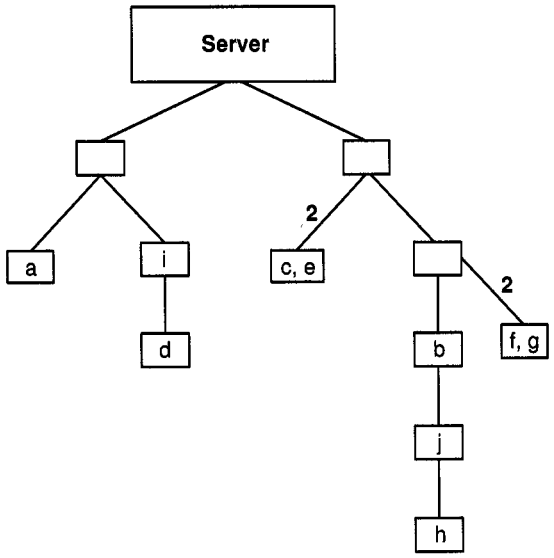


Figure 6.6: Distribution tree for client arrivals $a = 0$, $b = 3$, $c = 5$, $d = 6$, $e = 6$, $f = 7$, $g = 8$, $h = 9$, $i = 10$, and $j = 10$. The cost of the unlabeled edges in the distribution tree is 1.

6.1, we get the merge forest shown in Figure 6.7. The total network cost for this schedule is 382, which can be verified by running the total network cost algorithm on the merge forest in Figure 6.7.

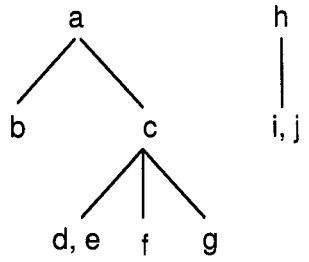


Figure 6.7: Merge forest created by running the 2-Dyadic algorithm on the arrival sequence shown in Table 6.1 with media length 18. The total network cost for this solution is 382.

A second approach takes into consideration the physical location of clients in the distribution tree. Instead of blindly running an algorithm designed to minimize server bandwidth,

Table 6.1: The arrival sequence for the running example used to compare online algorithms for minimizing network cost.

Client	Time
a	0
b	3
c	5
d	6
e	6
f	7
g	8
h	9
i	10
j	10

we partition the distribution tree into disjoint sets and run a standard media-on-demand algorithm on each set. We partition the distribution tree into sets such that all clients in the same set have a common ancestor that is not the server. In the running example with distribution tree shown in Figure 6.6, we partition the network into the following sets: $\{a, d, i\}$ and $\{b, c, e, f, g, h, j\}$. By running the dyadic algorithm independently on each set, we get the merge forest shown in Figure 6.8. The total network cost for this schedule is 344.

6.2.1 Physically Closest Merge Target

The second approach above partitions the distribution tree into disjoint sets, so proximity of clients within the distribution tree is taken into account. We present a new algorithm that takes into consideration the current stream data on the closest link to a newly arrived client. Clearly, this requires more bookkeeping, since we need to know which data is being transmitted along the edges in the distribution tree at all time points in the system. When a new client arrives to the system, we find the closest link (along the path to the server) that is transmitting data. The first link we find that is transmitting data must eventually transmit

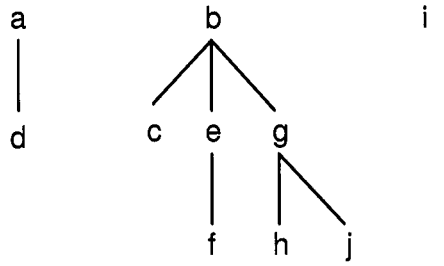


Figure 6.8: Merge forest created by running the 2-Dyadic algorithm on the two disjoint sets $\{a, d, i\}$ and $\{b, c, e, f, g, h, j\}$ of the arrival sequence shown in Table 6.1 with media length 18. The total network cost for this solution is 344.

a root stream, since all clients eventually merge to a root stream. If this root stream is currently transmitting a segment that is $L/2$ or greater, we start a new root stream for the newly arrived client. If the link is transmitting multiple streams, we find the stream that is sending the lowest numbered media segment and use the client that initiated the stream as the newly arrived client's parent in the merge tree.

Before formally presenting the algorithm, we specify helpful notation. Let DT be the distribution tree that includes all client arrivals in the client set C. Each client c will be represented by a lower-case letter. Let MF be the merge forest, which represents the merge schedule for the clients. Each client c exists in both DT and MF. The algorithm creates MF for the clients while updating information along the distribution tree. Let $LP(c)$ represent the link path from c to the server in DT. Let $CL(c)$ be the closest link along $LP(c)$ that contains usable media data. The data is usable if the currently transmitted segment along the root stream is less than $L/2$. Let $t(c)$ be the arrival time of client c. Let $P(c)$ be the parent node of c in MF and let $A(c)$ be the ancestor path from c to the root node in MF.

We represent streams by tuples with the following information: (client, current/start time, current segment, last segment). If s is a stream tuple, then $c(s)$ is the client, $t(s)$ is the current/start time, $s(s)$ is the current segment, and $\ell(s)$ is the last segment. The client is the arrival that initiated the stream. The current/start time is the current time in the system or the time in which the stream will start along the link (whichever is later). The current segment is the currently transmitted segment along the stream, assuming the

current/start time is equal to the current time in the system. The last segment is the final segment that will transmitted along the stream. Each link in the distribution tree contains a list of streams that are currently being transmitted. We remove the stream from the link when it terminates in the system.

The physically closest merge target algorithm proceeds as follows. Upon client arrival c, perform the following actions:

1. Update stream information along all links in $LP(c)$ to reflect the current time. For each stream s along $LP(c)$, calculate $d(s) = t(c) - t(s)$. If $d(s) > 0$ then set $t(s)$ to be $t(c)$. Add $d(s)$ to $s(s)$ if $d(s) > 0$. If $s(s) > \ell(s)$, then remove s from the link.

2. Find $CL(c)$.

   - If $CL(c)$ is null, start a new root stream for c. Add the stream $s = (c, t(c), 0, L-1)$ to each link in $LP(c)$.
   - Else if the root stream rs (the stream with last segment equal to $L-1$ and with the smallest current segment) along $CL(c)$ has $s(rs) > L/2$, then start a new root stream for c. Add the stream $s = (c, t(c), 0, L-1)$ to each link in $LP(c)$.
   - Else:
     (a) Find stream s' with smallest $s(s')$ along $CL(c)$. Set $c(s')$ as $P(c)$ in MF.
     (b) Add the stream $s = (c, t(c), 0, t(c) - t(c(s')) - 1)$ to each link in $LP(c)$.
     (c) For each non-root ancestor a in $A(c)$ of MF, calculate the new last segment $\ell(a) = 2t(c) - t(a) - t(P(a)) - 1$. Set $\ell(s)$ to be $\ell(a)$ for streams s with $c(s)$ equal to a along all links in $LP(c)$.
     (d) Build the collection of streams along links between c and $CL(c)$ in DT. For each non-root ancestor a in $A(c)$ of MF, add the stream $s = (a, t(c) + 1 + \ell(g(a)), 1 + \ell(d(a)), \ell(a))$ along links between c and $CL(c)$ in DT, where $g(a)$ is the grandchild of a along path toward c in MF and $d(a)$ is the daughter of a along path toward c in MF. If there is no $g(a)$, then $\ell(g(a)) = -1$. For the root ancestor r in $A(c)$ of MF, add the stream $s = (r, t(c) + 1 + \ell(g(r)), 1 + \ell(d(r)), L - 1)$ to links between c and $CL(c)$ in DT.

Example Execution

In this section we show how the physically closest merge target algorithm executes on client arrivals $a = 0$, $b = 3$, $c = 6$, $d = 6$, $e = 7$, $f = 8$, $g = 9$, $h = 10$, and $i = 10$ with the distribution tree shown in Figure 6.9. The media has length 16. Note that this is a different example than our running example, since the example was chosen to highlight all the possible paths through the algorithm.

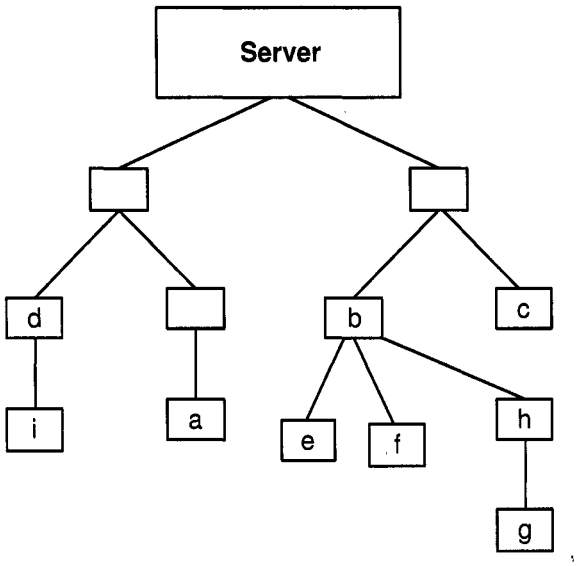


Figure 6.9: Distribution tree for physically closest merge target example.

Figure 6.10 shows the streams along the links along paths in the distribution tree after each client arrival. The algorithm proceeds from left to right and top to bottom in the figure. Only the link path from the client to the root stream is shown for each client arrival. Figure 6.11 shows the resulting merge forest.

When client a arrives, there are no streams along its link path, so a new root stream is initiated and its stream is added to each link in the link path. When client b arrives, there are no streams along its link path, so a new root stream is initiated. When client c arrives, we find that its closest link has stream b, so b becomes c's merge target and we add the necessary streams to c's link path. When d arrives, its closest link with stream

data has stream a, so a becomes d's merge target. When client e arrives, its closest link has stream b, so b is e's merge target and we add the necessary streams along e's link path. When f arrives, its closest link has streams b and e. Stream e is transmitting the closest media segment to 0, so e becomes f's merge target. The length of stream e is extended and its new last segment values are updated along the link path. (Revised last segments are shown in bold in Figure 6.10.) Notice that f does not start receiving stream b until time 9, since it receives streams e and f for the first time unit. When g arrives, its closest link is transmitting streams b and e. Since e has a lower current segment, e becomes g's merge target. The last segment for e is updated to 7 to reflect the fact that the arrival of g extends the stream for e. When h arrives, its closest link is connected to the node for h and the stream delivering the closest segment to 0 is stream g. Thus, g becomes h's merge target and both e and g get new last segments since they are non-root ancestors of h. When i arrives, its closest target is delivering streams a and d. But, a is the root stream and is currently delivering segment 10 which is greater than $L/2 = 8$. Thus, a new root stream for i is initiated.

Let us return to our running example, so we can compare how the physically closest merge target algorithm performs to the two approaches presented earlier. The physically closest merge target algorithm creates the merge schedule shown in Figure 6.12 for the arrivals in Table 6.1. The total network cost for this schedule is 322, which is better than the network costs found for 2-Dyadic (cost of 382) and network-aware 2-Dyadic (cost of 344) algorithms.

6.2.2 Special Cases

If we are trying to minimize both the total network bandwidth and total server bandwidth cost, we can construct a virtual link to represent the server bandwidth in the distribution tree. Zhao *et al.* show the minimum required network bandwidth for certain distribution trees, client sites within the tree, and arrival rates [78]. They show that they can achieve close to minimal network bandwidth in a fan-out topology by aggregating clients from the same client site. Achieving minimal network bandwidth can come at the expense of server

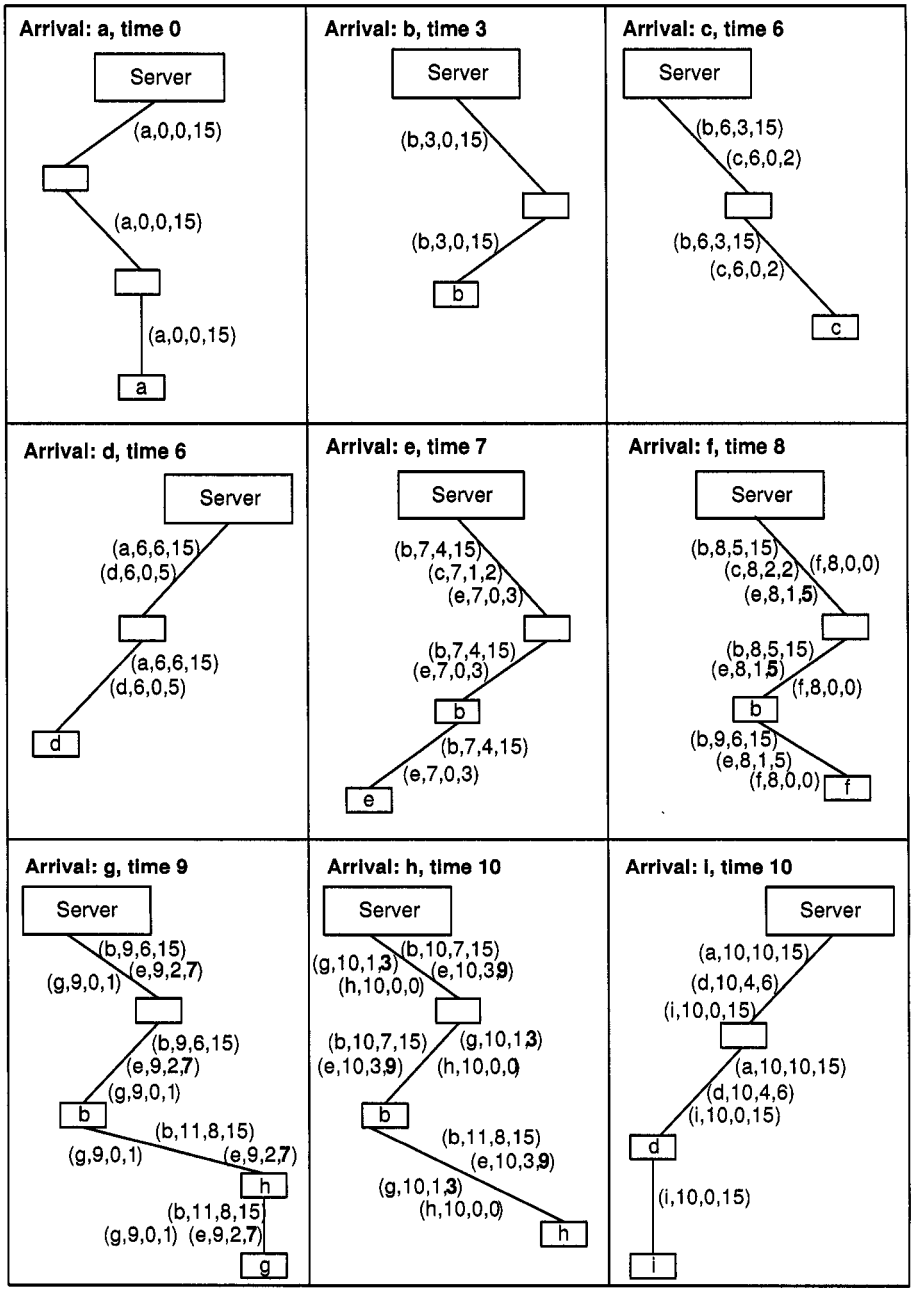


Figure 6.10: Physically closest merge target algorithm execution on the links of the distribution tree.

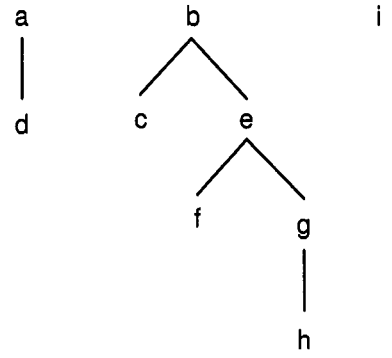


Figure 6.11: Resulting merge forest created by physically closest merge target algorithm.

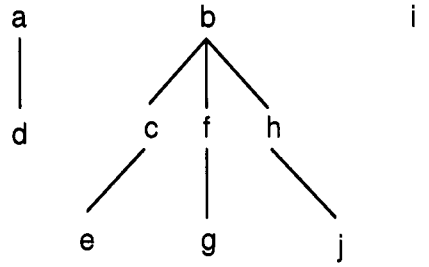


Figure 6.12: Resulting merge forest for client arrivals in Table 6.1 using the physically closest merge target algorithm. The total network cost for this solution is 322.

bandwidth. In order to take both costs into consideration, we can simulate the cost of the server bandwidth by adding a link from a virtual server in the distribution tree to the server. We can specify how much the server bandwidth is weighted against network bandwidth with the cost of the link between the virtual server and server. Now, all network costs will be calculated as if all streams start at the virtual server. Every stream must travel the link between the server and virtual server, so the introduced link is a substitute for server bandwidth.

If the distribution tree is shaped as a star, shown in Figure 6.13, with no co-located clients, then the total network cost will be the same regardless of the merge schedule created. All L media segments must travel along each link and there are no shared links among the clients, so the total network cost will be the total link cost multiplied by the media length

L. In this case, it makes sense to use an algorithm that produces minimal total server bandwidth.

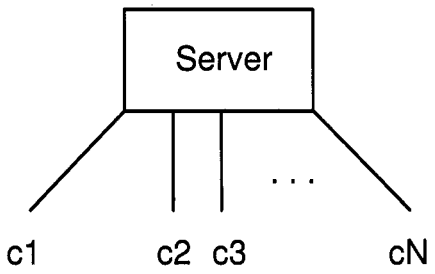


Figure 6.13: Star topology for client arrivals $c1$ through cN.

6.3 Conclusions and Future Work

The purpose of this chapter was to introduce the network cost model and propose online algorithms to achieve minimal total network cost. We have shown how to calculate the total network cost given a merge schedule depicted as a merge forest. We also proposed a new algorithm that chooses merge targets based on the closest client in the distribution tree. We showed how three different approaches produced three different merge schedules and total network cost on a single example.

Future work includes finding an optimal offline algorithm for total network cost or proving that finding the optimal offline solution is NP-hard. Because we showed an example where the optimal merge schedule had client c merging to the root client a instead of the root client b, it seems likely that there is no efficient offline solution for the total network cost problem.

Another future direction is to implement algorithms designed to optimize for network cost (like the one proposed in this chapter) and compare them experimentally on a variety of topologies. Algorithms designed for minimizing network bandwidth can then be evaluated against each other and against algorithms designed to minimize total server bandwidth. Zhao *et al.* did perform experiments on a few simple topologies and found that minimizing network cost came at the expense of higher server bandwidth cost.

Finally, another line of future research includes placing capacities along the links and devising algorithms such that concurrent distribution of streams along links does not exceed the capacity along links. This might require using a distribution graph instead of a tree, so that several paths to the server in the distribution graph can be considered. Otherwise, a client may not be served immediately if any link along its link path is saturated.

BIBLIOGRAPHY

[1] C. Aggarwal, J. Wolf, and P. S. Yu. On Optimal Piggyback Merging Policies for Video-On-Demand Systems. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 200–209, 1996.

[2] C. C. Aggarwal, J. L. Wolf, and P. S. Yu. On Optimal Batching Policies for Video-on-Demand Storage Servers. In *Proceedings of the Fourth ACM International Conference on Multimedia*, pages 253–258, 1996.

[3] J. M. Almeida, D. L. Eager, M. Ferris, and M. K. Vernon. Provisioning Content Distribution Networks for Streaming Media. In *Proceedings of the IEEE INFOCOM Conference on Computer Communications*, pages 1746–1755, 2002.

[4] D. P. Anderson. Metascheduling for Continuous Media. *ACM Transactions on Computer Systems*, 11(3):226–252, August 1993.

[5] A. Bar-Noy, J. Goshi, and R. E. Ladner. Off-line and On-line Guaranteed Start-Up Delay for Media-on-Demand with Stream Merging. In *Proceedings of the 15th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '03)*, pages 164–173, 2003.

[6] A. Bar-Noy, J. Goshi, R. E. Ladner, and K. Tam. Comparison of Stream Merging Algorithms for Media-on-Demand. *Multimedia Systems Journal*, 9:411–423, 2004.

[7] 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.

[8] 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.

[9] P. Basu, R. Krishnan, and T. D. C. Little. Optimal Stream Clustering Problems in Video-on-Demand. Technical Report MCL 04-22-98, Boston University, 1998.

[10] M. Brazil, D. A. Thomas, and J. F. Weng. A Polynomial Time Algorithm for Rectilinear Steiner Trees with Terminals Constrained to Curves. *Networks*, 33:145–155, 1999.

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

[12] Y. Cai and K. A. Hua. Sharing Multicast Videos Using Patching Streams. *Multimedia Tools and Applications*, 22(2):125–146, 2003.

[13] S. W. Carter and D. D. E. Long. Improving Bandwidth Efficiency on Video-on-Demand Servers. *Computer Networks*, 31(1-2):111–123, 1999.

[14] S.-H. G. Chan and F. Tobagi. Distributed Servers Architecture for Networked Video Services. *IEEE/ACM Transactions on Networking*, 9(2):125–136, April 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 27th International Symposium on Mathematical Foundations of Computer Science (MFCS '02)*, pages 188–200, 2002.

[16] W.-T. Chan, T.-W. Lam, H.-F. Ting, and P. W. H. Wong. On-line stream merging in a general setting. *Theoretical Computer Science*, 296:27–46, 2003.

[17] W.-T. Chan, T.-W. Lam, H.-F. Ting, and 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 '02) - Workshop on Scheduling and Telecommunications*, pages 2165–2172, 2002.

[18] W.-T. Chan, T.-W. Lam, H.-F. Ting, and W.-H. Wong. A Unified Analysis of Hot Video Schedulers. In *Proceedings of the ACM Symposisum on Theory of Computing (STOC '02)*, pages 179–188, 2002.

[19] M. Chesire, A. Wolman, G. Voelker, and H. Levy. Measurement and Analysis of a Streaming Media Workload. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS '01)*, pages 1–12, 2001.

[20] C. Y. Choi and M. Hamdi. A Scalable Video-On-Demand System Using Multi-Batch Buffering. In *Proceedings of the IEEE GLOBECOM Global Communications Conference*, pages 1989–1993, 2001.

[21] E. G. Coffman, P. Jelenković, and P. Momčilović. The Dyadic Stream Merging Algorithm. *Journal of Algorithms*, 43(1):120–137, 2002.

[22] B. Cohen. Incentives Build Robustness in BitTorrent. http://bitconjurer.org/BitTorrent/ documentation.html, 2003.

[23] Comcast. Comcast cable. http://www.comcast.com.

[24] J. Cong, A. B. Kahng, and K.-S. Leung. Efficient Algorithms for the Minimum Shortest Path Steiner Arborescence Problem with Applications to VLSI Physical Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(1):24–39, January 1998.

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

[26] G. Dammicco and U. Mocci. Optimal Server Location in VOD Networks. In *Proceedings of the IEEE GLOBECOM Global Communications Conference*, pages 197–201, 1997.

[27] A. Dan, D. Sitaram, and P. Shahabuddin. Scheduling Policies for an On-Demand Video Server with Batching. In *Proceedings of Second ACM International Conference on Multimedia*, pages 15–23, 1994.

[28] S. E. Deering and D. R. Cheriton. Multicast Routing in Datagram Internetworks and Extended LANs. *ACM Transactions on Computer Systems*, 8(2):85–110, May 1990.

[29] D.-Z. Du, J. M. Smith, and J. H. Rubinstein, editors. *Advances in Steiner Trees.* Kluwer Academic Publishers, 2000.

[30] D. Eager, M. Vernon, and J. Zahorjan. Optimal and Efficient Merging Schedules for Video-on-Demand Servers. In *Proceedings of the 7th International Conference on Multimedia*, pages 199–203, 1999.

[31] D. Eager, M. Vernon, and J. Zahorjan. Bandwidth Skimming: A Technique for Cost-Effective Video-on-Demand. In *Proceedings of Multimedia Computing and Networking (MMCN '00)*, pages 206–215, 2000.

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

[33] H. Eriksson. MBONE: The Multicast Backbone. *Communications of the ACM*, 37(8):54–60, August 1994.

[34] A. Fiat and G. J. Woeginger, editors. *Online Algorithms: The State of the Art.* Springer-Verlag, 1998.

[35] N. L. S. Fonseca and H. K. Rubinsztejn. Channel Allocation in True Video-on-Demand Systems. In *Proceedings of the IEEE GLOBECOM Global Communications Conference*, pages 1999–2004, 2001.

[36] L. Gao and D. 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.

[37] 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.

[38] J. Goshi. Efficient and secure media delivery. Ph.D Dissertation, University of Washington, Seattle, 2004.

[39] C. Griwodz. The Use of Stream Merging Mechanisms in a Hierarchical CDN. In *Proceedings of Multimedia Computing and Networking (MMCN '04)*, pages 1–15, 2004.

[40] M. Guo and M. H. Ammar. Scalable live video streaming to cooperative clients using time shifting and video patching. In *Proceedings of the IEEE INFOCOM Conference on Computer Communications*, pages 1501–1511, 2004.

[41] M. Guo, M. H. Ammar, and E. W. Zegura. Selecting among Replicated Batching Video-on-Demand Servers. In *Proceedings of the 12th IEEE International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV '02)*, pages 155–163, 2002.

[42] K. A. Hua, Y. Cai, and S. Sheu. Patching: A multicast technique for true video-on-demand services. In *Proceedings of the 6th ACM International Conference on Multimedia*, pages 191–200, 1998.

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

[44] F. K. Hwang, D. S. Richards, and P. Winter. *The Steiner Tree Problem.* Elsevier Science Publishers, 1992.

[45] R.-H. Hwang and J.-J. Wu. Scheduling Policies for an VOD System over CATV Networks. In *Proceedings of the IEEE GLOBECOM Global Communications Conference*, pages 438–442, 1997.

[46] A. O. Ivanov and A. A. Tuzhilin. *Minimal Networks: The Steiner Problem and Its Generalizations.* CRC Press, 1994.

[47] R. Janakiraman, M. Waldvogel, and L. Xu. Fuzzycast: Efficient Video-on-Demand over Multicast. In *Proceedings of the IEEE INFOCOM Conference on Computer Communications*, pages 920–929, 2002.

[48] R. Janakirman, M. Waldvogel, W. Deng, and L. Xu. Achieving Scalable and Efficient Video-on-Demand Over Multicast. IBM Research Report RZ-3495, December 2002.

[49] S. Jin and A. Bestavros. Scalability of Multicast Delivery for Non-sequential Streaming Access. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 97–107, 2002.

[50] B. A. Julstrom and A. Antoniades. Two Hybrid Evolutionary Algorithms for the Rectilinear Steiner Arborescence Problem. In *Proceedings of the ACM Symposium on Applied Computing*, pages 980–984, 2004.

[51] M. Kwon and S. Fahmy. Topology-Aware Overlay Networks for Group Communication. In *Proceedings of IEEE International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV '99)*, pages 127–136, 1999.

[52] 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.

[53] W. Liao and V. O. K. Li. The Split and Merge (SAM) Protocol for Interactive Video-on-Demand Systems. In *Proceedings of the IEEE INFOCOM Conference on Computer Communications*, pages 1349–1356, 1997.

[54] J. Liu, B. Li, and Y.-Q. Zhang. Adaptive Video Multicast over the Internet. *IEEE MultiMedia*, 10(1):22–33, January/March 2003.

[55] H. Ma and K. Shin. Multicast Video-on-Demand Services. *ACM SIGCOMM Computer Communication Review*, 32(1):31–43, January 2002.

[56] A. Mahanti, D. L. Eager, M. K. Vernon, and D. J. Sundaram-Stukel. Scalable On-Demand Media Streaming with Packet Loss Recovery. *IEEE/ACM Transactions on Networking (TON)*, 11(2):195–209, April 2001.

[57] C. K. Miller. *Multicast Networking and Applications.* Addison-Wesley, 1999.

[58] T. Osawa. Tree Construction and Stream Assignment Algorithms for Multicast of Multimedia Streams. Masters Thesis, University of Washington, Seattle, 1997.

[59] C. H. Papadimitriou. *Computational Complexity.* Addison-Wesley Publishing Company, 1994.

[60] S. Ramesh, I. Rhee, and K. Guo. Multicast with Cache (Mcache): An Adaptive Zero-Delay Video-on-Demand Service. In *Proceedings of the IEEE INFOCOM Conference on Computer Communications*, pages 85–94, 2001.

[61] S. K. Rao, P. Sadayappan, F. K. Hwang, and P. W. Shor. The Rectilinear Steiner Arborescence Problem. *Algorithmica*, pages 277–288, 1992.

[62] 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.

[63] 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.

[64] S. Sheu, K. A. Hua, and T.-H. Hu. Virtual Batching: A New Scheduling Technique for Video-on-Demand Servers. In *Proceedings of the Fifth International Conference on Database Systems for Advanced Applications*, pages 481–490, 1997.

[65] W. Shi and C. Su. The Rectilinear Steiner Arborescence Problem is NP-Complete. In *Proceeedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 780–787, 2000.

[66] W. D. Sincoskie. Video on demand: Is it feasible? In *Proceedings of the IEEE GLOBECOM Global Communications Conference*, pages 201–205, 1990.

[67] M. Sipser. *Introduction to the Theory of Computation.* PWS Publishing Company, 1997.

[68] W. Stallings. *Data and Computer Communications.* Prentice-Hall, Inc., 2000.

[69] H. Tan, D. L. Eager, and M. K. Vernon. Delimiting the range of effectiveness in scalable on-demand streaming. *Performance Evaluation*, 49(1-4):387–410, 2002.

[70] M. A. Tantaoui, K. A. Hua, and S. Sheu. A Broadcast Technique for Providing Better VCR-Like Interactions in a Periodic Broadcast Environment. In *Proceedings of the IEEE GLOBECOM Global Communications Conference*, pages 1648–1652, 2002.

[71] W. Tavanapong, M. Tran, J. Zhou, and S. Krishnamohan. Video Caching Network for On-Demand Video Streaming. In *Proceedings of the IEEE GLOBECOM Global Communications Conference*, pages 1723–1727, 2002.

[72] TiVo Inc. http://www.tivo.com.

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

[74] M. Waldvogel and R. Janakiraman. Efficient Media-on-demand over multiple Multicast groups. In *Proceedings of the IEEE GLOBECOM Global Communications Conference*, pages 1662–1666, 2001.

[75] B. Wang, S. Sen, M. Adler, and D. Towsley. Optimal Proxy Cache Allocation for Efficient Streaming Media Distribution. In *Proceedings of the IEEE INFOCOM Conference on Computer Communications*, pages 1726–1735, 2002.

[76] E. W. M. Wong and S. Chan. Modeling of Video-on-Demand Networks with Server Selection. In *Proceedings of the IEEE GLOBECOM Global Communications Conference*, pages 54–59, 1998.

[77] Y. Zhao, D. Eager, and M. K. Vernon. Scalable On-Demand Streaming of Non-Linear Media. In *Proceedings of the IEEE INFOCOM Conference on Computer Communications*, pages 1522–1533, 2004.

[78] Y. Zhao, D. L. Eager, and M. K. Vernon. Network Bandwidth Requirements for Scalable On-Demand Streaming. In *Proceedings of the IEEE INFOCOM Conference on Computer Communications*, pages 1119–1128, 2002.

VITA

Tammy VanDeGrift was born and raised in Silverton, Oregon. She graduated from Gustavus Adolphus College (Minnesota) in 1999 with a Bachelor of Arts in honors computer science and honors mathematics. She earned a Masters degree in Computer Science & Engineering from the University of Washington in 2001 with a thesis in the area of computational biology. In 2005 she graduated with a Doctor of Philosophy degree in Computer Science & Engineering from the University of Washington. Her research interests are broad across the discipline, including projects in computational biology, image processing, computer science education, educational technology, human computer interaction, and media-on-demand systems.