Scheduling Techniques for Packet Routing, Load Balancing and Disk Scheduling

by

Matthew Andrews

B.A. University of Oxford, England (1993)

Submitted to the Department of Mathematics
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1997

Author .....
Department of Mathematics
May 2, 1997

Certified by .....
Michel X. Goemans
Associate Professor of Applied Mathematics
Thesis Supervisor

Accepted by .....
Hung Cheng
Chairman, Applied Mathematics Committee

Accepted by .....
Richard Melrose
Chairman, Departmental Committee on Graduate Students

Scheduling Techniques for Packet Routing, Load Balancing and Disk Scheduling

by

Matthew Andrews

Submitted to the Department of Mathematics
on May 2, 1997, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

We consider three scheduling problems that arise in studies of packet routing, load balancing and disk scheduling.

A fundamental problem in the design of packet-switched communication networks is to provide effective methods for resolving contention when many packets wish to cross a link. End-to-end packet delays should be low and queue sizes should be small. For an adversarial connectionless model we provide upper and lower bounds on delay for many simple algorithms. For an adversarial session-oriented model we prove the existence of an asymptotically optimal schedule with per-packet delay guarantees of $O(\text{distance} + 1/\text{session rate})$ and constant queue size. We also describe randomized schedules with near-optimal bounds.

In the on-line load balancing problem, jobs arrive on-line and must be assigned to one of a set of machines, thereby increasing the load on that machine by a certain weight. Jobs also depart on-line. The goal is to minimize both the maximum load on a machine and the amount of job reassignment that occurs. For the cases of identical machines and related machines we consider arbitrary reassignment costs and provide the first algorithms that have constant competitive ratios against current load and constant reassignment factors.

In the disk scheduling problem we have a set of read and write requests on a computer disk and a convex reachability function that determines how fast the disk head travels between tracks. Our aim is to schedule the head so that it services all the requests in the shortest time possible. Among other things we present a 3/2-approximation algorithm (with a constant additive term) for the general case and an optimal polynomial-time solution for the special case in which the reachability function is linear. We also present a heuristic for the on-line problem in which requests arrive over time.

Thesis Supervisor: Michel X. Goemans
Title: Associate Professor of Applied Mathematics

Acknowledgments

I would first like to thank Professors Michel Goemans, Tom Leighton and David Karger for serving on my thesis committee. Michel's wonderful class, "Advanced Algorithms", convinced me that I wanted to devote my time in graduate school to the study of algorithms. As an advisor, he has been an excellent source of ideas and encouragement. Without him, I would have found it much more difficult to get started on research. I am by no means the first to mention Tom's powers of intuition but I feel that I must add my words of astonishment here. He never fails to come up with new ways of looking at problems and a meeting with him always left me optimistic. David has been a source of intriguing problems and he has given me a great deal of advice on the presentation of this thesis.

I also wish to thank Dr. Glenys Luke, my undergraduate tutor at St. Hugh's College, Oxford. Her infectious enthusiasm gave me a real sense of the enjoyment that mathematics offers.

During my time at MIT I have worked with many people, including Michael Bender, Antonio Fernández, Michel Goemans, Mor Harchol-Balter, David Karger, Jon Kleinberg, Tom Leighton, Takis Metaxas and Lisa Zhang. I feel fortunate to have had the chance to do research with them. Collaborative work is much more rewarding than solitary study. I am also grateful to have had the opportunity to work with Bill Aiello, Sandeep Bhatt, K. R. Krishnan and Kalyan Perumalla at Bellcore during the summer of 1996.

I would like to thank the Kennedy Memorial Trust for their financial support during my first year at MIT. The Kennedy scholarships provide their recipients with a superb opportunity to make their time in America a memorable cultural experience as well as an academic one. Anna Mason, the secretary of the Trust, deserves much of the credit for this. I would also like to thank Michel for providing me with a Research Assistantship for two semesters and a summer, and the MIT Department of Mathematics for providing me with a Fellowship for one semester.

Lisa Zhang has been a special part of my life in graduate school. Most of my

work here has been joint with her and she has been a wonderful friend to have in a new country. Her diligence, insight and extraordinary ability to handle complicated calculations make her the ideal coauthor. Her love, warmth and consideration have enriched my time at MIT immeasurably.

My final words of thanks are reserved for my family. They have constantly provided love and support in whatever I have chosen to do and they have always helped me to achieve my goals. In particular, I would like to thank them for encouraging me during my time in the US, even though I am sure that my presence on the other side of the Atlantic is not easy for them. I hope that I have given them reason to be proud of me.

Contents

List of Figures

Chapter 1

Introduction

Scheduling problems arise whenever objects or processes have to compete for available resources. Computer science is a rich source of such problems since processors may be able to perform many different kinds of tasks but are only able to perform one of them at any point in time. The field of networking also raises many scheduling issues since a network may have to handle many different classes and types of traffic but each link can only handle a certain number of connections at a time.

Classical scheduling problems usually assume an abstract set of *machines* and a set of *jobs* that need to be processed. Each job is known to consume a certain amount of resources and each machine is known to have a certain ability to process jobs. Typical performance measures are the *makespan*, which is the total time that is needed to process all the jobs and the *response time* of a job which is the time between its arrival into the system and its completion. For an overview of various different types of scheduling problems see [42]. In this thesis we shall consider three specific scheduling problems, namely contention resolution in packet switched networks, on-line load balancing, and disk scheduling. For each problem there will be some resource, such as a communication network or a computer disk drive, that we wish to use as efficiently as possible.

1.1 Packet Routing

In Part 1 we consider scheduling issues that arise in packet-switched networks. Future communication networks such as Broadband Integrated Services Digital Networks (B-ISDN) will be required to carry many different types of traffic. This is in marked contrast to traditional networks such as the telephone network which was only required to provide one type of connection, namely voice which is transmitted at 64 kilobits per second. The modern Internet is theoretically capable of carrying any digital traffic since it can transmit packets of differing sizes. A drawback of the Internet however is that it sends packets in a *best effort* manner, i.e. a packet is only transmitted when the network has sufficient resources. This means that when the network is heavily loaded all users will suffer regardless of the type of communication that they are trying to accomplish. It is a common belief in Europe that one should aim to use the Internet in the morning before the East Coast of the United States "wakes up".

It would therefore be desirable to have a network that could support different traffic classes and provide *Quality-of-Service* (QoS) performance guarantees for the connections. Moreover, different traffic classes should be able to request different QoS guarantees. The most commonly cited areas in which a connection might require QoS bounds are bandwidth, delay, delay jitter and loss. (See Keshav's book [38].)

The bandwidth of a connection is a measure of the number of bits per second that may need to be transmitted along it. The delay on a connection measures the time between the transmission of a bit from its source and its arrival at its destination. Delay jitter is a measure of the variance of delays experienced by bits. Loss is a measure of how many bits are lost during transmission.

We now give examples of potential traffic classes and their QoS requirements.

- Video Conferencing. Here we assume that two or more people are communicating in two directions via video streams. Bandwidth requirements are big since a picture involves a lot of data. The delay must be small otherwise meaningful two-way communication is not possible. Delay jitter must also be small oth-

erwise the pictures would appear to be "jerky". However if certain parts of a video frame are missing then it is still possible that the video stream is coherent and so some data loss is tolerable.

- Video-on-Demand. Here we assume that a viewer would like to watch a video that is provided by some service. As before the bandwidth must be big, the delay jitter must be small and the loss can be reasonably large. However, since the communication is in one direction only the delay can be high.

- World Wide Web browsing. When we request a web page we would like it to appear fast and we would like it to be accurate. Hence delay and loss must both be low. However, delay jitter is not important since we only care about the time to receive the entire page. The bandwidth required is dependent on the complexity of the page.

- Voice conversation. This application has similar requirements to video conferencing except that the bandwidth needed is much lower.

- File transfer. The only essential requirement here is that no data is lost. High bandwidth and low delay are also desirable but not critical.

The Asynchronous Transfer Mode (ATM) is the method that has been proposed for solving these problems. In ATM the traffic offered to a network is encoded as streams of 53 byte packets.[1] For each connection a path is set up on the network and packets are sent along this path. The crucial problem that we consider in this thesis is how to multiplex several connections onto one link. That is, if packets are arriving along several different connections and all wish to cross the same link, in what order should they be sent. It is important to be able to do this in such a way that the quality of the connection is guaranteed. The fact that ATM can support QoS means that ATM networks are likely to become ubiquitous in the not too distant future.

In this thesis we mostly concern ourselves with the question of bounding the delay experienced by data. We assume that the data has been packetized and that

[1] In the ATM literature the word "cell" is sometimes used instead of "packet".

all packets are the same size (as in ATM). We also assume that all packets take the same amount of time to cross an edge. We shall use this as our unit of time.

Our model of packet routing is a *store-and-forward* model. This means that at each time step at most one packet may cross a link. If two or more packets wish to cross the link at the same time then we assume that the others *queue up* at that edge. Our goal is to bound the total delay experienced by packets as they travel from their sources to their destinations. A secondary (and closely related) goal is to bound the size of the queues at the edges.

Before it is possible to analyze these packet-switched networks we must make some assumptions about the way in which packets are *injected* into the network and the manner in which they traverse the edges. Different communities of researchers have approached this issue in different ways. For example, there has been a great deal of work in the queueing theory community on networks where the injections are generated by Poisson processes and the time for a packet to cross an edge is exponentially distributed (see, for example, [37] and [39]). These assumptions are convenient in that the evolution of the system is memoryless.

Packet routing problems have also been the subject of a large body of work within the field of Theoretical Computer Science. For a comprehensive survey see Leighton's book [43]. Much of this work was initially motivated by the study of parallel computation. In particular it was a key part of the work that linked the study of idealized machines such as the PRAM to machines that could be built in practice. The usual assumption here is that the packets take unit time to cross a link. However, much of this work is not immediately applicable to communication networks since it assumes a static model, i.e. all the packets are present in the network initially.

More recent work within the Theoretical Computer Science community has considered the dynamic packet routing problem. This work usually assumes that the packets are generated by some stochastic process and are routed to random destinations [47, 34, 14, 13]. Another question that has attracted interest is whether or not it is possible to use results from queueing theory concerning exponential edge-traversal times to derive results in networks where the edge-traversal times are con-

stant [63, 29, 30, 49].

We consider a model that does not make any statistical assumptions, i.e. we assume that the packets are injected into the network by an adversary. By this we mean that both the times at which packets are injected *and* the routes that they must follow are chosen by an adversary. However we cannot allow the adversary to inject an unlimited number of packets into the network otherwise it would be impossible to provide delay bounds for packets. We consider two different methods for restricting the adversary, and we refer to the two resulting models as the *connectionless model* and the *connection-oriented* or *session-oriented* model.

In the connectionless model we assume that the adversary can choose to inject packets along arbitrary paths. The only restriction we make is that there are two constants $r < 1$ and σ such that for any edge e and for any interval of t time steps, no more than $rt + \sigma$ packets may be injected into the network that wish to cross edge e. Clearly a restriction such as this is necessary otherwise the number of packets that wish to cross edge e could grow without limit. This would lead to unbounded delay. For the connectionless model we focus on several simple *contention-resolution* protocols. These are protocols that determine which packet should cross an edge when more than one wish to do so. In this thesis we show (a) that the contention-resolution protocols Farthest-to-Go and Nearest-to-Source have bounded delay and queue size and (b) that the delays and queue sizes for Farthest-to-Go, Nearest-to-Source and Shortest-in-System can be exponential in the size of the network. The exact definitions of these protocols are given in Chapter 2. These results together with other results and analysis from [1] almost completely characterize several simple protocols in terms of packet delay.

In the session-oriented model we assume that packets can only be injected along one of a fixed set of paths in the network. We refer to these paths together with the packets that follow them as *sessions*. Each session i has an associated injection rate r_i. We restrict the adversary by saying that in any interval of t time steps no more than $r_i t + 1$ packets may be injected into session i. As with the connectionless model we must bound the rate at which packets that wish to cross a specific edge

are injected. In the session-oriented model we achieve this by requiring that for each edge e the sum of the rates of all sessions that pass through edge e is at most $1 - \varepsilon$ for some constant ε.

Our first result in the session-oriented model is a simple, randomized, distributed schedule that with high probability allows all session i packets to reach their destination in time $O(1/r_i + d_i \log m/r_{\min})$ where m is the number of edges in the network and $r_{\min}$ is the minimum rate of a session. Except for initial queues (which we define in Chapter 2), the queues have size at most $O(\log m/r_{\min})$. We then show the existence of a schedule that allows session i packets to reach their destinations in time $O(1/r_i + d_i)$ which is asymptotically optimal. All queues (except for initial queues) have size $O(1)$. These bounds contrast strongly with previous bounds which are either multiplicative (i.e. given in terms of d_i/r_i) or else are not session based (i.e. the bounds are at least $1/r_{\min} + d_{\max}$ where $r_{\min}$ is the minimum rate of a session and $d_{\max}$ is the maximum length of a session.)

The work on the connectionless model is joint with Baruch Awerbuch, Antonio Fernández, Jon Kleinberg, Tom Leighton and Zhiyong Liu and is contained in Chapter 3. The work on the session-oriented model is joint with Antonio Fernández, Mor Harchol-Balter, Tom Leighton and Lisa Zhang and is contained in Chapters 4 and 5.

1.2 Load Balancing

Part 2 of this thesis is devoted to the on-line load balancing problem. Consider a system of machines and a set of jobs that arrive in and depart from the system. Whenever a job arrives it must immediately be assigned to a machine for servicing. Our goal is to assign the jobs in such a way that the load on any machine is not too high, i.e. we should like the jobs to be evenly *balanced* on the machines. There are many variants of this problem. For instance, machines may have different *capacities* for servicing jobs or the jobs may have different requirements. In addition, there are a number of ways to measure the performance of candidate load balancing algorithms.

Load balancing problems arise naturally in settings in which jobs have to compete

for the available resources. Machines may represent various kinds of communication channels with certain bandwidth, and jobs requests for bandwidth [2, 4, 5, 6, 7, 69]. Alternatively, machines may represent distributed database platforms, and the jobs may be application programs accessing the database [69]; here, the increase in the load represents the time for an access. The on-line load balancing problem has been much studied in connection with the virtual circuit routing problem [2, 4, 69]. In this problem, the machines are the edges of a network, the jobs are requests for allocating a certain amount of capacity between two given endpoints, and the goal is to minimize the maximum congestion.

As usual, the performance of an on-line algorithm can be measured using the notion of competitive analysis [61]. In the context of load balancing, most analyses that have been performed compared the maximum load at time t of the on-line algorithm to the maximum load at any time between 0 and t, the *peak load*, of the best off-line algorithm presented with the same sequence of job arrivals and departures (see [4, 5, 6]). Westbrook [69], however, pointed out that a competitive analysis against peak load is somewhat unrealistic. Westbrook instead proposed the more realistic notion of competitiveness against *current load.* An on-line algorithm is said to be α-competitive against current load if, for any time t, the maximum load at time t is at most α times the lowest achievable load at time t for the jobs then in the system. If the model does not allow for job departures, as in [2, 7], there is no difference between current load and peak load. However, in general, competitiveness against current load is a much stronger notion than competitiveness against peak load. Even though the arrival or departure of a job may greatly affect the optimum assignment of jobs to machines, a competitive algorithm against current load needs to ensure that its maximum load before and after the arrival or departure is still within a factor of α of the corresponding optimum. Phillips and Westbrook [55] were the first to present competitive analyses against current load, but the distinction between peak and current load was highlighted in Westbrook [69].

To see why it is useful to have an algorithm competitive against current load, consider the example in which the machines are communication channels and the

jobs are requests for bandwidth. Suppose also that the data being transmitted comes from a real-time application such as audio or video. If many jobs arrive then the system will inevitably become overloaded. Data must be lost and so the video will be "jerky" and the audio will sound "broken up". Suppose however that later on there are many fewer jobs present. An algorithm that is competitive against peak load would not have to take advantage of this and so the quality of transmission might still be poor. In contrast, an algorithm that is competitive against current load would have to take advantage of the lightly loaded system and produce low load on each channel. Hence the video would be smooth and the audio would have good sound quality.

The notion of competitiveness against current load is too strong if we do not allow reassignments of jobs (also called job preemptions). Indeed, without reassignments, any algorithm is m-competitive against current load and no algorithm is better than m-competitive against current load, where m is the number of machines [69]. On the other hand, if we allow an arbitrary amount of reassignment then the problem loses its on-line aspect: the scheduler can "simply" reconfigure all the machines at any departure or arrival to match (or closely match) the optimum configuration (although this is an NP-hard problem). There is thus a trade-off between the competitive ratio and the amount of reassignment performed.

To be more specific, each job has an associated *reassignment cost* which is a measure of how much expense is involved in assigning the job to a machine. Whenever a job is assigned or reassigned to a machine we must pay the reassignment cost. An on-line load balancing algorithm has a reassignment factor of r if the total reassignment cost paid is always at most rS, where S is the total reassignment cost of all jobs that have arrived in the system. Intuitively, this means that in an amortized and weighted sense, each job is assigned to a machine at most r times.

We shall be considering two specific versions of the load balancing problem. They differ in the way that the load on a machine is defined. We assume that when a job arrives in the system we are told its *weight*. This is a measure of the amount of service that the job needs. In the *identical machines* problem, the load on any machine is

simply the sum of the weights of the jobs that have been assigned to it. In the *related machines* problem, each machine i has an associated capacity, cap_i that represents its ability to service jobs. The load on machine i is now the sum of the weights of jobs assigned to machine i divided by cap_i.

The main contribution of this thesis is to present the first algorithms with constant competitive ratios and constant reassignment factors for the general case in which the job reassignment costs are *unrelated* to the job weights. Most previous work concentrated on the case in which the reassignment costs are proportional to the job weights. In particular we present,

- An algorithm for the identical machines problem that is 3.5981-competitive against current load and has a reassignment factor of 6.8285.
- An algorithm for the related machines problem that is 32-competitive against current load and has a reassignment factor of 72.5.

For related machines we also present an algorithm whose competitive ratio is logarithmic in the ratio of the largest machine capacity to the smallest machine capacity. This algorithm is superior to the algorithm with constant bounds unless the capacities of the machines differ greatly. For identical machines we consider the special case of the problem in which the reassignment costs are unit and the special case in which the reassignment costs are proportional to the job weights. For these restricted problems we provide algorithms whose bounds are lower than those previously known.

The results for the identical machines problem are contained in Chapter 7 and the results for the related machines problem are contained in Chapter 8. Both chapters represent joint work with Michel Goemans and Lisa Zhang.

1.3 Disk Scheduling

In Part 3 of the thesis we consider the disk scheduling problem. Computer processor speed and disk and memory capacity are increasing by over 40% per year. In contrast, disk speed is increasing more gradually, growing by only 7% per year [59]. Since this

rate is unlikely to change substantially in the near future, I/O performance may become the bottleneck in most computer systems. However, despite the difficulty of improving mechanical components, we can still aim to *use* the disks more efficiently.

For example, disks generally operate at a small fraction of their maximum bandwidth. Experiments have shown that sophisticated disk head scheduling algorithms can deliver higher throughput [60, 33, 71]. This past research has focused almost exclusively on two types of work loads: synthetic work loads, where disk requests are randomly and uniformly distributed across the disk, and more recently, traces, where the requests to an actual disk are recorded and used to test algorithms. However, for these or for general work loads, researchers have made little attempt to develop algorithms with provable performance *guarantees.* In addition, no one has determined the computational complexity of the disk scheduling problem. There is a risk that synthetic work loads and traces from a few environments may not represent all possible situations.

In this thesis we propose several disk-scheduling algorithms with performance guarantees and we state a hardness result. The research has provided additional payoffs. The first, of practical interest, is a heuristic for the on-line problem. The second payoff is of theoretical interest: the disk problem suggests algorithms for a special case of the asymmetric traveling salesman problem with the triangle inequality (ATSP-Δ). Before defining our problem formally we describe the structure of a modern disk.

The Disk A computer disk is composed of several concentric, rapidly-rotating *platters*, where data may be written to both sides of each platter. Platters are logically divided into circular *tracks.* A *cylinder* is composed of all the circular tracks having the same radius. The smallest unit that can be written to disk is called a *sector*, which typically holds 512 bytes of data. Modern disks have approximately 2000 cylinders and 100 sectors per track. The data is transferred to and from the disk by a set of read/write heads (usually one per surface). The disk arm moves the heads in concert, so that all of the heads are contained in one cylinder. For this reason we can restrict

our attention to one disk platter and one disk head.

When a head accesses a particular sector, it suffers two kinds of delays. The *seek time* is the time required to move the head to the correct track; the *rotational latency* is the time necessary, once the head is in the correct track, for the requested sector to pass underneath the head. Modern disks rotate at a speed of 3600-7200 rpm (implying that one rotation takes 8-16 msec). With today's technology, the time for a *track-to-track* seek (one track to a neighboring track) is typically 1 msec; the time for a *full-seek* (the innermost to the outermost track) is typically 20 msec. Small seeks are dominated by a constant start-up time, medium-length seeks by a period of acceleration and deceleration, and long seeks by a period of constant speed. In the following table we give the specifications from [59] for the Hewlett-Packard 97560 disk.

sector size (bytes)	512
number of cylinders	1962
tracks per cylinder	19
data sectors per track	72
revolution speed (rpm)	4002
seek time (msec) for d tracks	
$d \leq 383$	$3.24 + 0.400\sqrt{d}$
$d > 383$	$8.00 + 0.008d$

The Problem In this paper we chiefly consider the off-line version of the disk scheduling problem. The input consists of a set of points on the disk (which we call *requests*) and a convex *reachability function* which determines how long it takes the disk head to move between tracks. Our goal is to schedule the disk head so that it services (i.e. visits) all of the requests in the shortest possible time. Note that if we consider the motion of the head relative to the disk then the problem becomes a special case of the Traveling Salesman Problem. We also consider an on-line version of the disk scheduling problem in which the requests arrive over time and are placed into a queue. The head is able to service any request that is currently in the queue.

Our goal is to maximize the throughput.

Our Results We first present an algorithm, HEADSCHEDULE, for the off-line problem; HEADSCHEDULE services all the requests in at most $\frac{3}{2}T_{\text{opt}} + a$ rotations, where T_{opt} is the number of rotations taken by an optimal algorithm and a is a term that depends solely on the reachability function. We can show that this problem is NP-hard and so in general it is not practical to look for the optimum algorithm. For the special case in which the reachability function is linear we present a polynomial-time algorithm, MONOTONE, and show that it is optimal. The algorithms HEADSCHEDULE and MONOTONE are presented in Sections 9.2 and 9.4 respectively. The NP-hardness result is contained in Section 9.3.

In Section 9.5 we relate the disk scheduling problem to the special case of the asymmetric Traveling Salesman Problem with the triangle inequality (ATSP-Δ) in which all distances are either 0 or α for some value $\alpha > 0$. We show how to find the optimal tour in polynomial time and describe how this gives another approximation algorithm for the disk scheduling problem.

In Section 9.6 we consider the on-line problem. Since the requests in real systems are known to arrive in a "bursty" fashion [58] our off-line algorithms are still useful. When a burst of requests arrive we can schedule them using an off-line algorithm. We also present an on-line algorithm, CHAIN, which is related to our algorithm for the above ATSP-Δ. Although we are unable to provide performance guarantees for this algorithm it has better look-ahead properties than algorithms that have been considered previously.

This work on disk scheduling is joint with Michael Bender and Lisa Zhang.

Part I

Packet Routing

Chapter 2

Packet Routing-Introduction

2.1 The Problem

In this section we study the problem of scheduling packets in a packet-switched network. We model the network by a directed graph in which the nodes represent the switching hardware and the edges represent the communication links. From now on we shall use the words node and switch interchangeably and we shall use the words edge and link interchangeably. When a packet that wishes to travel between two nodes arrives in the network we say that it has been *injected.* We shall assume that when a packet is injected the route that it must follow has already been determined, i.e. we shall assume that the problem of finding good routes for the packets has already been solved. Throughout this work we shall consider a *store-and-forward model* of packet transmission, i.e. we assume that packets take one time step to cross an edge and at most one packet can cross an edge at each time step. If two or more packets wish to cross an edge then one of them is transmitted and the rest queue up at the switch.

In this work we focus on three key questions. First, how do we ensure *stability* in the network, i.e. how do we make sure that the number of packets in the network remains bounded. Second, how do we reduce the *delay* experienced by packets as they travel through the network. We should like to be sure that packets travel from their sources to their destinations as quickly as possible. This is an important Quality-

of-Service (QoS) requirement for real-time applications such as video conferencing. The third question comes from the fact that packets that queue up at nodes must be stored in a buffer. It is desirable for these buffers to be small but we also need to be sure that we never lose packets. Hence we should like to know how to keep the number of packets that are ever queued up at a node small. We shall refer to this issue as *limiting the queue size.* Stability, packet delay and queue size are all closely related. (See Theorem 3.1.3.)

Unless we make some assumptions about the way in which packets are injected into the network then there will be no meaningful answers to these questions. If packets can be injected at an unbounded rate then it is possible that the network will become unstable, the delay experienced by packets will be unbounded and the queue sizes will be unbounded. We shall be considering two distinct models of packet injection, a connectionless model and a connection-oriented or session-oriented model. In both models we assume that the injections are made by an adversary and hence we shall be concentrating on worst-case scenarios. However, in the session-oriented model we impose more restrictions on the adversary.

2.2 The Connectionless Model

In the connectionless model when a packet is injected it may follow any *simple* path through the network. (A simple path does not cross any edge more than once.) We restrict the number of packets that may be injected into the network in the following manner. For some constants σ and r, $\sigma > 0$ and $0 < r < 1$, we specify that for any edge e in the network and any time interval of t steps, no more than $\sigma + rt$ packets that wish to cross edge e may be injected into the network during this interval. We refer to σ as the burst parameter and r as the rate. In some sense this is the weakest restriction that could be made that would still allow any hope of network stability. This is because if packets that wish to cross a particular edge are injected at a rate higher than one over a long period of time then packets will continue to accumulate in the network since at most one packet can cross the edge during any time step. We

make no further assumptions regarding the packet injections and so an adversary is allowed to determine when packets are injected and which paths they should follow as long as the above requirement is satisfied.

Our goal regarding the connectionless model is to analyze several simple protocols for deciding which packet should cross an edge when more than one wishes to do so. We should like to consider whether or not they lead to network stability and we should also like to provide bounds on the total delay experienced by packets and the maximum queue sizes that can occur. We shall consider the following nine protocols. Each of them is distributed in the sense that each switch decides which packet to send solely by examining the queue of candidate packets that wish to cross the edge at the next step. For some of the protocols the packets may be required to carry some timing or route information that is used in the decision. The protocols are:-

- FIFO (First-in-First-out) - The packet chosen to cross the edge is the one that has been in the queue for that edge for the longest amount of time.
- LIFO (Last-in-First-out) - The packet chosen is the one that has been in the queue for the shortest amount of time.
- SIS (Shortest-in-System) - The packet chosen is the one that was injected into the system most recently.
- LIS (Longest-in-System) - The packet chosen is the one that has been in the system the longest.
- NTG (Nearest-to-Go) - The packet chosen is the one that has to cross the fewest edges to reach its destination.
- FTG (Farthest-to-Go) - The packet chosen is the one that has to cross the most edges to reach its destination.
- NTS (Nearest-to-Source) - The packet chosen is the one that has crossed the fewest edges since its injection.

- FFS (Farthest-from-Source) - The packet chosen is the one that has crossed the most edges since its injection.

- RANDOM - A randomized protocol that is similar to LIS.

All of the above protocols are *greedy.* A greedy protocol will always send a packet whenever the queue is non-empty.[1]

2.2.1 Results – Connectionless Model

In Chapter 3 we prove the following results.

1. The protocol FTG is stable. This means that for all networks and all adversaries that obey the restrictions described above, the number of packets in the system can be bounded if the protocol FTG is used. In particular, if $\varepsilon = 1 - r$, m is the number of edges in the network and d is the maximum length of a path that a packet must follow, then the total number of packets in the system is never more than $O(\frac{m^d \sigma}{\varepsilon})$. In addition no packet takes more than $O(\frac{m^{d-1}\sigma}{\varepsilon})$ time steps to travel from its source to its destination and the maximum queue size that ever occurs is $O(\frac{m^{d-1}\sigma}{\varepsilon})$.

2. The protocol NTS is stable. This result is a corollary to the result for FTG. The bounds on number of packets in the system, the packet delay and the maximum queue size are identical. The protocol NTS is of interest because it is extremely easy to implement. Protocols such as LIS and SIS require the switches to have an accurate clock which is expensive in practice. The protocol FTG requires either the switches or the packets to carry information about the length of the routes that the packets will follow. With many schemes for determining the routes of packets this assumption is not satisfied. All that is required for the protocol NTS however is that the packets keep track of the number of links that they have crossed.

[1]The term *work conserving* is sometimes used instead of greedy.

3. The protocols FTG, SIS and NTS may produce exponentially large queues, i.e. there exists a graph with m edges and an adversary with rate strictly less than one such that using FTG, SIS or NTS results in a system in which there are queues of size $2^{\Omega(d)}$. An immediate corollary is that for these protocols the packet delay can be exponential.

2.2.2 Classification of Protocols

The above results combined with results of Andrews, Awerbuch, Fernández, Kleinberg, Leighton and Liu [1] enable us to determine almost completely the stability, packet delay and queue size of the nine protocols described earlier. We assume as above that the graph has m edges, the maximum length of a path that a packet must follow is d and the adversary has burst size σ and rate r, $0 < r < 1$. In the following table we indicate the size of the upper and lower bounds on packet delay for the various protocols. The exact bounds will be given in Chapter 3. We use EXP to denote an expression in terms of m, d and $1/\varepsilon$ that contains d in the exponent. We use POLY to denote an expression that is polynomial in m, d and $1/\varepsilon$.

	Stable?	Packet Delay	
		Upper Bound	Lower Bound
FIFO	No	∞	∞
LIFO	No	∞	∞
SIS	Yes	EXP	EXP
LIS	Yes	EXP	POLY
NTG	No	∞	∞
FTG	Yes	EXP	EXP
NTS	Yes	EXP	EXP
FFS	No	∞	∞
RANDOM	Yes	POLY	POLY

By Theorem 3.1.3 in Chapter 3, exponential bounds on packet delay imply exponential bounds on queue size and the number of packets in the network. Likewise,

polynomial bounds on packet delay imply polynomial bounds on these quantities. It should be emphasized that the lower bounds in the table are existential. For example, a lower bound of ∞ means that there exists a network and an adversary of rate $r < 1$ such that the maximum delay experienced by packets is unbounded.

2.2.3 Previous Work – Connectionless Model

The model that we are calling the connectionless model was first introduced by Borodin, Kleinberg, Raghavan, Sudan and Williamson in [11]. Their name for the study of this model was "Adversarial Queueing Theory". The adversaries considered by Borodin *et al.* were weaker than those consider here in that they were characterized solely by a rate parameter r. The number of packets injected during t time steps that wished to cross an edge e could not be greater than $\lceil rt \rceil$ for any value of t. The main results of [11] are,

- All greedy protocols are stable on directed acyclic graphs for adversaries of rate $r \leq 1$.
- All greedy protocols are stable on the ring (directed cycle) for adversaries of rate $r \leq \frac{1}{2} - \varepsilon$.
- FTG is stable on the ring for adversaries of rate $r \leq 1$, and LIS is stable on the ring for adversaries of rate $r \leq 1 - \varepsilon$.
- FIFO and LIS can be unstable on the ring for adversaries of rate $r = 1$.

The results of [11] were consistent with both of the following extreme scenarios. Maybe all greedy protocols are stable in all networks for all adversaries of rate $r \leq 1-\varepsilon$ or maybe no protocols are stable in this general setting. In [1], Andrews, Awerbuch, Fernández, Kleinberg, Leighton and Liu showed that neither of these possibilities are true. This has already been indicated in Section 2.2.2. The adversaries of [1] are stronger than those of Borodin *et al.* since they allow for some burstiness. Andrews *et al.* say that an adversary has rate (w, r) if at most rw packets that wish to cross

an edge e are injected during any interval of length w. Here w is some constant. If w is large then bursts of packets that all wish to cross a single edge are allowed.

Recall that in this thesis we are restricting the adversary by requiring that for some constants σ and r, no more than $\sigma + rt$ packets can be injected in t steps that wish to cross an edge e. Although this is not strictly equivalent to the definition in [1] the results in the two models are virtually identical. Most of the results from [1] have been described above and they will be considered in more detail in Chapter 3. The major results in [1] that we shall not consider later are,

- All greedy protocols are stable on the ring for adversaries of rate $r \leq 1 - \varepsilon$.
- There is a polynomial time algorithm that decides, given a network, whether or not all greedy protocols are stable on the network for all adversaries of rate $r \leq 1 - \varepsilon$.

2.3 The Session-Oriented Model

In the session-oriented or connection-oriented model we again assume that the packet injections are made by an adversary. However, we now restrict the adversary further by requiring that whenever a packet is injected it must follow one of a fixed set of paths through the network. We refer to one of these paths together with the packets that are injected along it as a *session*. With each session i we associate an *injection rate* r_i that specifies how many packets may be injected along it. We require that in t time steps no more than $r_i t + 1$ packets are injected into the system that wish to travel along session i. As with the connectionless model there must be some restriction on the injection rates otherwise the system could become unstable. In the session-oriented model the restriction we impose is the following. There exists some constant ε such that the sum of the rates of sessions that pass through any edge e add up to at most $1 - \varepsilon$ for some constant ε.

In the session-oriented model we consider two different types of queues at the switches. When a packet has been injected and is waiting to cross its first link we

assume that it is held in an *initial queue*. Once the packet has started to move it is held in *edge queues* at the switches. When we consider the issue of queue size we focus on bounding the size of the edge queues. The reason for this is that the initial queues can be external to the network and can consist of cheap, slow memory. The edge queues, in contrast, are part of the network and usually consist of fast and expensive memory.

The key difference between the connectionless model and the session-oriented model is the following. In the connectionless model we can have two distinct paths that pass through a common edge and an adversary that oscillates between injecting packets at a rate close to 1 on one of the paths and injecting packets at a rate close to 1 on the other one. This is not possible in the session-oriented model since the rates of the sessions associated with these paths would sum up to a value greater than 1.

The session-oriented model describes a scenario in which the connections are permanent or have long duration. In contrast the connectionless model describes situations where packets are routed individually (e.g. the Internet) and situations where temporary connections of short duration are set up.

2.3.1 Results – Session-Oriented Model

Since we have established that the session model is more restrictive in terms of allowable adversaries we should expect that it allows us to prove stronger results. This is indeed true. Our main result is a proof that there exists a way to schedule the packets such that packets in session i reach their destinations in time $O(1/r_i + d_i)$. An existential lower bound is $\Omega(1/r_i + d_i)$ and so our result is optimal up to constant factors. In addition the edge queues have constant size.

The novel feature of our result is that the guarantee is both additive and session-based. By session-based we mean that the bound for session i is given in terms of r_i and d_i. Previous results were either given in terms of $r_{\min} = \min_i r_i$ and $d_{\max} = \max_i d_i$ or else were given in terms of d_i/r_i.

The main drawback of our result is that it uses the Lovász Local Lemma and

hence is non-constructive.[2] However the schedule will be periodic and so we could in principle calculate the schedule for one period off-line and then run it repeatedly. In order to show a simpler result and also to remove the reliance on the Lovász Local Lemma we describe randomized algorithms that with high probability schedule session-i packets so that they reach their destinations in time $O(1/r_i + d_i \log m/r_{\min})$. (Recall that m is the number of edges in the network.) We initially show how to do this in a centralized fashion but we then describe how it can be done with a distributed algorithm. For the centralized version the edge queues have size $O(\log m/r_{\min})$.

In a common extension of this model the traffic is assumed to be *leaky-bucket constrained* with parameters b_i and r_i. This means that no more than $b_i + r_i t$ packets are injected into session i in any time interval of length t. Our results generalize easily to this model. The first delay bound and the existential lower bound increase to $O(b_i/r_i + d_i)$. The second delay bound increases to $O(b_i/r_i + d_i \log(m/r_{\min}))$.

In Chapter 4 we present the randomized algorithms that have delay bound $O(1/r_i + d_i \log m/r_{\min})$. We prove the existence of a schedule with delay bound $O(1/r_i + d_i)$ in Chapter 5.

2.3.2 Previous Work – Session-Oriented Model

There has been a great deal of work on the problem of providing delay bounds for leaky bucket constrained traffic. One of the most widely studied algorithms was first proposed by Demers, Keshav and Shenker [19] and is called Weighted Fair Queueing (WFQ). It is based on an idealized algorithm known as Generalized Processor Sharing (GPS). GPS assumes a fluid model in which fractions of packets from different sessions can be serviced simultaneously. In particular, if packets from sessions $I = \{i_1, i_2, \ldots, i_n\}$ are queued at a switch then packets from session i_k are continuously serviced at a rate of,

$$\frac{\Phi_{i_k}}{\sum_{j \in I} \Phi_{i_j}},$$

[2]The result in this thesis is non-constructive. However, Zhang has shown that our methods can be made constructive [74]. The techniques used are based on ideas of Leighton, Maggs and Richa [45].

per time step for some parameters Φ_{i_j}. Parekh and Gallager [53, 54] showed that if we set $\Phi_i = r_i$ then GPS has excellent delay properties. It is of course unimplementable since in practice only whole packets can be scheduled. The idea of WFQ is to approximate GPS. Under WFQ each switch simulates the schedule produced by GPS. Whenever a packet must be chosen the switch examines its queue and sends the packet that would be serviced earliest under GPS, assuming that no further packets arrive. Parekh and Gallager [53, 54] analyzed WFQ (although they used the name Packet-by-Packet Generalized Processor Sharing (PGPS)) and showed that if $\Phi_i = r_i$ then all session-i packets reach their destinations in time,

$$\frac{b_i + 2(d_i - 1)}{r_i} + d_i = O\left(\frac{b_i + d_i}{r_i}\right).$$

One advantage of Weighted Fair Queueing is that it is simple to implement. Its main disadvantage is that the delay bound grows as the *product* of d_i and $1/r_i$. It is tempting to believe that this is best possible since it is conceivable that each packet must wait for $1/r_i$ steps at every switch on its path. Indeed, there are a number of other algorithms with delay bound $O((b_i + d_i)/r_i)$, for example Virtual Clock [73] and Frame-based Fair Queueing [64, 65]. Our results show however that this multiplicative bound is not optimal since we prove the existence of a schedule with delay bound $O(b_i/r_i + d_i)$.

Rabani and Tardos [57] obtained an randomized algorithm with an additive delay bound. They allowed each packet to be dropped with probability p and they achieved a delay bound of $O(R) + (\log^* p^{-1})^{O(\log^* p^{-1})} D + poly(\log p^{-1})$ where $R = \max_i 1/r_i$ and $D = \max_i d_i$. (They only considered the case in which $b_i = 1$.) Note that this bound is not session based which implies that if one session has a small rate or a long distance then the delay bounds for all sessions will suffer. Ostrovsky and Rabani [51] were able to improve the above delay bound to $O(R + D + \log^{1+\delta} p^{-1})$ for some small constant δ. The algorithms of Rabani *et al.* are distributed in the sense that packets carry some information but knowledge of the entire network is not assumed. The delay bounds that we achieve in this thesis are session based and we do not drop

packets.

The earliest work on the session-oriented model was performed by Cruz [17, 18]. He showed that the maximum delay is bounded for all greedy protocols on all layered directed acyclic graphs. Tassiulas and Georgiadis [66] were able to show a similar result for the ring. Papers that study leaky-bucket regulated traffic include [21, 68].

Chapter 3

Connectionless Packet Routing

3.1 Preliminaries

Throughout our discussion of the connectionless model we shall phrase results in terms of an *adversary* that adds packets to the system, and a *contention-resolution protocol* that moves packets across edges. We use $(G, \mathcal{A}, \mathcal{P})$ to denote an adversary $\mathcal{A}$ and a protocol $\mathcal{P}$ acting on a graph G. We view each time step t of this system as consisting of three phases.

(i) Packets are injected by $\mathcal{A}$.

(ii) Packets are moved by $\mathcal{P}$.

(iii) Packets that reach their destinations in phase (ii) are absorbed.

Definition 3.1.1 *A packet is said to* require *an edge e at time t if e lies on the path from its position at time t to its destination.*

For simplicity we shall assume that when a packet is injected, its assigned path is *simple*; namely, it does not contain any edge more than once. It is not difficult, however, to remove this assumption.

Definition 3.1.2 *We say that $\mathcal{A}$ is a* bounded adversary, *of rate (σ, r), if for all edges e and all intervals I of t consecutive steps it injects no more than $\sigma + rt$ packets during I that require e at their time of injection.*

Our results will be presented in terms of the maximum delay experienced by packets, the maximum queue size and the maximum number of packets in the system. The following result shows that there is a close relation between these quantities.

Theorem 3.1.3 *Let $(G, \mathcal{A}, \mathcal{P})$ be a system, where G is a graph with m edges, $\mathcal{P}$ is any greedy protocol, and $\mathcal{A}$ is an adversary of rate $(w, 1-\varepsilon)$, $0 < \varepsilon < 1$.*

1. *If the maximum queue size is k, then the maximum number of packets in the system is at most mk.*

2. *If the maximum number of packets in the system is n and $n \geq \sigma + 1$ then the delay for a packet whose path has length d is at most $2nd\varepsilon^{-1}$.*

3. *If the maximum delay experienced by a packet is τ then the maximum queue size is at most τ.*

Proof: Properties 1 and 3 are immediate. It remains to prove Property 2. Suppose there are never more than n packets in the system, and consider a queue q at time t. We claim that q becomes empty sometime in the next $2n\varepsilon^{-1}$ steps. For if not, then a packet must leave q in each of the next $2n\varepsilon^{-1}$ time steps. But there are only n packets in the system at time t, and no more than

$$\begin{aligned} \sigma + 2n\varepsilon^{-1}(1-\varepsilon) &\leq \sigma + 2n\varepsilon^{-1} - 2n \\ &< 2n\varepsilon^{-1} - n \end{aligned}$$

packets arrive over the next $2n\varepsilon^{-1}$ time steps — this contradicts the assumption that a packet leaves q in every one of these time steps.

From this it follows that no packet remains in a queue for more than $2n\varepsilon^{-1}$ time steps. Hence, if the path of a packet has length d then the delay that it experiences will be at most $2nd\varepsilon^{-1}$. □

3.2 Stability of protocols

In this section we focus on the issue of stability for *protocols*: given a contention resolution protocol $\mathcal{P}$, can we provide a delay bound for every network G and every bounded adversary $\mathcal{A}$? We first discuss four simple protocols for which the answer is affirmative. Our upper bounds for all these protocols are exponential in the maximum path length d; thus, while the bounds are large in general, they are better when all packets require only short paths. In Section 3.2.2 we discuss several simple and very common protocols that are not stable.

3.2.1 Stable protocols

Recall that the *Farthest-to-Go* (FTG) protocol gives priority to a packet that still has to cross the largest number of edges.

Theorem 3.2.1 *Let G be a directed network, and $\mathcal{A}$ a bounded adversary of rate $(\sigma, 1-\varepsilon)$, with $\varepsilon > 0$. Then the system $(G, \mathcal{A}, FTG)$ is stable. If m is the number of edges in G and d is the maximum path length then the maximum delay experienced by a packet is at most $O(m^{d-1}\sigma/\varepsilon)$.*

Proof: We prove this result by a backwards induction. Let us define $k_i = 0$ for $i > d$ and $k_i = m\sum_{j>i} k_j + m\sigma$ for $1 \leq i \leq d$. We claim that for all $j > i$ the number of packets in the system that still have to cross *exactly* j edges is at most k_j.

This is trivial for $j > d$ since each packet has to cross at most d edges. Now consider a particular edge e and let $X_i(t)$ be the set of packets in the queue of e that still have to cross *at least* i edges at time t. Let t be the current time, let t' be the most recent time step preceding t in which $X_i(t')$ was empty and let t'' be some time step in the interval $(t', t]$. The number of packets with at least $i+1$ edges to cross is at most $\sum_{j>i} k_j$ by the inductive hypothesis. The number of packets injected between times t' and t'' that could be in the queue for edge e is at most $\sigma + (1-\varepsilon)(t''-t')$ by the definition of the adversary. Since we are using FTG, at time t'' a packet from $X_i(t'')$ is chosen to cross edge e. (Note that $X_i(t'') \neq \emptyset$.)

Set $t'' = t$. By the above discussion we have,

$$\begin{aligned} |X_i(t)| &\leq \sum_{j>i} k_j + \sigma + (t-t')(1-\varepsilon) - (t-t') \\ &\leq \sum_{j>i} k_j + \sigma - \varepsilon(t-t'). \end{aligned}$$

The above inequalities have two consequences. First, the number of packets in the system that still have to cross i edges is always at most $m\sum_{j>i} k_j + m\sigma = k_i$ and so the inductive step holds. Second, $t - t'$ cannot be greater than $\frac{1}{\varepsilon}(\sum_{j>i} k_j + \sigma)$. Hence this expression gives the maximum amount of time that a packet with i edges still to cross can remain in a queue. Therefore under FTG the maximum number of packets in the system is bounded by $\sum_{j\geq 1} k_j$ and the maximum delay experienced by any packet is at most,

$$\begin{aligned} \sum_{i=1}^{i=d} \frac{1}{\varepsilon}\left(\sum_{j>i} k_j + \sigma\right) &= \frac{1}{m\varepsilon}\sum_{i=1}^{d} k_i \\ &= \frac{1}{m\varepsilon}\sum_{i=1}^{d} \frac{(m^{i+1}-m)\sigma}{m-1} \\ &= \frac{(m^{d+2}-m^2-dm(m-1))\sigma}{\varepsilon m(m-1)^2} \\ &= O\left(\frac{m^{d-1}\sigma}{\varepsilon}\right). \end{aligned}$$

□

Theorem 3.2.2 *The protocol Nearest-to-Source (NTS), which gives priority to the packet that is closest to its source, is stable. The maximum delay is at most* $O(m^{d-1}\sigma/\varepsilon)$.

Proof: The proof is almost identical to the proof for FTG. The only difference is that we use a standard induction on the number of edges crossed as opposed to a backwards induction on the number of edges still to cross. □

In [1] the following theorem about Shortest-in-System and Longest-in-System is proved. Recall that SIS (resp. LIS) gives priority to the packet that has been in the system for the shortest (resp. longest) time.

Theorem 3.2.3 *The protocols SIS and LIS are both stable and have maximum delay upper bounded by* $O(\sigma/\varepsilon^d)$.

As discussed in Chapter 2, Nearest-to-Source is the simplest of these four stable protocols to implement. This is because a packet must simply keep track of the number of edges that it has crossed. For SIS and LIS global timing is required and for FTG each packet must possess information about the length of its path.

3.2.2 Protocols that are not stable

The classification of the eight simple, distributed, deterministic protocols with respect to stability is completed by the following theorem from [1].[1]

Theorem 3.2.4 *The protocols NTG, FFS, FIFO, and LIFO are not stable, i.e. for each of these protocols there exists a network and an adversary of rate less than one such that the number of packets in the resulting system grows without bound (and hence the delay experienced by packets grows without bound.*[2]*)*

The instability of NTG is somewhat surprising since one would think that by allowing the packets that are near their destinations to move forward, one would be keeping the number of packets in the system as small as possible. Indeed, NTG is a protocol that has been proposed for routing in parallel machines. However, a bad property of NTG is that a queue of packets that has a long distance to travel can be held up at a switch by a stream of packets that only wish to cross one edge. This allows the adversary to continually increase the number of packets in the system.

The protocols NTG and FIFO can be made unstable on the simple network shown in Figure 3-1. For this network NTG can be made unstable by an adversary of rate $r > 1/\sqrt{2}$ and FIFO can be made unstable by an adversary of rate $r > 0.85$. The

[1]The protocol FFS is not considered in [1] but the proof of its instability is very similar to the proof for NTG.

[2]However, each packet *will* leave the system eventually. This is because if the queue for edge e is non-empty then the rate at which packets cross edge e is faster than the rate at which packets requiring edge e are injected. Hence every queue will empty at some time in the future.

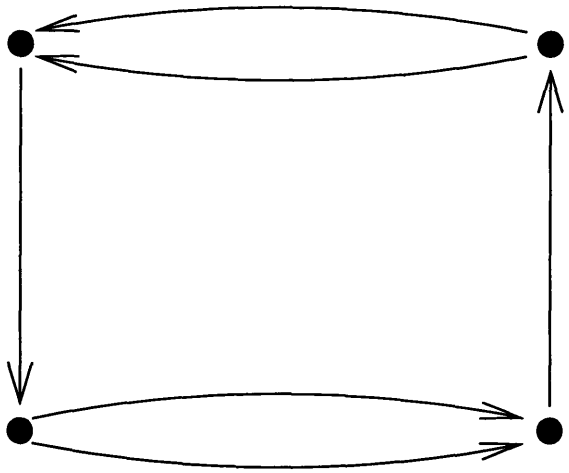

Figure 3-1: A 4-node network on which NTG and FIFO can be made unstable.

LIFO protocol and the FFS protocol can be made unstable on similar networks. A rate of $r > 1/\sqrt{2}$ is sufficient for instability of both LIFO and FFS.

3.3 Bounds on delay for stable protocols

We have already noted that for all four of the stable protocols presented in Section 3.2.1, we have only been able to show exponential upper bounds on the maximum delay. In this section we show that three of the protocols presented there actually *produce* exponential delay, for some network G and some adversary $\mathcal{A}$.

In Section 3.3.2, we describe a simple distributed randomized greedy protocol that with high probability has polynomially bounded delays and polynomially bounded queue sizes.

3.3.1 Exponential lower bounds for SIS, NTS and FTG

We now show that under the protocols SIS, NTS and FTG the delays can be exponential. We first present a proof of this result for SIS and NTS; the proof for FTG is similar.

In order to make the result more general, we use an adversary with a minimal amount of burstiness. We say that an adversary $\mathcal{A}$ has rate $1 - \varepsilon$, if for every $t \geq 1$,

every interval I of t steps, and every edge e, $\mathcal{A}$ injects no more than $1 + (1 - \varepsilon)t$ packets during I that require e at the time of injection.

Consider first a linear array L with $m+2$ nodes $0, 1, \ldots, m+1$, with two parallel edges e_i^0 and e_i^1 from node i to node $i+1$, for $0 \leq i \leq m-1$, and with an edge e_m from node m to node $m+1$. Choose an $\varepsilon \leq 1/(m+2)$ and an $s \geq 2m+1$. We attach to L a tree T such that during an interval of s steps, an adversary $\mathcal{A}$ with rate $1-\varepsilon$ can inject $(1-\varepsilon)s$ packets at the leaves of T with the following properties. They all reach the root of T in the last step of the interval and they all require a common edge. This tree T simply consists of $(1-\varepsilon)s$ "branches", each of which is a chain connected to the root.[3] The kth branch has length $k/(1-\varepsilon)$. In total T has $O(m^2)$ edges. The graph G is obtained by connecting L and T, making the node 0 of L the root of T. (See Figure 3-2.)

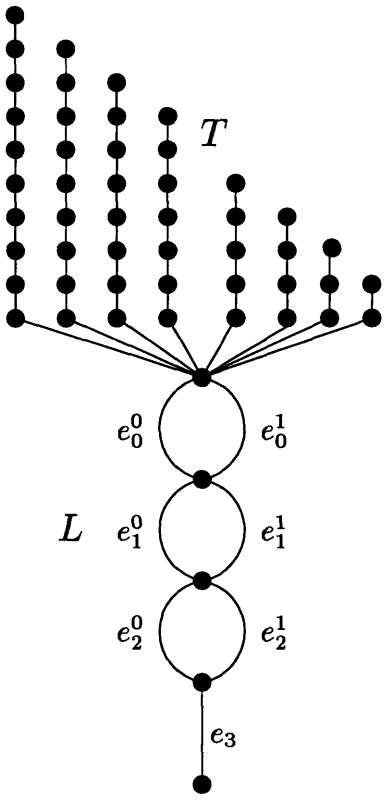

Figure 3-2: The network used for the lower bound. Here $m = 3$, $s = 10$ and $\varepsilon = 1/5$.

[3]Note that if the burst parameter satisfied $\sigma \geq (1-\varepsilon)s$ then we could simply inject $(1-\varepsilon)s$ at the root. We would not need to use the rest of the tree T. However, the tree allows us have burst parameter 1, thus making the result more general.

We now construct an adversary $\mathcal{A}$ with rate $1-\varepsilon$ that injects packets in phases of s steps each. We number the 2^m first phases from 0 to 2^m-1. For some fixed $i \in \{0,\ldots,2^m-1\}$, let $b_{m-1}\ldots b_0$ be the m-bit binary representation of i. Then, in phase i the adversary injects $(1-\varepsilon)s$ packets at the leaves of the subgraph T of G, all requiring edges $e_0^{b_0}e_1^{b_1}\ldots e_{m-1}^{b_{m-1}}e_m$, so that all of them reach node 0 in the last step of phase i. It also injects $(1-\varepsilon)s$ packets requiring only edge $e_j^{\bar{b}_j}$, for all $0 \le j \le m-1$. (Here $\bar{b}_j$ is the complement of b_j.)

Let us define $k_0 = (1-\varepsilon)s$, and $k_j = 2k_{j-1} - \varepsilon s 2^{j-1}$ for $1 \le j \le m$. The crucial lemma is the following. The intuition is that we build up a set of packets that wish to cross e_j^0. We then simultaneously hold up this set of packets using single edge injections and build up another set of packets that wish to cross e_j^1. The two sets of packets are then able to merge since they were injected at different times. This creates a larger set of packets that all wish to cross an edge lower down the graph.

Lemma 3.3.1 *For all $j \in \{0,\ldots,m\}$, let $i_j \in \{0,1,\ldots,2^{m-j}-1\}$ and $b_{m-j-1}\ldots b_0$ be the $(m-j)$-bit binary representation of i_j. Then, at the end of phase $2^j(i_j+1)-1$ there are at least k_j packets in the system still requiring edges $e_j^{b_0}e_{j+1}^{b_1}\ldots e_{m-1}^{b_{m-j-1}}e_m$. All these packets are queued at edges of the subgraph L of G.*

Proof: We use induction on j. The claim is trivially true for $j=0$ since, by the definition of $\mathcal{A}$, at the end of phase i there are $(1-\varepsilon)s = k_0$ packets in node 0 all requiring edges $e_0^{b_0}e_1^{b_1}\ldots e_{m-1}^{b_{m-1}}e_m$, where $b_{m-1}\ldots b_0$ is the m-bit binary representation of i.

Let now assume the result holds for some j and consider some i_{j+1} whose $(m-j-1)$-bit binary representation is $b_{m-j-2}\ldots b_0$. Let $i_j^0 = 2i_{j+1}$ and let $i_j^1 = 2i_{j+1}+1$. Then, the $(m-j)$-bit binary representation of i_j^0 is $b_{m-j-2}\ldots b_0 0$ and the $(m-j)$-bit binary representation of i_j^1 is $b_{m-j-2}\ldots b_0 1$.

From the induction hypothesis, at the end of phase $2^j(i_j^0+1)-1$ there are k_j packets in L requiring $e_j^0 e_{j+1}^{b_0}\ldots e_{m-1}^{b_{m-j-2}}e_m$. Since i_j^0+1 is an odd number, the m-bit binary representation $b_{m-1}\ldots b_0$ of any $i \in \{2^j(i_j^0+1),\ldots,2^j(i_j^0+1)+2^j-1\}$ has the bit $b_j = 1$. Hence, during these 2^j phases all the packets injected requiring e_j^0 are

single-edge injections. Therefore, during these 2^j phases there are $(1-\varepsilon)s2^j$ packets injected that require the single edge e_j^0. Under SIS and NTS these new injections have higher priority. In total, $s2^j$ packets cross edge e_j^0 during the above 2^j phases. Hence at the end of phase $2^j(i_j^0+1)+2^j-1 = 2^j(i_j^1+1)-1$ there are at least $k_j - s2^j + (1-\varepsilon)s2^j = k_j - \varepsilon s2^j$ packets in L still requiring edges $e_j^0 e_{j+1}^{b_0} \ldots e_{m-1}^{b_{m-j-2}} e_m$.

Also by the induction hypothesis, at the end of phase $2^j(i_j^1+1)-1$ there are at least k_j packets in L requiring edges $e_j^1 e_{j+1}^{b_0} \ldots e_{m-1}^{b_{m-j-2}} e_m$. Therefore, there are at least $2k_j - \varepsilon s2^j = k_{j+1}$ packets in L requiring edges $e_{j+1}^{b_0} \ldots e_{m-1}^{b_{m-j-2}} e_m$ at the end of phase $2^j(i_j^1+1)-1 = 2^{j+1}(i_{j+1}+1)-1$. □

Theorem 3.3.2 *At the end of phase 2^m-1 there are at least $(2m+1)2^{m-1}$ packets in the system requiring edge e_m, and there are at least 2^{m-1} packets in some queue of the system. Hence some packet will experience delay 2^{m-1}.*

Proof: From Lemma 3.3.1 with $j=m$ and $i_j=0$, at the end of phase 2^m-1 there are at least k_m packets in L requiring edge e_m. Then, the theorem follows, since $k_m = 2^m k_0 - m\varepsilon s2^{m-1} = s2^{m-1}(2-\varepsilon(m+2)) \geq (2m+1)2^{m-1}$. There are only $2m+1$ queues where these packets can be held, hence some queue contains at least 2^{m-1} packets. One of these packets will experience delay 2^{m-1}. □

The proof for FTG is very similar. For $1 \leq i \leq m$ attach a chain of $m+1$ edges to node i of L. Call this chain C_i. The total number of edges is still $O(m^2)$. Now, instead of injecting packets that wish to traverse the single edge $e_i^{\bar{b}_i}$ the adversary injects packets that wish to traverse the concatenation of the edge $e_i^{\bar{b}_i}$ and the chain C_{i+1}. This ensures that these packets have priority over the packets that they need to block.

3.3.2 A randomized greedy protocol with polynomial delay bounds

For completeness and in order to provide a contrast with the exponential lower bounds of the previous section, we briefly describe the randomized greedy protocol RANDOM

from [1] that has a polynomial delay bound. This bound is not worst-case since if the system is run indefinitely then the number of packets in the system will eventually be more than any fixed bound. We say that a randomized protocol $\mathcal{P}$ has a polynomial delay bound if there is a polynomial $p(\cdot)$ such that for any network G with m edges, any adversary $\mathcal{A}$, any $t > 0$, and any $k > 1$, the probability that at time t there are packets that have been in the system $(G, \mathcal{A}, \mathcal{P})$ for more than time $kp(m)$ is exponential in $-k$.

The bound obtained is polynomial in $d \log m$; thus, for systems in which only short paths are used, this bound is polylogarithmic in the network size.[4]

A brief description of the protocol RANDOM

Let $\mathcal{A}$ be an adversary of rate (σ, r). Let d denote the length of the longest simple directed path and m the number of edges in G. Time is divided into intervals $X_1, X_2, \ldots$ of length $T = \Theta(d \log m/(1-r))$. Priorities are randomly assigned to the packets in such a way that packets injected during the interval X_i always have priority over packets injected during the interval X_j for $i < j$. For this reason the protocol resembles LIS. The exact method for assigning priorities is derived from techniques of Leighton, Maggs and Rao for a static routing problem [44]. (We shall consider the results of Leighton et al. in more detail in Chapter 5.) It is shown in [1] that by using RANDOM the probability that at time t there are packets that have been in the system for more than time $k \cdot poly(d \log m)$ is exponential in $-k$, i.e. RANDOM has a polynomial delay bound.

3.4 Remarks

The main open question concerning the connectionless model is whether or not there is a *deterministic, distributed* queueing protocol with a polynomial delay bound. Given the similarities between LIS and RANDOM, and the fact that there is no known

[4] It can also be shown that if the number of packets in the system is ever more than $kp(m)$, then with high probability there is a net decrease in packets over a fixed subsequent time interval.

exponential lower bound for LIS, a specific open question is to determine whether LIS itself has a polynomial delay bound. (It is noted in [1] that RANDOM can be converted into a deterministic, centralized protocol with a polynomial delay bound; thus, the emphasis is on finding a protocol that is both deterministic and distributed.)

Most of this chapter has concentrated on adversaries with rates arbitrarily close to 1. It is also interesting to study the behavior of protocols against adversaries of rates bounded away from 1. The instability results stated in Section 3.2.2 for NTG, FFS, FIFO and LIFO were for adversaries of rate greater than 0.5. However, a result of Borodin, Kleinberg, Sudan, and Williamson [12] shows that on a torus-like network there exist adversaries of arbitrarily small positive rates that cause NTG and FFS to be unstable. It would be interesting to know if for either FIFO or LIFO there is some constant injection rate below which it is stable. An extreme possibility is that there is a "Zero-One" type of result. Such a result would say that all greedy protocols are either stable for rates arbitrarily close to 1 or else can be made unstable for arbitrarily small rates.

A similar issue to the above is whether or not FTG, NTS or SIS have polynomial delay bounds for sufficiently small injection rates? It should also be noted that the adversary used in Section 3.3 has rate $1 - \Theta(1/m)$. Recently, Leighton [48] has shown that FTG has an exponential lower bound against an adversary whose rate is independent of the size of the network. This result extends easily to NTS. The network used is similar to the one on which FIFO and NTG were unstable. (See Figure 3-1.)) An analogous result is not known for SIS.

Chapter 4

The Session-Oriented Model

4.1 The Model

We recall the definition of the session-oriented model. Let $\mathcal{N}$ be a network of arbitrary topology. A *session* consists of a simple path through the network and a set of packets that travel along the path. (A path is *simple* if it uses each edge at most once.) We use d_i to denote the *length* of session i, i.e. the number of edges in the path associated with session i.

Each session i has an *injection rate* r_i. This rate constrains the injection of new packets into the session so that, during any interval of t consecutive steps, at most $tr_i + 1$ packets can be injected into session i, for any t. We assume that the exact injection patterns are determined by an adversary, subject to this constraint.

As in the connectionless model we assume that at most one packet can traverse an edge at each time step. When two packets simultaneously contend for the same edge, one of them must wait in a queue. During the traversal of their paths, packets wait in two different kinds of queues. After a packet has been injected, but before it leaves its source, the packet is stored in an *initial queue*. Once the packet has left its source the packet is stored in an *edge queue* whenever it is waiting to cross an edge. The *delay* experienced by a packet is the total time between its injection and the time at which it reaches its destination.

Our main goal is to minimize the delay experienced by packets. A secondary

goal is to bound the size of the edge queues. As discussed in Chapter 2, it is more important to bound the size of the edge queues than the initial queues. Recall that for the connectionless model we had to restrict the rate at which packets are injected into the network, otherwise the number of packets in the system would grow without bound, thus making it impossible to provide delay bounds and bounds on queue size. The same is true here. The assumption that we make in the session-oriented model is that the sum of the rates of the sessions using any edge e is at most $1 - \varepsilon$, for a constant $\varepsilon \in (0, 1)$.

4.1.1 Template-Based Schedules

We focus on the problem of timing the movements of the packets along their paths. A *schedule* specifies which packets move and which packets wait in queues at each time step. In this thesis most of the schedules that we consider are *template-based.* In a template-based schedule each edge is associated with a *template.* A template of size M can be viewed as a wheel with M slots. Each slot contains at most one *token.* Each of these tokens is associated with a particular session. The wheel "spins" at a rate of one slot per time step. We allow a session-i packet to cross the corresponding edge only if a session-i token appears. We then say that the packet has *used* or *obtained* the token. For each session-i token, the session-i packet that uses it will be the one that has been waiting to cross the edge for the longest amount of time, i.e. the session-i packets use the session-i tokens in a First-Come-First-Served manner.

If we are considering template-based schedules our problem reduces to the problem of assigning tokens to the slots on the templates. Our usual strategy will be to assign tokens in *packet-groups.* A packet-group of tokens for session i will consist of d_i tokens, one for each edge on session i. (There will be many packet-groups for each session.) For each method of assigning tokens we initially provide bounds on delay and queue size for a *packet-group* schedule. In this schedule, packets use tokens for the initial edge in a First-Come-First-Served manner. However, once a packet has used one token from a particular packet-group then for subsequent edges it *only* uses tokens from that packet-group. In Section 4.2 we show that these bounds give us bounds

for the corresponding template-based schedule in which session-i packets use session-i tokens in a First-Come-First-Served manner on all edges. If no ambiguity arises, we shall often refer to scheduling packets rather than assigning tokens in packet-groups.

Template-based schedules are periodic. The length of the period is the lowest common multiple of all the template sizes. It is not unreasonable to consider schedules that can only be computed with a large amount of computational effort. This is because the templates only need to be constructed once and can then be used to schedule packets indefinitely. All that a switch must do is store the template and "spin" it one slot every time step. The only schedule we consider that is not template-based is the distributed schedule of Section 4.3.2.

4.1.2 Lower Bound

Observe that d_i is always a lower bound on the delay for session i, since every session-i packet crosses d_i edges.

It is easy to see that $\Omega(1/r_i)$ (and hence $\Omega(1/r_i + d_i)$) is an existential lower bound. Consider n sessions with rate $r = (1-\varepsilon)/n$, all of which have the same initial edge. If a packet is injected into each session simultaneously, one of the packets requires $n = \Omega(1/r)$ steps to cross e.

For *any* given set of sessions, if we are using a template-based schedule then $\Omega(1/r_i)$ is a lower bound on delay for some session i. Consider some edge e and suppose that the sum of the rates of sessions that pass through edge e is $1-\varepsilon$. Suppose also that on the template for edge e the maximum time between appearances of session-i tokens is at most $(1-2\varepsilon)/r_i$ for all sessions i that pass through edge e. This template must have at least $\sum_i Mr_i/(1-2\varepsilon) > M$ tokens where M is the number of slots on the template. This contradicts the fact that each slot has at most one token. Hence there is some session i that has two consecutive tokens that are at least $(1-2\varepsilon)/r_i$ slots apart. If a session i packet is injected just after one of these tokens appears then it must wait at least $\Omega(1/r_i)$ time steps before it can cross edge e. This means that for template-based schedules, there exists a session i such that some session-i packet must experience delay $\Omega(1/r_i + d_i)$. An interesting open problem is to determine

whether or not this bound holds for non-template-based schedules.

4.1.3 Delay Insertion

The main technique that we use for obtaining schedules with small delay is random "delay-insertion". The intuition here is that if each packet is delayed by a random amount then it is unlikely that many packets will try to cross an edge at the same time. This delay insertion technique was introduced by Leighton et al. in [44, 45] in the context of static routing. (In the *static* routing problem, all packets are present in the network initially.) Since our main result employs many techniques from [44], we shall summarize them in detail in Section 5.1.

4.1.4 Leaky-bucket injection model

Our techniques can be generalized for bursty sessions that are leaky-bucket constrained. A leaky-bucket constrained session i has an associated "bucket" size $b_i \geq 1$. The number of session i packets injected into session i during a time interval of length t is at most $r_i t + b_i$ for all t. In order to deal with bursty injections, we "shape" the session-i arrivals at the initial queue. If a large burst of packets arrive, we hold them in the initial queue as though they had not been injected yet, and then treat them as if they were injected every $1/r_i$ steps. To do this, the initial queue requires b_i extra space, and the delay experienced by session-i packets is increased by at most b_i/r_i steps.

4.1.5 Chernoff Bounds and the Lovász Local Lemma

We shall make frequent use of a Chernoff Bound [15] and the Lovász Local Lemma [62, pages 57-58]. For ease of reference we present them here.

[Chernoff Bound] *Let X_i be n independent Bernoulli random variables with probability of success p_i. Let $Y = \sum_{i=1}^{n} X_i$ and let $\mu \geq \sum_{i=1}^{n} p_i$. Then for $0 < \delta < 1$, we*

have,

$$\Pr[\, Y > (1+\delta)\mu \,] \leq e^{-\delta^2\mu/3}.$$

[Lovász Local Lemma] *Let $A_1, \ldots, A_m$ be a set of "bad events" in a probability space each occurring with probability p and with dependence at most d (i.e. every bad event is mutually independent of some set of $m - d$ other bad events). If $4pd < 1$, then with probability greater than zero no bad event occurs.*

Our discussion of the session-oriented model is organized as follows. We first show that if we can provide bounds for delay and queue size in packet-group schedules then this gives us bounds for delay and queue size in template-based schedules. In Section 4.3.1 we present a randomized centralized schedule that with high probability has a delay bound of $O(1/r_i + d_i \log(m/r_{\min}))$ and a queue size of $O(\log(m/r_{\min}))$ where m is the number of edges and $r_{\min} = \min_i r_i$. In Section 4.3.2 we present a distributed schedule that has the same delay bound (but no bound on queue size). Chapter 5 is dedicated to a proof that there exists a schedule with delay bound $O(1/r_i + d_i)$ and constant queue size.

4.2 Converting Packet-Group Schedules into Template-Based Schedules

In this section we show how to use bounds on delay and queue size for packet-group schedules to obtain bounds on delay and queue size for template-based schedules. Recall that a session-i packet-group consists of d_i tokens, one for each edge on session i. The token for the first edge is known as the *initial token* in that packet-group. In a packet-group schedule, once a packet has used the initial token from a packet-group it can only use tokens from that packet-group to cross edges. In a template-based schedule, the session-i packets use the session-i tokens in a First-Come-First-Served manner on all edges.

To emphasize the distinction between the two types of schedule suppose that e and f are two consecutive edges on the path for session i. Let $\lambda_{1,e}$ and $\lambda_{1,f}$ be two tokens for edges e and f respectively that belong to a packet-group L_1. Let $\lambda_{2,e}$ and $\lambda_{2,f}$ be two tokens for edges e and f that belong to a packet-group L_2. Suppose also that $\lambda_{1,e}$ appears *before* $\lambda_{2,e}$ but $\lambda_{1,f}$ appears *after* $\lambda_{2,f}$. In a packet-group schedule, if a packet uses $\lambda_{1,e}$ to cross edge e then it must use $\lambda_{1,f}$ to cross edge f. However, in a template-based schedule, a packet could use $\lambda_{1,e}$ to cross edge e and then use $\lambda_{2,f}$ to cross edge f.

We now state the result of this section. Suppose that we have a packet-group schedule $\mathcal{S}_{PG}$ such that,

1. Each session-i packet obtains a session-i token for the first session-i edge within $\ell_i^{(1)}$ steps of its injection. (We refer to these tokens as *initial tokens.*)

2. Once a session-i packet has obtained an initial token then it reaches its destination within another $\ell_i^{(2)}$ steps.

3. If each session-i token is used by a session-i packet then there are never more than q_i session-i packets in any edge queue.

Now let $\mathcal{S}_{TB}$ be the template-based schedule that uses the same tokens.

Theorem 4.2.1 *The schedule $\mathcal{S}_{TB}$ satisfies the above three properties 1, 2 and 3. Therefore bounds for delay and edge-queue size in $\mathcal{S}_{PG}$ imply bounds for delay and edge-queue size in $\mathcal{S}_{TB}$.*

Proof: The fact that $\mathcal{S}_{TB}$ satisfies Property 1 is trivial since in both $\mathcal{S}_{PG}$ and $\mathcal{S}_{TB}$ the session-i packets obtain tokens for the initial edge in a First-Come-First-Served manner.

We now show that $\mathcal{S}_{TB}$ satisfies Property 2. Let $e_1, e_2, \ldots, e_d$ be the edges for session i. Let κ_{m,e_j} be the mth session i token to appear for edge e_j. Let P_m be the packet-group that contains κ_{m,e_1}.

Lemma 4.2.2 *The token $\kappa_{m,e_{j+1}}$ appears after the token κ_{m,e_j}.*

Proof: Suppose not. Note that for each packet-group $P_{m'}$, the token in $P_{m'}$ for edge e_{j+1} must appear after the token in $P_{m'}$ for edge e_j, otherwise $\mathcal{S}_{PG}$ would have no delay bound. Therefore the set of packet-groups that contain $\kappa_{1,e_{j+1}}, \ldots, \kappa_{m,e_{j+1}}$ is a subset of the set of packet-groups that contain $\kappa_{1,e_j}, \ldots, \kappa_{m-1,e_j}$. However, each packet-group contains exactly one token for each edge. Hence we have $m-1$ packet-groups that contain m tokens for edge e_{j+1}. This is a contradiction. □

Lemma 4.2.3 *If the initial token used by a packet p is in P_m then in $\mathcal{S}_{TB}$, packet p crosses edge e_j either at or before the time that κ_{m,e_j} appears.*

Proof: The proof is by induction on m and j. If packet p uses the initial token in P_1 then by Lemma 4.2.2 and the fact that session-i packets obtain their tokens in $\mathcal{S}_{TB}$ in a First-Come-First-Served manner, p is always able to use $\kappa_{1,j}$ to cross edge e_j. For all m the result is clear for $j = 1$ by definition of P_m. Now suppose that it holds for $j' < j$ and $m' < m$. Therefore p crosses e_{j-1} by the time that κ_{m,e_j} appears and hence by Lemma 4.2.2 it is waiting to cross edge e_j by the time that κ_{m,e_j} appears. By the inductive hypothesis all of the packets that obtained initial tokens from packet-groups in $\{P_1, \ldots, P_{m-1}\}$ have crossed edge e_j by the time κ_{m-1,e_j} appears. Therefore p can use κ_{m,e_j} if it has not already used an earlier token. □

We can now prove that in $\mathcal{S}_{TB}$, no session-i packet experiences delay more than $\ell_i^{(2)}$ once it has obtained its initial token. Let p be a packet that uses the initial token in P_m. Let κ_{x,e_d} be the token for edge e_d that is a member of P_m. If $\mathcal{S}_{PG}$ is used then packet p experiences delay at most $\ell_i^{(2)}$ once it obtains its initial token. Hence there are at most $\ell_i^{(2)}$ steps between the appearance of the initial token and the appearance of token κ_{x,e_d}. By Lemma 4.2.3, in $\mathcal{S}_{TB}$ packet p crosses e_d by the time that κ_{m,e_d} appears.

- **Case 1.** $m \leq x$. By the above comments the delay experienced by packet p in $\mathcal{S}_{TB}$ once it has obtained its initial token is at most $\ell_i^{(2)}$.

- **Case 2.** $m > x$. There are exactly $m-1$ packet-groups whose tokens for e_d appear strictly before κ_{m,e_d}. Since $m > x$, P_m must be one of these packet-groups. Therefore, by the pigeonhole principle, there exists an $m' < m$ such

that the token for edge e_d in $P_{m'}$ appears *no earlier* than κ_{m,e_d}. Let p' be a session-i packet that uses the initial token κ_{m',e_1} in $P_{m'}$. Since $m' < m$, κ_{m',e_1} must appear before the initial token κ_{m,e_1} in P_m. However, if $\mathcal{S}_{PG}$ is used then by the definition of $P_{m'}$, packet p' reaches its destination *no earlier* than the time at which κ_{m,e_d} appears. Hence in $\mathcal{S}_{TB}$, packet p spends less time between obtaining its initial token and reaching its destination than packet p' does in $\mathcal{S}_{PG}$. Therefore in $\mathcal{S}_{TB}$ the delay experienced by p once it has obtained its initial token is at most $\ell_i^{(2)}$.

This completes the analysis of delay bounds. It remains to show that $\mathcal{S}_{TB}$ satisfies Property 3. Suppose that $q_i + 1$ session-i packets are waiting to cross e_j at time t where $j > 1$. (Recall that we are only interested in bounding the size of edge queues.) Suppose that these packets used the tokens $\kappa_{m_1,e_{j-1}}, \kappa_{m_2,e_{j-1}}, \ldots, \kappa_{m_{q_i+1},e_{j-1}}$ to cross edge e_{j-1}. By an argument almost identical to the proof of Lemma 4.2.3, if a packet uses $\kappa_{m,e_{j-1}}$ to cross e_{j-1} then it can use κ_{m,e_j} to cross e_j. Therefore $\kappa_{m_1,e_j}, \kappa_{m_2,e_j}, \ldots, \kappa_{m_{q_i+1},e_j}$ have not appeared by time t otherwise not all of the $q_i + 1$ packets would be waiting. Hence there are at least $m_{q_i+1} - (m_1 - 1) \geq q_i + 1$ packet-groups whose token for edge e_j has appeared at time t but whose token for edge e_{j-1} has not appeared. Therefore if each session-i initial token is used by a session-i packet then in $\mathcal{S}_{PG}$ there would be $q_i + 1$ session-i packets waiting to cross edge e_j at time t. This contradicts the definition of q_i. □

4.3 A Preliminary Result

In this section we present a simple centralized schedule and a simple distributed schedule that achieve delay bounds of $O(1/r_i + d_i \log(m/r_{\min}))$ with high probability. In addition, the centralized schedule has a maximum queue size of $O(\log(m/r_{\min}))$ with high probability. These preliminary results are substantially simpler to prove than the results of Chapter 5 because of the relaxed bounds on delay and queue sizes. Nevertheless, they illustrate the basic ideas that are necessary.

In this section, for ease of presentation, we omit floors and ceilings where they

are necessary. We shall also assume that $1/r_i$ is an integer for all i and there is a constant k such that $k/r_{\min}$ is a multiple of $1/r_i$ for all i.[1]

4.3.1 A Centralized Schedule

We now describe the centralized schedule which we shall call TEMPLATE. Our analysis will apply to the packet-group schedule. However, by the results of Section 4.2, the corresponding template-based schedule will have the same bounds. Let $M = k/r_{\min}$ where k is the constant chosen above. Each template will have size M. We first place r_iM *initial tokens* on the template for the first edge of session i, spaced $1/r_i$ slots apart.

Lemma 4.3.1 *Each session i packet will obtain a session i token at most $2/r_i$ steps after its injection.*

Proof: Suppose that packet p is injected at time t but has not obtained an initial token by time $t + 2/r_i + 1$. Let t' be the last time before $t + 2/r_i + 1$ that there were no session i packets waiting for initial tokens. (Note that $t' < t$.) Between times t' and t at most $(t - t')r_i + 1$ session i packets are injected. However, at least $(t - t' + 2/r_i)r_i - 1 = (t - t')r_i + 1$ initial tokens for session i appear between times t' and $t + 2/r_i$. By the definition of t', each of these tokens was used by a packet. Hence none of the packets injected between t' and t (and in particular packet p) can still be waiting for an initial token at time $t + 2/r_i$. □

Once the initial session-i tokens are placed we delay each of them by an amount chosen uniformly and independently at random from $[L + 1, L + 1/r_i]$, where $L = \frac{\alpha}{2}\log(mM)$ and α is a constant. The intuition is that the random delays would spread out the tokens. After the tokens have been delayed we can be sure that each packet

[1] In this section, we choose to assume the existence of k so as to avoid obscuring the main ideas. If there is no such constant k, we can always show the existence of $\hat{r}_i$ for each session i such that the following holds. i) $\hat{r}_i$ is a fraction of the form s_i/ℓ_i, where s_i and ℓ_i are integers and $\ell_i = \Theta(1/r_i)$ is a power of 2; ii) $r_i \leq \hat{r}_i$; iii) $\sum_i \hat{r}_i \leq 1 - \varepsilon/2$ for all edges. In what follows, if we choose the template size $M = \max_i \ell_i$, then $M = \Theta(1/r_{\min})$ is a multiple of all the ℓ_i's. Then, instead of placing one token every $1/r_i$ slots, we place s_i initial tokens in one slot every ℓ_i slots. In Chapter 5 we shall be more rigorous.

obtains an initial token within $L + 3/r_i$ steps. We now create the packet-groups of tokens. (Recall the definition of packet-group from Section 4.1.1). For every session-i token a placed in the template corresponding to the jth edge, we place a session-i token b on the template corresponding to the $(j+1)$st edge in such a way that b appears exactly $2L$ steps after a.[2]

We observe that two different session-i packets p and p' have different starting times T and T', and therefore different session-i packets do not cross an edge simultaneously. Unfortunately, tokens from different sessions may be placed in one slot, which causes packets from different sessions to cross the same edge simultaneously. The following lemma shows that the tokens are not clustered to any great extent.

Lemma 4.3.2 *At most L tokens appear in any L consecutive slots on any template with probability $1 - 1/(mM)$, where $L = \frac{\alpha}{2}\log(mM)$ and α is a sufficiently large constant.*

Proof: Since the initial tokens for session i are spaced $1/r_i$ apart and each is delayed by an amount chosen independently and uniformly at random from $[L+1, L+1/r_i]$, the expected number of session-i tokens in a single slot is r_i. For a particular interval of L consecutive slots on a particular template, let the random variable X equal the number of tokens in these slots. By linearity of expectations, $E[X] \leq \sum_i r_i L \leq (1-\varepsilon)L$. Since the delays for the initial tokens are chosen independently and all session paths are simple, we have the following by a Chernoff bound. (See Section 4.1.5.)

$$\Pr[\, X > L \,] \leq \Pr[\, X > (1+\varepsilon)(1-\varepsilon)L \,] \leq e^{-\varepsilon^2(1-\varepsilon)L/3}.$$

In the m templates there are at most mM intervals of L consecutive slots. Therefore, by a union bound the probability that more than L tokens appear in *any* L consecutive

[2]Due to the periodic nature of the templates, some session-i tokens for an edge may appear before the token from the first packet-group. This contradicts an implicit assumption of Section 4.2 that all tokens are members of packet-groups. However, if we let all templates spin for one full rotation before packets are injected then this problem does not arise.

slots is bounded by,

$$\begin{aligned} mM \Pr[\, X > L \,] &\leq mM \Pr[\, X > (1+\varepsilon)(1-\varepsilon)L \,] \\ &\leq mMe^{-\varepsilon^2(1-\varepsilon)L/3} \\ &= mMe^{-\varepsilon^2(1-\varepsilon)\alpha \log(mM)/6}. \end{aligned}$$

By choosing a sufficiently large constant α, we can bound the above probability by $1/(mM)$. □

Lemma 4.3.2 is not sufficient to guarantee one token per slot. We resolve this problem by partitioning each template into intervals of L consecutive slots and "smoothing out" each interval as follows.[3] We take the at most L tokens from these slots and rearrange them arbitrarily so that there is at most one token in each slot. We have,

Lemma 4.3.3 *Consider a packet p. Let $\mathcal{K}^p$ be the packet-group that contains the initial token used by p* before *the smoothing process. Let κ_j^p be the token for the jth edge in this packet-group. Then,* after *the smoothing process, token κ_1^p (the initial token) appears after the injection of p and κ_{j+1}^p appears after κ_j^p. Therefore, the packet-group schedule is legal.*

Proof: A token is shifted by at most $L-1$ steps by the smoothing process. Before the smoothing, κ_1^p appears at least L steps after the injection of p and κ_{j+1}^p appears exactly $2L$ steps after κ_j^p. The lemma follows. □

Lemma 4.3.4 *Each session-i packet waits for at most $O(\log(m/r_{\min}))$ steps to cross each edge.*

Proof: Suppose that packet p uses token κ_j^q from packet-group $\mathcal{K}^q$ to cross its jth edge. Then packet p uses κ_{j+1}^q to cross its $j+1$st edge. The token κ_{j+1}^q appears at most $4L = O(\log(m/r_{\min}))$ steps later than κ_j^q. The result follows. □

Therefore,

[3]Here we assume that M is a multiple of L. This can be achieved by choosing M and L to be powers of 2. (See the earlier footnote regarding the existence of an M that is a small multiple of $1/r_i$ for all i.)

Theorem 4.3.5 *With high probability, the randomized centralized schedule* TEMPLATE *has a delay bound of* $O(1/r_i + d_i \log(m/r_{\min}))$ *and a queue size of* $O(\log(m/r_{\min}))$.

Proof: The bound on queue size is immediate from Lemma 4.3.4. The bound on delay follows from Lemma 4.3.4 and the fact that each packet receives an initial token at most $2L + 3/r_i = O(1/r_i + \log(m/r_{\min}))$ steps after its injection. □

With high probability the schedule TEMPLATE assigns at most one token to each template slot. Note that if the first execution of TEMPLATE assigns more than one token to a slot, TEMPLATE can be executed again until the condition of one token per slot is satisfied.

4.3.2 A Distributed Schedule

The above result suggests the following simple *distributed* strategy for scheduling packets so as to achieve small delay. As with the centralized schedule, we place initial tokens on the first edge of session i and then delay each token by an amount chosen independently and uniformly at random from $[1, 1/r_i]$. Suppose that a packet now has its initial token at time T. Then for the kth edge on this packet's path the packet is given a "deadline" of $T + 2L(k-1) + L$, where $L = \frac{\alpha}{2}\log(mM)$. Whenever two or more packets contend for the same edge simultaneously, the packet with the earliest deadline moves. We call this scheme EARLIEST-DEADLINE-FIRST (EDF).

Lemma 4.3.6 *For any edge, at most L deadlines appear in any L consecutive time steps with probability at least $1 - 1/(mM)$, where $L = \frac{\alpha}{2}\log(mM)$ and α is a sufficiently large constant.*

Proof: The proof is almost identical to that of Lemma 4.3.2. □

Lemma 4.3.7 *If for any edge at most L deadlines appear in any L consecutive time steps, then each packet crosses every edge by its deadline.*

Proof: For the purpose of contradiction, let D be the first deadline that is missed. This implies that all deadlines earlier than D are met. Let p be the packet that

misses deadline D for edge e. Since packet p makes its previous deadlines, p must have crossed its previous edge by time $D - L$, or else e must be p's first edge and p must have obtained its initial token by time $D - L$. Hence, at every time step from time $D - L + 1$ to D, packet p is held up by another packet with deadline no later than D. Furthermore, these deadlines must be later than $D - L$ since all deadlines earlier than D are met. Therefore, at least $L + 1$ packets have deadlines for edge e from time $D - L + 1$ to D. This contradicts Lemma 4.3.6. □

Lemmas 4.3.6 and 4.3.7 imply,

Theorem 4.3.8 *With high probability, the randomized distributed schedule* EARLIEST-DEADLINE-FIRST *achieves a delay bound of* $O(1/r_i + d_i \log(m/r_{\min}))$.

Note that EDF does not generate a template for each edge. Instead, it generates a list of $r_i M$ initial deadlines for the first edge of session i, and gives them in order to the session-i packets injected.

Chapter 5

The Session-Oriented Model – $O(1/r_i + d_i)$ Bound

In this chapter we prove the existence of a packet-group schedule with delay bound $O(1/r_i + d_i)$ and constant queue size. The results of Section 4.2 imply that the corresponding template-based schedule has the same bounds. We begin in Section 5.1 by describing an algorithm of Leighton, Maggs and Rao (LMR) [44] for a static routing problem since we shall make use of a number of their techniques. In Section 5.2 we give an overview of our methods for the dynamic problem. The remainder of the chapter contains the details of the proof.

5.1 The Leighton-Maggs-Rao $O(c+d)$ algorithm for static routing

In the static routing problem all the packets are present in the system initially. Each packet wishes to follow a specified path. The congestion c of the problem is the maximum number of paths that pass through any edge and the dilation d is the maximum length of a path. As with our dynamic problem each packet takes one time step to cross each edge and no more than one packet may cross any edge at any time step. The much celebrated result of Leighton, Maggs and Rao states that there exists

a schedule such that all packets reach their destinations in time $O(c+d)$. It is clear that both c and d are lower bounds on the length of the schedule, hence this result is asymptotically optimal. We now summarize their techniques.

An *initial schedule* is constructed in which each packet moves one edge at every time step until it reaches its destination. This is an excellent schedule in terms of delay but it is clearly illegal in the sense that more than one packet could cross an edge during a single time step. The key idea is to *refine* the schedule a certain number of times by introducing delays to packets. In order to understand why this is useful we must present some definitions from [44].

A T-frame is an interval of T consecutive time steps. A T-frame has *frame congestion* C if at most C packet cross any edge during the frame. The T-frame has *relative congestion* R if it has frame congestion C and $R = C/T$. We say that a schedule has *relative congestion* x *and frame size* I if the relative congestion in any frame of size I or larger is at most x.

Since at most c packets wish to cross any edge it is clear that the initial schedule in which packets move one edge at every time step has relative congestion 1 and frame size c. It can be shown that for any schedule with relative congestion x and frame size I, there exists a *good* set of delays that can be inserted into the schedule so that the frame congestion is at most $O(\log^5 I)$ and the relative congestion is at most $(1+o(1))x$. The new schedule is called a *refinement* of the original schedule. When we refer to a delay being *inserted* into a schedule we mean that some packet is held up for an extra time step before it crosses an edge. Note that when a schedule is refined the frame size decreases by a large amount whereas the relative congestion increases by only a small amount.

It can be shown that after $O(\log^* c)$ refinements we obtain a schedule with congestion $O(1)$ and frame size $O(1)$. This means that there is some constant w such that at most w packets cross any edge during any time step. The schedule is then "stretched" by a factor of "w", i.e. every time step is simulated by w time steps to obtain a legal schedule in which at most one packet crosses an edge per time step. In this schedule no packet is delayed by more than $O(c+d)$ steps. (A packet waits

at most $O(c+d)$ steps before it starts moving and then takes time $O(d)$ to reach its destination.) Moreover, the maximum size of each queue is bounded by a constant.

5.2 An $O(1/r_i + d_i)$ bound for the dynamic problem

To obtain a solution for the dynamic problem we shall use many of the ideas from LMR. However, our delay requirements are somewhat more rigorous and so there are three major problems with doing this. We now present these problems and give a brief indication of how we solve them.

1: Packets are injected indefinitely (for infinite time)

In the problem solved by LMR there are only a finite number of packets that need to be scheduled. However in the dynamic problem that we are considering there are an infinite number of packets that arrive over time. Our solution is to divide time into intervals of length $\mathcal{T} = O(1/r_{\min} + d_{\max})$. For some $\mathcal{T}_i = O(1/r_i + d_i)$ we aim to schedule all the packets from session i that are injected during the interval $[k\mathcal{T} - \mathcal{T}_i, (k+1)\mathcal{T} - \mathcal{T}_i)$ so that they move and arrive at their destination within the interval $[k\mathcal{T}, (k+1)\mathcal{T})$. (See Figure 5-1.) Hence we shall be scheduling the intervals of the form $[k\mathcal{T}, (k+1)\mathcal{T}_i)$ independently. Our key result will be to show that we can in fact carry out this scheduling.

2: Delay bounds are session based

In the refinement steps of LMR delays are added to all sessions. In particular, when the frame size is I the packets in each session can be delayed by as many as I time steps. However, if we are to employ a similar strategy with our current dynamic problem then in general we must start with a frame size of $O(1/r_{\min})$, otherwise we would not be able to bound the relative congestion by a constant. Hence the delays that would be inserted to all sessions at the start of the refinement process would have size $O(1/r_{\min})$. This amount of delay is tolerable for session i if $1/r_i + d_i$ is large.

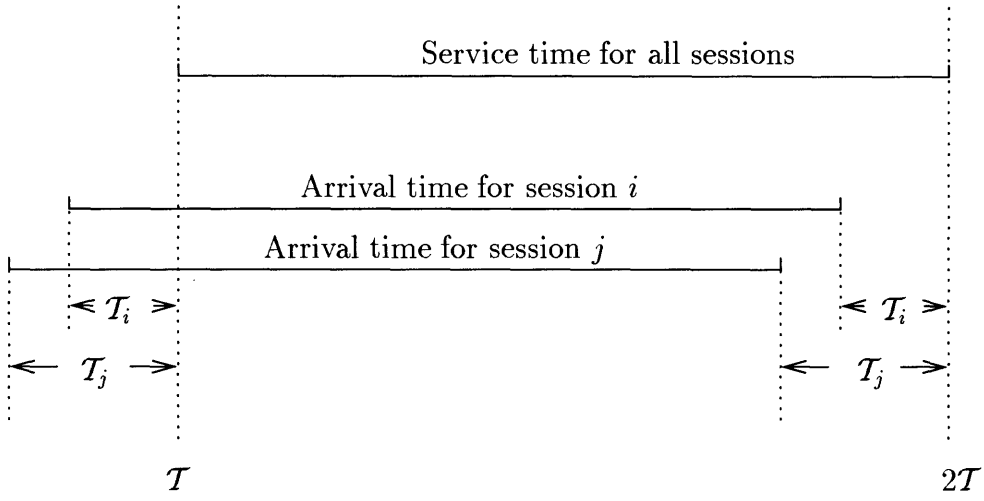

Figure 5-1: The session-i packets that arrive during $[k\mathcal{T} - \mathcal{T}_i, (k+1)\mathcal{T} - \mathcal{T}_i)$ are scheduled during $[k\mathcal{T}, (k+1)\mathcal{T})$. In this figure, $k = 1$.

However, if $1/r_i + d_i$ is small then adding a delay of $O(1/r_{\min})$ time steps to a session-i packet may prevent it from being able to reach its destination in time $O(1/r_i + d_i)$. Hence we would not achieve the desired result.

The solution is to view each session as being either *integral* or *fractional* at each stage of the refinement process. When we view a session of rate r_i as being integral we assume that whole packets are being injected into session i at rate r_i. When we view it as being fractional however we assume that an $\hat{r}_i$ fraction of a packet is being injected at every time step, where $\hat{r}_i$ is a value slightly greater than r_i. At each stage of the refinement process the integral sessions will be those with a high $1/r_i + d_i$ value. These will be able to tolerate the delay added. The fractional sessions in contrast will be those with a small value of $1/r_i + d_i$ which means that they will not be able to tolerate the delay added. However, since we are assuming that fractional packets are being injected into such sessions, the congestion that they produce will be only slightly larger than the sum of the corresponding session rates. Hence it is not necessary to add delays to the packets in these sessions.

Of course, we eventually need to have a schedule for integral packets on all sessions. To achieve this we *convert* some sessions from being fractional to being integral at each stage of the refinement process. Initially we view all sessions as being fractional. By the end of the process all sessions will be integral.

Each stage of the refinement process now consists of two steps, a *refinement* step and a *conversion* step. Suppose that we have a schedule with frame size I and relative congestion x. In the refinement step we introduce some delays to the *integral sessions only*. This is done in order to reduce the frame size in a similar manner to LMR. In the conversion step we convert some sessions from fractional to integral that can tolerate the delay that will be inserted in subsequent refinement steps. The result of the conversion and refinement steps will be a schedule with frame size $\log^5 I$ and relative congestion slightly larger than x. Some sessions may have been converted from being integral to being fractional.

The result of the entire refinement process is a schedule with frame size $O(1)$ and relative congestion 1. In addition all of the sessions are integral and so we are indeed able to schedule whole packets.

3: Obtaining the final schedule

In LMR the result of the refinement process is a schedule with frame size $O(1)$ and relative congestion $O(1)$. This implies that there exists some constant y such that at most y packets cross any edge in any time step. For the static problem it is now easy to construct a schedule in which at most 1 packet crosses an edge in any time step. Every time step is simply replaced by y time steps and the at most y packets that were scheduled to cross an edge in any original time step are now scheduled arbitrarily within the corresponding y time steps.

For our dynamic problem however the solution is not so straightforward. Recall that when discussing the first problem we indicated that we shall divide time up into intervals and then construct an independent schedule in each interval. However, if we are to expand time by replacing each time step by a larger number of steps then these intervals will overlap and hence will no longer be independent.

The solution is to construct a new network and then use the techniques that we discussed above on this new network. Recall that after the refinement process we obtain a schedule with relative congestion 1 and frame size $O(1)$. Hence we can choose a constant w that is an upper bound on this frame size. We construct a new network

by replacing each edge of the old network by $2w$ edges in the new network. We then carry out all the techniques described earlier to obtain a schedule with relative congestion 1 and frame size w *on the new network.* This means that in any interval of w time steps at most w packets cross any edge. We divide time into intervals of length w and reschedule the at most w packets that cross an edge during this time so that at most 1 packet crosses any edge during any time step. For the new network this rescheduling means that we might now have a schedule in which packets cross edges out of order. Despite this, we can use it to construct a valid schedule for the original network. Suppose that edge e in the original network corresponds to $e_1, e_2, \ldots, e_{2w}$ in the new network. We schedule a packet to cross edge e in the original network at the same time as it is scheduled to cross edge e_{2w} in the new network. We show in Section 5.5 that this gives a legal schedule in which at most one packet crosses an edge in any time step and the packets cross the edges in order. The complete analysis will also show that the delay experienced by each packet in session i is $O(1/r_i + d_i)$.

5.3 Parameter Definitions

The remainder of this chapter contains the details of the proof that there exists a schedule with delay bound $O(1/r_i + d_i)$ and constant queue size. Our analysis makes use of a sizeable amount of notation. In this section we collect together all definitions for ease of reference.

5.3.1 The new network $\mathcal{M}$

Let $\mathcal{N}$ be the original network. In Section 5.5 we choose a constant w that is an upper bound on the frame size of the schedule that we obtain at the end of the refinement process. The new network $\mathcal{M}$ is constructed by replacing each edge of $\mathcal{N}$ by $2w$ edges. Each session i in $\mathcal{N}$ induces a corresponding session i in $\mathcal{M}$.

5.3.2 Interval lengths

Let $D_i = 2wd_i$. It is clear that D_i is the length of session i in $\mathcal{M}$. Let,

$$\mathcal{T}_i = 6D_i + 1 + (8/\varepsilon + 4)/r_i,$$

and let $\mathcal{T}$ be the smallest integer multiple of w that satisfies,

$$\mathcal{T} \geq (1 + 4/\varepsilon) \max_i \mathcal{T}_i.$$

We divide time into intervals of length $\mathcal{T}$ and independently construct schedules for the intervals $[0, \mathcal{T}), [\mathcal{T}, 2\mathcal{T})$, etc. During the interval $[k\mathcal{T}, (k+1)\mathcal{T})$ we assign tokens for the session-i packets that are injected during $[k\mathcal{T} - \mathcal{T}_i, (k+1)\mathcal{T} - \mathcal{T}_i)$. Throughout this proof we shall concentrate on assigning tokens for the packets that are injected during the interval $[\mathcal{T} - \mathcal{T}_i, 2\mathcal{T} - \mathcal{T}_i)$. (These tokens will be placed during the interval $[\mathcal{T}, 2\mathcal{T})$.) Clearly $\mathcal{T}_i = O(1/r_i + D_i)$. Since w is a constant we also have $\mathcal{T}_i = O(1/r_i + d_i)$.

5.3.3 Fractional packet size and parameters for the initial tokens

Let,

$$\begin{aligned} \ell_i &= \lceil 8/\varepsilon r_i \rceil, \\ s_i &= \lfloor \ell_i r_i (1 + \varepsilon/2) \rfloor. \end{aligned}$$

When session i is converted from fractional to integral we place s_i tokens in one time slot every ℓ_i slots. When session i is fractional the packets have size,

$$\hat{r}_i = \frac{s_i}{\ell_i}.$$

Note that,

$$\begin{aligned}\hat{r}_i &\geq \frac{\ell_i r_i(1+\varepsilon/2)-1}{\ell_i} \\ &= r_i(1+\varepsilon/2)-1/\ell_i \\ &\geq r_i(1+\varepsilon/4)+\varepsilon r_i/4-1/\ell_i \\ &\geq r_i(1+\varepsilon/4)+1/\ell_i.\end{aligned}$$

The last inequality comes from the fact that $\ell_i \geq 8/\varepsilon r_i$. We also have that,

$$\hat{r}_i \leq r_i(1+\varepsilon/2).$$

Hence,

$$r_i(1+\varepsilon/4)+1/\ell_i \leq \hat{r}_i \leq r_i(1+\varepsilon/2).$$

For some edge e let E be the set of sessions that pass through edge e.

Lemma 5.3.1 *For all edges e, $\sum_{i\in E}\hat{r}_i < 1-\varepsilon/2$.*

Proof:

$$\begin{aligned}\sum_{i\in E}\hat{r}_i &\leq \sum_{i\in E} r_i(1+\varepsilon/2) \\ &\leq (1-\varepsilon)(1+\varepsilon/2) \\ &= 1-\varepsilon/2-\varepsilon^2/2 \\ &\leq 1-\varepsilon/2.\end{aligned}$$

□

5.3.4 Parameters for refinement and conversion

Let $S^{(q)}$ be the schedule that is constructed after the qth refinement and conversion steps. Let $X_i = D_i + 1/r_i$ and let $X_{\max} = \max_i X_i$. We shall show in Section 5.4 that $S^{(q)}$ has relative congestion $c^{(q)}$ and frame size $I^{(q)}$ where $c^{(q)}$ and $I^{(q)}$ are given by

the following recurrences. The parameter β is a positive constant that will be chosen later. Let,

$$\begin{aligned} I^{(0)} &= e^{\log^{2/5} X_{\max}}, \\ I^{(q+1)} &= \log^5 I^{(q)}, \\ \delta^{(q)} &= \frac{\beta}{\sqrt{\log I^{(q)}}} \\ c^{(0)} &= 1 - \varepsilon/2, \\ c^{(q+1)} &= (1+\delta^{(q)})^2 c^{(q)}. \end{aligned}$$

Notice that $I^{(q)}$ decreases polylogarithmically with q whereas for large $I^{(q)}$, $c^{(q+1)}$ is only slightly larger than $c^{(q)}$.

We use $A^{(q)}$ to denote the set of integral sessions at the end of the qth conversion and refinement steps. We use $B^{(q)}$ to denote the set of sessions that become integral during the qth conversion step. The refinement process ends when $q = \zeta$ for some parameter ζ to be chosen later. We define $A^{(q)}$ and $B^{(q)}$ by,

$$\begin{aligned} A^{(0)} &= \emptyset, \\ A^{(q+1)} &= A^{(q)} \cup B^{(q+1)}, \\ B^{(q+1)} &= \{i \notin A^{(q)} : \left(I^{(q+1)}\right)^2 \leq X_i \leq e^{\sqrt{I^{(q+1)}}}\} \text{ for } q < \zeta - 1, \\ B^{(q+1)} &= \{i \notin A^{(q)} : X_i \leq e^{\sqrt{I^{(q+1)}}}\} \text{ for } q = \zeta - 1. \end{aligned}$$

Lemma 5.3.2 *The sets $B^{(q)}$ form a partition of all the sessions.*

Proof: It is clear that if $q \neq q'$ then $B^{(q)} \cap B^{(q')} = \emptyset$. Also, $I^{(1)} = \log^2 X_{\max}$ by definition which implies that $X_{\max} = e^{\sqrt{I^{(1)}}}$. Hence for all sessions i there exists a q such that $X_i \leq e^{\sqrt{I^{(q+1)}}}$. This means that if the $B^{(q)}$ do not form a partition then there must be some q for which $e^{\sqrt{I^{(q+1)}}} < \left(I^{(q)}\right)^2$. But,

$$e^{\sqrt{I^{(q+1)}}} = e^{\log^{5/2} I^{(q)}} = \left(I^{(q)}\right)^{\log^{3/2} I^{(q)}}.$$

We shall ensure that at all stages of the refinement process we have $I^{(q)} > e^{\sqrt[3]{4}}$. (See Section 5.4.4.) This expression is equivalent to $\log^{3/2} I^{(q)} > 2$ which implies that $e^{\sqrt{I^{(q+1)}}} > \left(I^{(q)}\right)^2$. By the above observations this completes the proof of the lemma. □

Note that the sessions with large X_i values become integral first. We make use of the bound $X_i \geq \left(I^{(q+1)}\right)^2$ in the refinement step and we use the bound $\log^2 X_i \leq I^{(q+1)}$ in the conversion step.

5.4 A Schedule for the Intermediate Network $\mathcal{M}$

In this section we describe an initial schedule $\mathcal{S}^{(0)}$ and the successive iterations needed to transform it to a schedule $\mathcal{S}^{(\zeta)}$ that satisfies Theorem 5.4.1 below. Each intermediate schedule $\mathcal{S}^{(q)}$ is produced by applying the refinement and conversion steps to the schedule $\mathcal{S}^{(q-1)}$. All of these schedules are defined on the network $\mathcal{M}$.

Theorem 5.4.1 *Given a set of sessions defined on the network $\mathcal{M}$ there is a schedule $\mathcal{S}^{(\zeta)}$ with the following properties.*

1. *The relative congestion is at most 1 for any frame of size larger than some constant;*

2. *After leaving its source, each packet waits at most once every $O(1)$ steps, which implies that all edge queues in $\mathcal{M}$ have size $O(1)$;*

3. *For all sessions i, each session-i packet reaches its destination within $O(1/r_i + D_i)$ steps of its injection;*

4. *All session-i packets that are injected during $[\mathcal{T} - \mathcal{T}_i, 2\mathcal{T} - \mathcal{T}_i)$ are scheduled during $[\mathcal{T}, 2\mathcal{T})$, i.e. each packet leaves its source no earlier than $\mathcal{T}$ and reaches its destination before $2\mathcal{T}$.*

The proof of this theorem is given at the end of Section 5.4.

We first define or recall several basic concepts. Given some schedule $\mathcal{S}$, a *region* R of the schedule is some interval of contiguous time steps in the schedule. A *T-frame*

is a region of length T. The *congestion* C in a T-frame is the maximum number of packets that cross any edge in that interval and the *relative congestion* is the ratio C/T. If session i is fractional then it always contributes exactly $\hat{r}_i$ to the relative congestion of any frame.

5.4.1 An Initial Schedule $\mathcal{S}^{(0)}$

In $\mathcal{S}^{(0)}$, all sessions are fractional, i.e. $A^{(0)} = \emptyset$. Each packet (of a fractional size) crosses one edge per time step whether or not other packets are using the same edge at the same time. Since the relative congestion is entirely due to fractional sessions, the relative congestion is at most $\sum r_i < 1 - \varepsilon/2 = c^{(0)}$ on any edge e (see Lemma 5.3.1).

Note that the above relative congestion holds for any frame size. We choose the initial frame size $I^{(0)} = e^{\log^{2/5} X_{\max}}$, so that $I^{(1)} = \log^2 X_{\max}$, which implies $X_{\max} = e^{\sqrt{I^{(1)}}}$. This allows the sessions with the largest X_i value to be converted in the first iteration of the algorithm (see the definition of $B^{(1)}$).

5.4.2 The Refinement Step

In this section we describe the refinement step. For each schedule, the packets from integral sessions are delayed in a way that dramatically reduces the frame size, but does not increase the relative congestion and the length of the schedule by much.

To be more precise, for schedule $\mathcal{S}^{(q)}$, we inductively assume that the relative congestion is at most $c^{(q)}$ for frames of size $I^{(q)}$ or larger and that each integral packet waits at most once every $I^{(q-1)}$ steps after leaving its source. In this frame refinement step we show that there is a way to delay (by an amount related to the frame size) the packets from $A^{(q)}$ so that, in the resulting schedule $\mathcal{S}^{(q+\frac{1}{2})}$, the relative congestion is at most $(1 + \delta^{(q)})c^{(q)}$ for any frame of size $I^{(q+1)} = \log^5 I^{(q)}$ or larger, where $\delta^{(q)} = \beta/\sqrt{\log I^{(q)}}$. *In addition each integral packet waits at most once every* $I^{(q)}$ *steps.*

The initial schedule $\mathcal{S}^{(0)}$ is described in Section 5.4.1. Since there are no integral sessions, no delays are inserted in this step. Clearly, the resulting relative congestion

is at most $(1+\delta^{(0)})c^{(0)}$ for any frame of size $I^{(1)}$ or larger at the end of this step, and no packet ever waits.

We now describe how to refine the schedule $\mathcal{S}^{(q)}$, for $q > 0$. The refinement is divided into two steps. In the first refinement step we divide the current schedule into blocks of length $2(I^{(q)})^3 + 2(I^{(q)})^2 - I^{(q)}$, and insert delays into each block so that its length increases to $2(I^{(q)})^3 + 2(I^{(q)})^2$. We show that these delays can be introduced in such a way that in the central $2(I^{(q)})^3$ steps of each block the relative congestion of frames of at least length $I^{(q+1)}$ is only a little larger than $c^{(q)}$. (See Figure 5-2.) At the beginning and end of each block there are "fuzzy" regions of length $(I^{(q)})^2$ each. In the second step we move the block boundaries so that the fuzzy regions at the end and beginning of adjacent blocks are at the center of the new blocks of $2(I^{(q)})^3 + 2(I^{(q)})^2$ steps. Again, we insert delays into each block increasing the size of the block by $(I^{(q)})^2$ steps. We show that there is a way to insert these delays so that the final conditions for refining $\mathcal{S}^{(q)}$ are indeed satisfied. (See Figure 5-3.)

In the following we present Lemma 5.4.3 that will be used extensively in both steps of the refinement. We continue by presenting both steps in detail.

A Useful Lemma

The following lemma is used to prove Lemma 5.4.3.

Lemma 5.4.2 *Let X and Y be independent random variables. Let Y be binomially distributed with mean μ_y, and let σ_1, σ_2, and v be values such that $\sigma_2 = \sigma_1 - 1/v$. Then,*

$$\Pr[\, X + \mu_y > (1+\sigma_1)v \,] \leq 2\Pr[\, X + Y > (1+\sigma_2)v \,].$$

Proof: Let $w = (1+\sigma_1)v - \mu_y$. We have,

$$\Pr[\, X + \mu_y > (1+\sigma_1)v \,] = \Pr[\, X > w \,], \tag{5.1}$$

$$\Pr[\, X + Y > (1+\sigma_2)v \,] = \Pr[\, X + Y > \mu_y + w - 1 \,]. \tag{5.2}$$

Note also that,

$$\begin{aligned}\Pr[\, X+Y > \mu_y + w - 1\,] &\geq \Pr[\, X > \mu_y + w - 1 - \lfloor \mu_y \rfloor \text{ and } Y \geq \lfloor \mu_y \rfloor\,] \\ &= \Pr[\, X > w - 1 + \mu_y - \lfloor \mu_y \rfloor\,] \Pr[\, Y \geq \lfloor \mu_y \rfloor\,].\end{aligned}$$

This last equality follows from the independence of X and Y. Theorem B.1 in [46] shows that $\Pr[\, Y \geq \lfloor \mu_y \rfloor\,] \geq 1/2$. Since $\mu_y - \lfloor \mu_y \rfloor < 1$, we have,

$$\Pr[\, X+Y > \mu_y + w - 1\,] \geq \frac{1}{2} \Pr[\, X > w\,].$$

Our Lemma follows from equalities (5.1) and (5.2) and the above inequality. □

We say that a packet is *active* during some region of a schedule if the packet belongs to some integral session and it traverses at least one edge during the region. Recall that we are maintaining the following invariant. In the schedule $\mathcal{S}^{(q)}$ a packet waits at most once every $O(I^{(q-1)})$ steps after leaving its source. Hence an *inactive* packet is either at its source or its destination during the entire region. Lemma 5.4.3 below is a stepping stone that allows us to reduce the frame size from $I^{(q)}$ to *poly* $\log I^{(q)}$. We invoke this lemma for various values of s, t, r and R.

Lemma 5.4.3 *Consider some region R of a schedule where the relative congestion is at most $r = \Theta(1)$ for frames of length s, where $\log^3 I^{(q)} \leq s \leq (I^{(q)})^2$. Consider any edge e and any t-frame, where $\log^2 I^{(q)} \leq t \leq 2\log^2 I^{(q)}$. Assume that each active packet in the region is delayed between the beginning of R and the beginning of the t-frame by a number of steps randomly, independently, and uniformly chosen from $[1, s]$. Then, for any constant k there is some value $\gamma = \Theta(1)/\sqrt{\log I^{(q)}}$ such that if $I^{(q)}$ is large, the probability of having relative congestion greater than $r(1+\gamma)$ on e during the t-frame is at most $(I^{(q)})^{-k}$.*

Proof: Let the random variable X be the frame congestion on e during the t-frame due to the active packets after they have been delayed. If the relative congestion due to fractional sessions is r_f, the frame congestion due to fractional sessions in the t-frame is exactly $r_f t$. Since the active packets are the only integral-session packets

that can cross e during the region, the frame congestion on e during the t-frame is $X + r_f t$ after the packets have been delayed.

Consider now a binomial random variable Y with parameters $(r_f s, t/s)$ and mean $E[Y] = r_f t$. From Lemma 5.4.2, the probability p that the congestion in the t-frame is larger than $(1+\gamma)rt$ after the packets have been delayed is,

$$p = \Pr[\, X + r_f t > (1+\gamma)rt \,] \leq 2\Pr[\, X + Y > (1+\sigma)rt \,],$$

where $\sigma = \gamma - 1/rt$. Since $t \geq \log^2 I^{(q)}$ and $r = \Theta(1)$, then $\gamma = \Theta(1)/\sqrt{\log I^{(q)}}$ if and only if $\sigma = \Theta(1)/\sqrt{\log I^{(q)}}$. Let $\sigma = v/\sqrt{\log I^{(q)}}$, where v is a large constant. We shall choose an appropriate value v so that the lemma is satisfied.

We first concentrate on X. Since the active packets are delayed up to s steps, an active packet that crosses e in the t-frame after the delay insertion could cross e in an interval of $t + s$ steps before the delay insertion. The relative congestion due to active packets is at most $r - r_f$ in that interval before the delay insertion. Hence, at most $(t+s)(r-r_f)$ active packets can cross e in the t-frame after the delay insertion, and each of them has probability of at most t/s of doing so.

Recall that Y is a binomial random variable with parameters $(r_f s, t/s)$. We define Z to be a binomial random variable with parameters $(n, t/s)$, where $n = r(t+s) > (r - r_f)(t+s) + r_f s$. It is easy to see that,

$$p \leq 2\Pr[\, X + Y > (1+\sigma)rt \,] \leq 2\Pr[\, Z > (1+\sigma)rt \,].$$

Therefore, we can bound probability p by,

$$p \leq 2 \sum_{i=(1+\sigma)rt}^{r(t+s)} \binom{r(t+s)}{i} (t/s)^i (1 - t/s)^{r(t+s)-i}.$$

We bound the sum by observing that $(1+\sigma)rt$ is larger than $E[Z] = (t+s)rt/s$, since $t/s \leq 2/\log I^{(q)}$. Thus, the first term of the sum is the largest. By applying the inequalities $(1+x) \leq e^x$ for all x, $\ln(1+x) \geq x - x^2/2$ for $0 \leq x \leq 1$, and

$\binom{a}{b} \leq (ae/b)^b$ for $0 < b < a$, and the fact that there are at most $r(t+s)$ terms in the sum, we have,

$$\begin{aligned} p &\leq 2r(t+s)\left(\frac{r(t+s)e}{(1+\sigma)rt}\right)^{(1+\sigma)rt}\left(\frac{t}{s}\right)^{(1+\sigma)rt}\left(1-\frac{t}{s}\right)^{rs-\sigma rt} \\ &= 2r(t+s)\left(\frac{\left(1+\frac{t}{s}\right)e}{1+\sigma}\right)^{(1+\sigma)rt}\left(1-\frac{t}{s}\right)^{rs-\sigma rt} \\ &\leq 2r(t+s)e^{(1+\sigma)rt(1+t/s-\sigma+\sigma^2/2)-rt+\sigma rt^2/s} \\ &= 2r(t+s)e^{-rt\sigma^2(1/2-\sigma/2-t/\sigma^2 s-2t/\sigma s)}. \end{aligned}$$

The bounds on s and t and the definitions of r and σ imply that we can choose a constant v large enough so that $2r(t+s)e^{-rt\sigma^2/4} < (I^{(q)})^{-k}$ for any constant $k > 0$. For values of $I^{(q)}$ larger than some other constant (dependent on v) we have $1/2 - \sigma/2 - t/\sigma^2 s - 2t/\sigma s \geq 1/4$. Hence we can choose v large enough so that $p < (I^{(q)})^{-k}$ when $I^{(q)}$ is large. □

The First Refinement Step for Schedule $\mathcal{S}^{(q)}$

We first divide the interval $[\mathcal{T}, \mathcal{T}+|\mathcal{S}^{(q)}|)$ into blocks of length $2(I^{(q)})^3+2(I^{(q)})^2-I^{(q)}$. We shall reschedule each block B independently. During a block B we only delay active packets.

For each block B, each active packet in B is assigned a delay randomly, uniformly, and independently chosen from $[1, I^{(q)}]$. An active packet p, whose assigned delay is x, is delayed in the first $xI^{(q)}$ steps of B once every $I^{(q)}$ steps. In order to independently reschedule the next block, packet p is also delayed in the last $(I^{(q)}-x)I^{(q)}$ steps of B once every $I^{(q)}$ steps. Therefore, a rescheduled block has length $2(I^{(q)})^3 + 2(I^{(q)})^2$.

Before the delays are inserted to reschedule block B, an active packet p is delayed at most once within the block, provided that $2(I^{(q)})^3 + 2(I^{(q)})^2 - I^{(q)} < I^{(q-1)}$, which holds as long as $I^{(q)}$ is larger than some constant. In Section 5.4.4 we ensure that $I^{(q)}$ is large enough. Prior to inserting any new delay into a block, we check if it is within $I^{(q)}$ steps of the single old delay. If the new delay would be too close to the old delay, then it is simply not inserted. The loss of one delay in a block has a negligible effect

on the probability analysis that follows.

Lemma 5.4.5 shows that with the insertion of delays we can dramatically reduce the frame size in the center of the block and increase the relative congestion only slightly. In order to prove Lemma 5.4.5, we need the following fact.

Lemma 5.4.4 *If the relative congestion in every frame of size T to $2T-1$ is at most r, then the relative congestion in any frame of size T or greater is at most r.*

Proof: Consider a frame of size T', where $T' > 2T-1$. The first $\lfloor T'/T \rfloor T - T$ steps of the frame can be broken into T-frames, each with relative congestion r. The remainder of the T'-frame consists of a single frame of size between T and $2T-1$ steps in which the relative congestion is also at most r. □

Lemma 5.4.5 *There exists a way of choosing delays so that in between the first and last $(I^{(q)})^2$ steps of the block B, the relative congestion of any frame of size $\log^2 I^{(q)}$ or larger is at most $(1+\gamma_1)c^{(q)}$, for some $\gamma_1 = \Theta(1)/\sqrt{\log I^{(q)}}$.*

Proof: With each edge e, we associate a bad event. A bad event on e happens when the frame congestion on edge e is more than $(1+\gamma_1)c^{(q)}I$ during *any* I-frame of size $\log^2 I^{(q)}$ or larger. Due to Lemma 5.4.4, it is sufficient to prove the statement for all frames of size between $\log^2 I^{(q)}$ and $2\log^2 I^{(q)}$. We shall use the Lovász Local Lemma to show that the probability that no bad event occurs is nonzero. (See Section 4.1.5 for the statement of the Lovász Local Lemma.)

We first bound the dependence, d, of bad events. Two bad events on two edges are dependent only if a packet from a session $i \in A^{(q)}$ can use both edges. At most $c^{(q)}(2(I^{(q)})^3 + 2(I^{(q)})^2$ packets (from sessions in $A^{(q)}$) can cross the same edge in the block B, and each packet crosses at most $2(I^{(q)})^3 + 2(I^{(q)})^2 - I^{(q)}$ edges in B. As we shall show later, $c^{(q)} \leq 1$. Therefore, a bad event can be dependent on at most $O((I^{(q)})^6)$ other bad events.

We now bound the probability, p, that a bad event happens on e. Consider a particular I-frame, where $\log^2 I^{(q)} \leq I \leq 2\log^2 I^{(q)}$, that lies completely between the first and last $(I^{(q)})^2$ steps of B. By setting $R = B$, $r = c^{(q)}$, $s = I^{(q)}$ and $t = I$,

we can apply Lemma 5.4.3 to show that for any constant k_1 there is some value $\gamma_1 = \Theta(1)/\sqrt{\log I^{(q)}}$ such that the probability p_1 of a bad event happening on e in the I-frame is smaller than $(I^{(q)})^{-k_1}$.

Since there are $O((I^{(q)})^3 \log^2 I^{(q)})$ possible I-frames in B, the probability that a bad event happens on e during any I-frame is $p < p_1 O((I^{(q)})^3 \log^2 I^{(q)})$. We can set the value k_1 appropriately so that this probability is upper bounded by $O((I^{(q)})^{-7})$.

Therefore, we have $4pd < 1$ and our lemma follows from the Lovász Local Lemma.

□

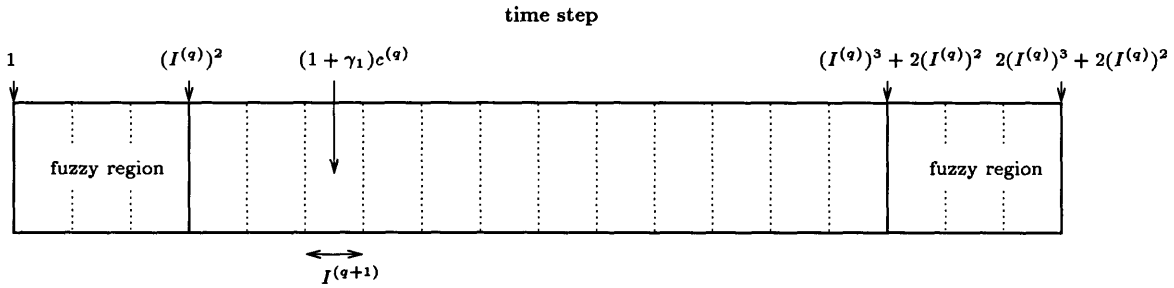

Figure 5-2: Situation after the first refinement step

At the end of the first refinement step, the center of each block has small relative congestion for small frame sizes. However there are regions of $(I^{(q)})^2$ steps at the beginning and end of each block that may have very large relative congestion. We call these "fuzzy" regions, and we deal with them in the second refinement step.

The Second Refinement Step for Schedule $\mathcal{S}^{(q)}$

We start the second step of the refinement by relocating the block boundaries so that blocks still have $2(I^{(q)})^3 + 2(I^{(q)})^2$ steps, but now the fuzzy regions that were at the beginning and end of adjacent blocks are in the center of new blocks. Then, a new block has two "clean" regions of $(I^{(q)})^3$ steps each at the beginning and the end, and a fuzzy region of length $2(I^{(q)})^2$ steps in the center.

As in the first step of the refinement we now concentrate on individual blocks. We first show that the relative congestion is not very large for frames of size $(I^{(q)})^2$ or larger (even in the fuzzy region).

Lemma 5.4.6 *For any choice of delays in the first step of the refinement, the relative congestion in any frame of size* $(I^{(q)})^2$ *or larger is at most* $(1 + 2/I^{(q)})c^{(q)}$.

Proof: Consider an I-frame with I_1 steps before the center of the block and I_2 steps after the center ($I = I_1 + I_2$, and either I_1 or I_2 could be zero). A packet crosses some edge e in the I_1-frame only if it did so in some frame of length $I_1 + I^{(q)}$ before the delays were inserted. (This is true for both fractional and integral packets.) Therefore, at most $(I_1 + I^{(q)})c^{(q)}$ packets can cross edge e in the I_1-frame. Similarly, at most $(I_2 + I^{(q)})c^{(q)}$ packets can cross edge e in the I_2-frame. Therefore, the congestion in the I-frame is at most $(I_1 + I_2 + 2I^{(q)})c^{(q)} = (I + 2I^{(q)})c^{(q)}$, and for $I \geq (I^{(q)})^2$ the relative congestion is at most $(1 + 2/I^{(q)})c^{(q)}$. □

Now, in order to reduce the frame size in the fuzzy region, we consider only the active packets in each block B, and assign a delay randomly, independently, and uniformly chosen from $[1, (I^{(q)})^2]$ to each active packet. A packet p with delay x waits once every $(I^{(q)})^3/x$ at the beginning of the block and once every $(I^{(q)})^3/((I^{(q)})^2 - x)$ at the end. As in the first step a delay is not inserted if it is going to be within $I^{(q)}$ steps of an existing delay for a moving packet.

The block length after the delay insertion is $2(I^{(q)})^3 + 3(I^{(q)})^2$, and the fuzzy region can be $(I^{(q)})^2$ steps longer, spanning steps $(I^{(q)})^3$ to $(I^{(q)})^3 + 3(I^{(q)})^2$.

The next lemma shows that there is some way of inserting delays so that the frame size in the fuzzy region is reduced, and the frame size and relative congestion in the rest of the block are increased by only a small amount.

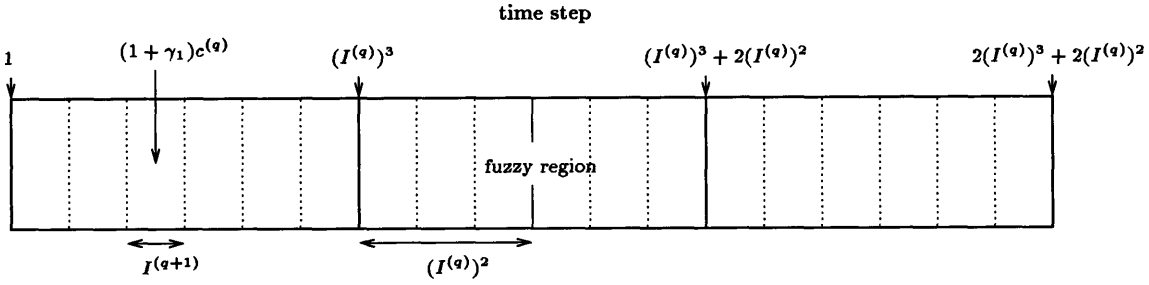

Figure 5-3: Situation after relocating block boundaries

Lemma 5.4.7 *In a block B, there exists a way of choosing delays so that in the fuzzy region (i.e. interval $[(I^{(q)})^3, (I^{(q)})^3 + 3(I^{(q)})^2]$) the relative congestion of any frame of*

size $\log^2 I^{(q)}$ *or larger is at most* $(1+\gamma_2)c^{(q)}$, *for some* $\gamma_2 = \Theta(1)/\sqrt{\log I^{(q)}}$, *and so that in the intervals* $[I^{(q)}\log^3 I^{(q)}, (I^{(q)})^3]$ *and* $[(I^{(q)})^3 + 3(I^{(q)})^2, 2(I^{(q)})^3 + 3(I^{(q)})^2 - I^{(q)}\log^3 I^{(q)}]$ *the congestion of any frame of size* $\log^2 I^{(q)}$ *or larger is at most* $(1+\gamma_3)c^{(q)}$, *for some* $\gamma_3 = \Theta(1)/\sqrt{\log I^{(q)}}$.

Proof: As in Lemma 5.4.5, we shall use the Lovász Local Lemma to prove the claim. We associate a bad event with every edge e, so that a bad event happens on e if, for any $I \geq \log^2 I^{(q)}$,

- more than $(1+\gamma_2)c^{(q)}I$ packets cross e in any I-frame in $[(I^{(q)})^3, (I^{(q)})^3 + 3(I^{(q)})^2]$ (the fuzzy region), or
- more than $(1+\gamma_3)c^{(q)}I$ packets cross e in any I-frame in either $[I^{(q)}\log^3 I^{(q)}, (I^{(q)})^3]$ or $[(I^{(q)})^3 + 3(I^{(q)})^2, 2(I^{(q)})^3 + 3(I^{(q)})^2 - I^{(q)}\log^3 I^{(q)}]$.

The dependency, d, of the bad events is bounded as in Lemma 5.4.5. Two bad events on two edges are dependent only if packets from some session $i \in A^{(q)}$ use both edges. At most $O((I^{(q)})^3)$ packets cross any edge in a block, and each of them can cross at most $O((I^{(q)})^3)$ other edges. Therefore, $d = O((I^{(q)})^6)$.

Now, to bound the probability p of a bad event happening on some edge e, we consider the three intervals separately and sum their respective probabilities. From Lemma 5.4.4 we only consider frames of length I such that $\log^2 I^{(q)} \leq I \leq 2\log^2 I^{(q)}$.

Take first some I-frame in $[(I^{(q)})^3, (I^{(q)})^3 + 3(I^{(q)})^2]$ (the fuzzy region). From Lemma 5.4.6 we know that the relative congestion for frames of size $(I^{(q)})^2$ or longer is at most $(1 + 2/I^{(q)})c^{(q)} = \Theta(1)$. Then, by choosing $R = B$, $r = (1 + 2/I^{(q)})c^{(q)}$, $s = (I^{(q)})^2$, and $t = I$, we can use Lemma 5.4.3 to show that, for any constant k_2, there is some $\sigma_2 = \Theta(1)/\sqrt{\log I^{(q)}}$ such that the probability of having relative congestion on e in the I-frame larger than $c^{(q)}(1 + 2/I^{(q)})(1 + \sigma_2) = c^{(q)}(1 + \gamma_2)$ is smaller than $(I^{(q)})^{-k_2}$. Note that $\gamma_2 = \Theta(1)/\sqrt{\log I^{(q)}}$.

Take now some I-frame in $[I^{(q)}\log^3 I^{(q)}, (I^{(q)})^3]$, which starts at step j. Given the way delays are inserted, by the jth step an active packet with delay x has been delayed $jx/(I^{(q)})^3$ steps. Thus, the delay of an active packet at the jth step is

essentially a random value uniformly chosen from $[1, j/I^{(q)}]$. For $j \geq I^{(q)} \log^3 I^{(q)}$ the value $j/I^{(q)} \geq \log^3 I^{(q)}$.

Note that before the delays were inserted, Lemma 5.4.5 implies that the relative congestion in any frame of length $\log^2 I^{(q)}$ or larger in the interval $[1, (I^{(q)})^3]$ was at most $(1+\gamma_1)c^{(q)}$. Therefore, we can set $R = [1, (I^{(q)})^3]$, $r = (1+\gamma_1)c^{(q)}$, $s = \log^3 I^{(q)}$, and $t = I$, and use Lemma 5.4.3 to show, for any constant k_3, the existence of some $\sigma_3 = \Theta(1)/\sqrt{\log I^{(q)}}$ such that the probability of having relative congestion larger than $(1+\sigma_3)(1+\gamma_1)c^{(q)} = (1+\gamma_3)c^{(q)}$ on e in the I-frame is smaller than $(I^{(q)})^{-k_3}$. Again, $\gamma_3 = \Theta(1)/\sqrt{\log I^{(q)}}$.

By symmetry, the same value γ_3 makes the probability of a bad event happening on e in some I-frame in $[(I^{(q)})^3 + 3(I^{(q)})^2, 2(I^{(q)})^3 + 3(I^{(q)})^2 - I^{(q)} \log^3 I^{(q)}]$ smaller than $(I^{(q)})^{-k_3}$.

There are $O((I^{(q)})^3 \log^2 I^{(q)})$ possible I-frames in the intervals that we have considered. Hence, we can choose values for k_2 and k_3 such that the probability of a bad event is bounded by $p \leq (p_1 + 2p_2)O((I^{(q)})^3 \log I^{(q)}) < O((I^{(q)})^7)$. Therefore, we can guarantee $4pd < 1$ and invoke the Lovász Local Lemma to prove the claim. □

Finally, we bound the frame size and the relative congestion in the remaining intervals of the block in the following lemma.

Lemma 5.4.8 *The relative congestion in any frame of size* $\log^4 I^{(q)}$ *or larger in the intervals* $[1, I^{(q)} \log^3 I^{(q)}]$ *and* $[2(I^{(q)})^3 + 3(I^{(q)})^2 - I^{(q)} \log^3 I^{(q)}, 2(I^{(q)})^3 + 3(I^{(q)})^2]$ *is at most* $(1+\gamma_1)(1+1/\log I^{(q)})c^{(q)} = (1+\gamma_4)c^{(q)}$.

Proof: Let us first consider some I-frame in $[1, I^{(q)} \log^3 I^{(q)}]$. Recall that, before inserting delays, the relative congestion for frames of size $\log^2 I^{(q)}$ or more was at most $(1+\gamma_1)c^{(q)}$. In the interval no packet is delayed more than $\log^3 I^{(q)}$ steps. Therefore, the packets crossing some edge e in the I-frame could have crossed e in some interval of at most $I + \log^3 I^{(q)}$ steps. Hence the congestion in the I-frame can be at most $(I + \log^3 I^{(q)})(1+\gamma_1)c^{(q)}$. For $I \geq \log^4 I^{(q)}$ the claim follows. The proof for interval $[2(I^{(q)})^3 + 3(I^{(q)})^2 - I^{(q)} \log^3 I^{(q)}, 2(I^{(q)})^3 + 3(I^{(q)})^2]$ is similar. □

From the above two lemmas we have that any frame of length at least $\log^4 I^{(q)}$

in each of the different intervals has relative congestion at most $(1+\gamma)c^{(q)}$, where $\gamma = \max(\gamma_2, \gamma_3, \gamma_4) = O(1)/\sqrt{\log I^{(q)}}$. We need to be careful now with the relative congestion in frames that overlap several intervals or several blocks. We can however say that for any frame of size $I^{(q+1)} = \log^5 I^{(q)}$ or larger, the relative congestion is at most $(1+\delta^{(q)})c^{(q)}$, for some $\delta^{(q)} = \beta/\sqrt{\log I^{(q)}}$, where β is a large constant. We use $\mathcal{S}^{(q+\frac{1}{2})}$ to denote the schedule that has now been produced.

5.4.3 The Conversion Step

In the conversion step we take the schedule $\mathcal{S}^{(q+\frac{1}{2})}$ that has been produced by the refinement step and transform it into the schedule $\mathcal{S}^{(q+1)}$. In this schedule we shall view the sessions in $B^{(q+1)}$ as integral sessions. The relative congestion of $\mathcal{S}^{(q+1)}$ in frames of size $I^{(q+1)}$ or larger is at most $(1+\delta^{(q)})^2 c^{(q)}$, where $\delta^{(q)} = \beta/\sqrt{\log I^{(q)}}$. (Recall that for the schedule $\mathcal{S}^{(q+\frac{1}{2})}$ the relative congestion is at most $(1+\delta^{(q)})c^{(q)}$ for any frame of size $I^{(q+1)}$ or larger.)

In the conversion step we apply the following two substeps to each session $i \in B^{(q+1)}$. (a) In the discretization substep we take the schedule for fractional session-i packets and convert it to a schedule in which all session-i packets reach their destinations quickly. (b) In the delay-insertion process we insert delays into the schedule in such a way that the relative congestion due to sessions in $B^{(q+1)}$ is not too large.

Discretization

We now describe how to transform a schedule for a fractional session into a schedule for an integral session. Recall that when we are providing a schedule for a fractional session we assume that an $\hat{r}_i$ fraction of a packet is injected at every time step. The schedule for the fractional session is extremely simple. Each fractional packet crosses one edge every time step until it reaches its destination. Clearly the relative congestion due to session i is $\hat{r}_i$ on any edge through which session i passes.

In the discretization step we provide a schedule for integral packets in which every packet reaches its destination without too much delay. The key problem is the placing

of *initial tokens* on the first edge of the session. When a session-i packet is injected it waits until all the other session-i packets have crossed the first edge and then waits for the first session-i initial token to appear. The packet crosses the first edge at this time. Packet-groups of tokens are then created in such a way that a packet can cross one edge every time step once it has crossed its initial edge. In other words, if there is a token for session i on its kth edge at time t then there is a session i token on its $(k+1)$st edge at time $t+1$.

In order to describe the distribution of initial tokens we consider the two intervals $U = [\mathcal{T} - \mathcal{T}_i, 2\mathcal{T} - \mathcal{T}_i)$ and $V = [\mathcal{T}, 2\mathcal{T} - \mathcal{T}_i)$. (See Figure 5-1.) The significance of these intervals is that all the packets that are injected during the interval U will be assigned an initial token during the interval V. Although the interval V is shorter than the interval U we shall ensure that the initial tokens appear within V at rate $\hat{r}_i$. Since packets are injected into session i at rate r_i and $r_i < \hat{r}_i$ we shall be able to bound the time that a packet must wait before it receives an initial token.

The exact method for assigning initial tokens is as follows. The interval V is divided into intervals of length ℓ_i. (See Figure 5-4.) If the length of V is not divisible by ℓ_i then the first such interval is incomplete. We then place s_i initial tokens at the end of each interval.

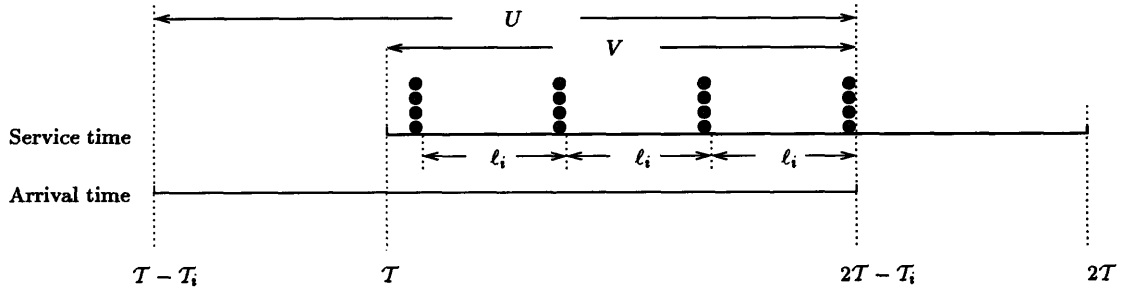

Figure 5-4: Session-i packets that are injected in interval U are assigned initial tokens in interval V. The interval V is divided into consecutive intervals of length ℓ_i, each of which has s_i initial tokens. The initial tokens are shown in solid dots.

Lemma 5.4.9 *Each session-i packet that is injected during U can cross the first edge on the session within $\mathcal{T}_i + \ell_i = O(1/r_i + D_i)$ steps of its injection.*

Proof: Let $x = \frac{\mathcal{T}}{\mathcal{T} - \mathcal{T}_i}$ be the ratio of the length of the interval U to the length of

the interval V. We first show that s_i is larger than the number of session-i packets that can arrive during an interval of length $x\ell_i$. By the definition of injection rate no more than $n = x\ell_i r_i + 1$ session-i packets can be injected during $x\ell_i$ time steps. By the definition of $\mathcal{T}$ and $\mathcal{T}_i$ we have $\mathcal{T} \geq (1 + 4/\varepsilon)\max_i \mathcal{T}_i$. Hence $x \leq 1 + \varepsilon/4$ which implies that $n \leq s_i$ by the definition of s_i.

Recall that we divided V into intervals with boundaries,

$$\mathcal{T}, 2\mathcal{T} - \mathcal{T}_i - \alpha\ell_i, 2\mathcal{T} - \mathcal{T}_i - (\alpha - 1)\ell_i, \ldots, 2\mathcal{T} - \mathcal{T}_i,$$

for some α. Now divide U into intervals with boundaries,

$$\mathcal{T} - \mathcal{T}_i, 2\mathcal{T} - \mathcal{T}_i - \alpha x\ell_i, 2\mathcal{T} - \mathcal{T}_i - (\alpha - 1)x\ell_i, \ldots, 2\mathcal{T} - \mathcal{T}_i.$$

The above argument shows that all the packets injected during $[\mathcal{T} - \mathcal{T}_i, 2\mathcal{T} - \mathcal{T}_i - \alpha x\ell_i)$ can obtain an initial token during $[\mathcal{T}, 2\mathcal{T} - \mathcal{T}_i - \alpha\ell_i)$ and all the packets injected during $[2\mathcal{T} - \mathcal{T}_i - kx\ell_i, 2\mathcal{T} - \mathcal{T}_i - (k-1)x\ell_i)$ can obtain an initial token during $[2\mathcal{T} - \mathcal{T}_i - k\ell_i, 2\mathcal{T} - \mathcal{T}_i - (k-1)\ell_i)$. This implies that all packets can obtain an initial token within,

$$\ell_i + \mathcal{T} - (\mathcal{T} - \mathcal{T}_i) = \ell_i + \mathcal{T}_i = O(1/r_i + D_i)$$

steps of their injection. □

Delay Insertion

The above discussion shows that we can provide a schedule for the new integral sessions in which all packets reach their destinations quickly. However, the relative congestion could now be large if many packets wish to cross an edge at the same time. The solution is to delay each initial token by an amount chosen uniformly at random from $[1, \ell_i]$. This has the effect of delaying the time at which we can be sure that a packet crosses the first edge. If an initial token κ is assigned a delay x then we delay all the tokens in the same packet-group as κ by an amount x.

Lemma 5.4.10 *Consider an edge e and a t-frame in the interval $[T, 2T)$. Suppose that session i passes through e and consider the conversion step in which session i becomes integral. Then the expected number of session-i packets that cross edge e during the t-frame after the delay insertion substep is at most $ts_i/\ell_i = \hat{r}_i$.*

Proof: Consider any time step. Before the delays are inserted the number of session-i packets that can cross edge e in this time step is at most s_i and each does so with probability $1/\ell_i$. Hence the expected number of packets that cross edge e during this time step is s_i/ℓ_i. By linearity of expectations the expected number of packets that cross edge e during the t-frame is ts_i/ℓ_i. □

Lemma 5.4.11 *The initial delays for sessions in $B^{(q+1)}$ can be chosen in such a way that the relative congestion in any frame of size $I^{(q+1)}$ or bigger is at most $c^{(q+1)}$ after the delays are inserted.*

Proof: We shall use a Chernoff bound and the Lovász Local Lemma. Due to Lemma 5.4.4 it is sufficient to prove the result for frames of size between $I^{(q+1)}$ and $2I^{(q+1)}$. In order to use the Lovász Local Lemma we must define a set of bad events. We associate a bad event $B_{\{e,I\}}$ with each edge e and each I-frame, where $I^{(q+1)} \leq I \leq 2I^{(q+1)}$. We define $B_{\{e,I\}}$ to be the event that more than $Ic^{(q+1)}$ packets use the edge e during the i-frame. In the analysis below we show that with non-zero probability no bad event occurs. Let,

$$\begin{aligned} D_{\max} &= \max_{i \in B^{(q+1)}} D_i, \\ r_{\min} &= \min_{i \in B^{(q+1)}} r_i, \\ \ell_{\max} &= \max_{i \in B^{(q+1)}} \ell_i, \\ X &= \max_{i \in B^{(q+1)}} D_i + \frac{1}{r_i}. \end{aligned}$$

Note that $D_{\max} = O(X)$, $r_{\min} = O(X)$ and $\ell_{\max} = O(X)$. To apply the Lovász Local Lemma we must first bound the dependency d between bad events. Note that the probability of a bad event occurring is dependent solely on the delays assigned to

packets from sessions in $B^{(q+1)}$. Hence, a bad event $B_{\{e',I'\}}$ is dependent on the bad event $B_{\{e,I\}}$ only if there is a packet p from a session $i \in B^{(q+1)}$ such that there is a non-zero probability that p uses e in the I-frame and a non-zero probability that p uses e' in the I'-frame.

Consider some bad event $B_{\{e,I\}}$. We now bound the number of bad events $B_{\{e',I'\}}$ that can be dependent on it. There are at most $1/r_{\min}$ sessions in $B^{(q+1)}$, each of which is at most $D_{\max}$ long. Therefore there are at most $D_{\max}/r_{\min} = O(x^2)$ choices for e'. Furthermore, the intervals I and I' cannot be more than $D_{\max} + \ell_{\max}$ steps apart. (Otherwise any session-i packet either has probability 0 of crossing e during I or probability 0 of crossing e' during I'.) Therefore, the starting point of I' can only be in one of $2D_{\max} + 2\ell_{\max} + 4I^{(q+1)}$ locations which implies that the total number of possible choices for I' is at most $(2D_{\max} + 2\ell_{\max} + 4I^{(q+1)})I^{(q+1)} = O(X(I^{(q+1)})^2)$. This means that there are at most $O(X^3(I^{(q+1)})^2)$ bad events $B_{\{e',I'\}}$ that depend on $B_{\{e,I\}}$. We conclude that the dependency d of the bad events is $O(X^3(I^{(q+1)})^2)$.

It remains to bound the probability that a bad event $B_{\{e,I\}}$ happens. By our assumptions about the schedule $\mathcal{S}^{(q+\frac{1}{2})}$ we know that before the conversion step the frame congestion on edge e during the I-frame is at most $(1+\delta^{(q)})c^{(q)}I$. Let S be the set of sessions in $B^{(q+1)}$ that use edge e. When the sessions in $B^{(q+1)}$ are fractional they contribute exactly $I\sum_{i\in S}\hat{r}_i$ to the frame congestion. Lemma 5.4.11 implies that the expected frame congestion due to the sessions in $B^{(q+1)}$ is at most $I\sum_{i\in S}\hat{r}_i$ after the delays are inserted. The congestion due to sessions not in $B^{(q+1)}$ does not change during the conversion. Hence, if we denote by μ the expected frame congestion on edge e during the I-frame then $\mu \leq (1+\delta^{(q)})c^{(q)}I$. Note that by adding extra "ghost" packets that cross edge e during the I-frame with probability 1 we can ensure that $\mu \geq \frac{1}{2}(1+\delta^{(q)})c^{(q)}I$. This only increases the probability that the bad event $B_{\{e,I\}}$ happens.

Now let P be the set of packets that have a non-zero probability of crossing edge e during the I-frame. For each packet $p \in P$ let Z_p be a Bernoulli random variable

that is equal to 1 if and only if p crosses edge e during the I-frame. We have,

$$\begin{aligned}
\Pr[B_{\{e,I\}} \text{ occurs}] &= \Pr[\text{ Frame congestion on } e \text{ in } I > c^{(q+1)}I \,] \\
&\leq \Pr[\text{ Frame congestion on } e \text{ in } I > (1+\delta^{(q)})\mu \,] \\
&\leq e^{-(\delta^{(q)})^2\mu/3} \\
&\leq e^{-(1-\varepsilon)\beta^2 I^{(q+1)}/2\log I^{(q)}} \\
&\leq e^{-(1-\varepsilon)\frac{\beta^2}{6}(I^{(q+1)})^{1/5}(I^{(q+1)})^{3/5}} \\
&\leq e^{-(1-\varepsilon)\frac{\beta^2}{6}(I^{(q+1)})^{1/5}\log^{6/5} X}.
\end{aligned}$$

The first inequality comes from the fact that $c^{(q+1)} = (1+\delta^{(q)})^2 c^{(q)}$ and $\mu \leq (1+\delta^{(q)})c^{(q)}I$. The second inequality follows from a Chernoff Bound. The third inequality holds since $\mu \geq \frac{1}{2}(1-\varepsilon)I \geq \frac{1}{2}(1-\varepsilon)I^{(q+1)}$ and $\delta^{(q)} = \frac{\beta}{\sqrt{\log I^{(q)}}}$. The fourth inequality follows from the equation $\log I^{(q)} = (I^{(q+1)})^{1/5}$. The last inequality comes from the fact that $\log^2 X \leq I^{(q+1)}$. (This explains the need for $\log^2 X_i \leq I^{(q+1)}$ in the definition of $B^{(q+1)}$.)

When β is a sufficiently large constant we have $4dp < 1$. By the Lovász Local Lemma this implies that with nonzero probability no bad events occur. Therefore there exists a way to choose the initial delays for sessions in $B^{(q+1)}$ such that for all frames of size $I^{(q+1)}$ or larger the relative congestion is at most $c^{(q+1)}$. Hence there exists a schedule, which we call $\mathcal{S}^{(q+1)}$, with frame size $I^{(q+1)}$ and relative congestion $c^{(q+1)}$. □

Note that in the proof of the above lemma we associate a bad event with each edge e and each interval I. Recall that in the analysis of the refinement step we started by considering one particular interval and a bad event was associated with an edge only. We then used a union bound to obtain a result for all intervals. The reason that we cannot do this for the conversion step is that we must consider the whole interval $\mathcal{T}$. For some sessions i, however, X_i could be much smaller than $\mathcal{T}$ and so we would have many intervals to consider. The probability of failure for each interval would not be small enough to allow us to use a union bound.

5.4.4 The Termination of the Algorithm

Let x be the smallest constant that satisfies the following conditions.

1. $(1 - \frac{\beta}{\sqrt{\log x}})^2 \geq 1 - \frac{\epsilon}{2}$, i.e. $x \geq e^{\beta^2/(1-\sqrt{1-\frac{\epsilon}{2}})^2}$.
2. $\frac{\beta}{\log\log^5 x} > \frac{\beta}{\sqrt{x}}$.
3. $\log^5 x < \frac{x}{2}$.
4. $\log^5 x > e^{\sqrt[3]{4}}$.
5. $2(\log^5 x)^3 + 2(\log^5 x)^2 - \log^5 x < x$.
6. If $I^{(q)} \geq x$ then the proof of Lemma 5.4.3 holds whenever it is invoked in the analysis of the refinement step.

(It can be verified that such a constant exists.) The refinement process terminates with a schedule $\mathcal{S}^{(\zeta)}$ when $I^{(\zeta)}$ is smaller than x. The above conditions give us the inequalities that we need for our analysis.

Lemma 5.4.12 *In the schedule $\mathcal{S}^{(\zeta)}$ all sessions are integral and the relative congestion is at most $c^{(\zeta)} < 1$ for any frame of size $I^{(\zeta)}$ or larger.*

Proof: By Lemma 5.3.2 the sets $B^{(q+1)}$ form a partition of all the sessions. Hence in $\mathcal{S}^{(\zeta)}$ all the sessions are integral. By our analysis, the relative congestion in $\mathcal{S}^{(\zeta)}$ is at most $c^{(\zeta)}$ for all frames of size $I^{(\zeta)}$ or larger. Hence it remains to show that $c^{(\zeta)} < 1$.

By termination condition 1 we have $x \leq I^{(\zeta-1)}$ where $x \geq e^{\beta^2/(1-\sqrt{1-\epsilon/2})^2}$. Let $\Delta = \frac{\beta}{\sqrt{\log x}}$ and observe that $\delta^{(\zeta-1)} \leq \Delta < 1$. Note also that,

$$(\delta^{(q+1)})^2 = \frac{\beta^2}{\log I^{(q+1)}} = \frac{\beta^2}{\log\log^5 I^{(q)}} > \frac{\beta}{\sqrt{\log I^{(q)}}} = \delta^{(q)},$$

by termination condition 2. By the recursive definition of $c^{(\zeta)}$ we have,

$$c^{(\zeta)} = (1 + \delta^{(\zeta-1)})^2(1 + \delta^{(\zeta-2)})^2 \ldots (1 + \delta^{(0)})^2 c^{(0)}$$

$$
\begin{aligned}
&< (1+\Delta)^2(1+\Delta^2)^2(1+\Delta^4)^2(1+\Delta^8)^2 \ldots c^{(0)} \\
&\leq (1-\Delta)^{-2}\left\{(1-\Delta)^2(1+\Delta)^2(1+\Delta^2)^2(1+\Delta^4)^2(1+\Delta^8)^2 \ldots\right\} c^{(0)} \\
&\leq (1-\Delta)^{-2}c^{(0)} \\
&= (1-\beta/\sqrt{\log x})^{-2}c^{(0)} \\
&= \frac{1-\varepsilon/2}{1-\varepsilon/2} \\
&= 1.
\end{aligned}
$$

The first inequality comes from the fact that $\delta^{(q)} < (\delta^{(q+1)})^2$. The third inequality holds since $\Delta < 1$ and hence the product in the braces is less than 1. The penultimate equality follows from the choice of x and the fact that $c^{(0)} = 1 - \varepsilon/2$. □

We now bound the delay that a packet experiences in the schedule $\mathcal{S}^{(\zeta)}$. The analysis of the conversion step shows that when session i becomes integral, the delay experienced by a session-i packet before it starts moving is at most $O(D_i + 1/r_i)$ steps. We also know by the analysis of the refinement step that once a packet starts moving it waits at most one step every $I^{(\zeta-1)}$ steps. However, the delay added during the refinement steps can postpone the time at which a packet starts moving. The following lemma bounds this delay.

Lemma 5.4.13 *During the refinement steps, a session-i packet is delayed by at most $2(D_i + 1/r_i)$ time steps before it starts moving.*

Proof: Suppose that $\mathcal{S}^{(q')}$ is the first schedule in which session i is integral. Let p be a session-i packet. For $q \leq q' - 1$, p is not delayed during the refinement of $\mathcal{S}^{(q)}$. For $q \geq q'$ the time at which p starts moving is only affected by the delays added to one particular block during the refinement step of $\mathcal{S}^{(q)}$. By the definition of the refinement step, p is delayed by at most $I^{(q)} + (I^{(q)})^2$ steps before it starts moving. Therefore the start time of p is delayed by at most $\sum_{q \geq q'} I^{(q)} + (I^{(q)})^2$ steps during refinement. Since $\mathcal{S}^{(q')}$ is the first schedule in which session i is integral we have that $i \in B^{(q')}$. By the definition of $B^{(q')}$, $D_i + 1/r_i (= X_i) \geq (I^{(q')})^2$. By termination condition 3, $I^{(q+1)} \leq \frac{1}{2} I^{(q)}$, hence a session-$i$ packet is delayed during refinement by at most $4(D_i + 1/r_i)$ steps before it starts moving. □

We can now prove that $\mathcal{S}^{(\zeta)}$ has all the properties stated in Theorem 5.4.1.

Proof of Theorem 5.4.1

1. By Lemma 5.4.12, the relative congestion is at most 1 for any frame of size $I^{(\zeta)}$ or larger. The termination conditions imply that $I^{(\zeta)}$ is at most some constant.

2. By the invariant maintained throughout the refinement process, a packet waits for at most one step every $I^{(\zeta-1)}$ steps once it leaves its source. In addition, by Property 1 above, at most $I^{(\zeta)}$ packets cross an edge during any time step. Therefore, the edge queues have size at most $2I^{(\zeta)}$.

3. Let p be a session-i packet and consider the schedule produced by the discretization substep of the conversion step in which session-i is converted. Let t_p be the time at which p obtains an initial token in this schedule. The time at which p starts moving is delayed in both the conversion and the refinement steps. The delay inserted during the conversion step is at most $\ell_i \leq 1+8/\varepsilon r_i$ and the delay due to refinement is at most $4(D_i + 1/r_i)$ by Lemma 5.4.13. By Property 2, the packet p reaches its destination in at most $2D_i$ steps once it starts moving. Therefore, p reaches its destination by time $t_p + 6D_i + 1 + (8/\varepsilon + 4)/r_i = t_p + \mathcal{T}_i$.

   Lemma 5.4.9 implies that t_p is at most $\mathcal{T}_i + \ell_i$ steps after the time at which p is injected. Hence p reaches its destination within $2\mathcal{T}_i + \ell_i = O(1/r_i + D_i)$ steps of its injection.

4. Let p and t_p be as above. By the definition of the discretization substep, if p is injected during $[\mathcal{T} - \mathcal{T}_i, 2\mathcal{T} - \mathcal{T}_i)$ then $t_p \in [\mathcal{T}, 2\mathcal{T} - \mathcal{T}_i)$. Hence the discussion of Property 3 implies that all session-i packets injected during $[\mathcal{T} - \mathcal{T}_i, 2\mathcal{T} - \mathcal{T}_i)$ are scheduled during $[\mathcal{T}, 2\mathcal{T})$.

□

5.5 A Schedule for the Original Network, $\mathcal{N}$

The schedule $\mathcal{S}^{(\zeta)}$ is a schedule for the network $\mathcal{M}$. Of course, we are really concerned with providing a schedule for the network $\mathcal{N}$. We now show how to construct such a schedule which we shall call $\mathcal{S}_{\mathcal{N}}$. Recall that in the construction of $\mathcal{M}$ from $\mathcal{N}$, each edge e in $\mathcal{N}$ was replaced by $2w$ consecutive edges $e_1, \ldots, e_{2w}$. The length of the ith session in $\mathcal{M}$ was therefore $D_i = 2wd_i$. However, we did not define w. We now do this by setting $w = x$, where x is the constant defined in the termination conditions. (See Section 5.4.4.) In the schedule $\mathcal{S}_{\mathcal{N}}$, which we define below, all session-i packets will reach their destinations within time $O(1/r_i + d_i)$ and at most one packet will cross an edge during any time step. Hence the schedule $\mathcal{S}_{\mathcal{N}}$ will be a legal schedule and it will have the desired delay bounds.

We first partition the time interval $[\mathcal{T}, 2\mathcal{T})$ into consecutive intervals of length w. (Recall that we set $\mathcal{T}$ to be an integer multiple of w). For each w-interval and each edge f in $\mathcal{M}$, at most w packets can cross f during the w-interval in the schedule $\mathcal{S}^{(\zeta)}$. (This is because $w \geq I^{(\zeta)}$.) We now *smooth out* the schedule $\mathcal{S}^{(\zeta)}$ to obtain a schedule $\bar{\mathcal{S}}^{(\zeta)}$. Let $p_1, p_2, \ldots$ be an arbitrary ordering of the packets that cross an edge f during a w-interval in the schedule $\mathcal{S}^{(\zeta)}$. In $\bar{\mathcal{S}}^{(\zeta)}$ we schedule p_j to cross f during the jth step of the w-interval. (Recall that we are considering at most w packets.) The schedule $\bar{\mathcal{S}}^{(\zeta)}$ has the property that at most one packet may cross an edge during any time step. Unfortunately however, since the times at which packets cross edges have been shifted, a packet may not be scheduled to cross the edges on its route in order. For example, a packet may be scheduled to cross g before f, whereas g follows f on its route in $\mathcal{M}$. A packet may also be scheduled to leave its source before its injection time. We can use $\bar{\mathcal{S}}^{(\zeta)}$ however to define a legal schedule for the network $\mathcal{N}$.

Definition 5.5.1 *In the schedule $\mathcal{S}_{\mathcal{N}}$, a packet p crosses edge e in $\mathcal{N}$ at time t if and only if it crosses edge e_{2w} in $\mathcal{M}$ at time t in the schedule $\bar{\mathcal{S}}^{(\zeta)}$.*

Lemma 5.5.2 *In $\mathcal{S}_{\mathcal{N}}$, each packet is scheduled to leave its source after its injection and is scheduled to cross the edges on its route in order.*

Proof: We first show that each packet crosses the edges on its route in order. Consider a packet p. Let e and $\hat{e}$ be two edges on p's route in $\mathcal{N}$ and suppose that $\hat{e}$ follows e. Let t and $\hat{t}$ be the times that p crosses e and $\hat{e}$ in the schedule $\mathcal{S}_{\mathcal{N}}$. We shall show that $t < \hat{t}$. Let e_{2w} and $\hat{e}_{2w}$ be the edges in $\mathcal{M}$ that correspond to e and $\hat{e}$. Let τ and $\hat{\tau}$ be the times that p crosses e_{2w} and $\hat{e}_{2w}$ in the schedule $\mathcal{S}^{(\zeta)}$ (i.e. *before the smoothing*). Since in $\mathcal{S}^{(\zeta)}$ the packet p crosses the edges in $\mathcal{M}$ in order we have,

$$\tau + 2w \leq \hat{\tau}.$$

In the schedule $\mathcal{S}_{\mathcal{N}}$, p crosses e at time t which is shifted by at most $w-1$ steps from τ. (Note that in $\bar{\mathcal{S}}^{(\zeta)}$, p crosses edge e_{2w} at time t.) Similarly, $\hat{t}$ is shifted by at most $w-1$ steps from $\hat{\tau}$. Hence,

$$\tau - (w-1) \leq t \leq \tau + (w-1),$$

$$\hat{\tau} - (w-1) \leq \hat{t} \leq \hat{\tau} + (w-1).$$

The above inequalities imply that $t < \hat{t}$. Therefore, p crosses the edges on its route in order.

The proof that p leaves its source after its injection time is similar. Suppose that p is injected at time s. Let e be the first edge on the route of p in the network $\mathcal{N}$, and let t be the time that p crosses e in $\mathcal{S}_{\mathcal{N}}$. Let e_{2w} be the edge in $\mathcal{M}$ that corresponds to e and let τ be the time that p crosses e_{2w} in $\mathcal{S}^{(\zeta)}$. Since in $\mathcal{S}^{(\zeta)}$ packet p crosses the edges in order and leaves its source after its injection we have,

$$s + 2w \leq \tau.$$

In the schedule $\mathcal{S}_{\mathcal{N}}$, p crosses e at time t, which is shifted by at most $w-1$ steps from τ. Hence,

$$\tau - (w-1) \leq t \leq \tau + (w+1).$$

Therefore, $s < t$ which means that in $\mathcal{S}_{\mathcal{N}}$, packet p is scheduled to leave its source

after its time of injection. □

Theorem 5.5.3 *Schedule $\mathcal{S}_{\mathcal{N}}$ satisfies the following properties.*

1. *At most one packet crosses an edge in $\mathcal{N}$ during any time step.*
2. *After leaving its source, each packet waits at most a constant number of steps to cross each edge, which implies that all the edge queues in $\mathcal{N}$ have constant size.*
3. *Each session-i packet reaches its destination within $O(1/r_i + d_i)$ steps of its injection time.*
4. *All session-i packets that are injected during $[\mathcal{T} - \mathcal{T}_i, 2\mathcal{T} - \mathcal{T}_i)$ are scheduled during $[\mathcal{T}, 2\mathcal{T})$.*

Proof: The smoothing process guarantees Property 1. Properties 2 and 3 come from Properties 2 and 3 of $\mathcal{S}^{(\zeta)}$ given in Theorem 5.4.1, the construction of $\mathcal{M}$ from $\mathcal{N}$ and the fact that packets are scheduled to reach their destination in $\mathcal{S}_{\mathcal{N}}$ at most w steps later than in $\mathcal{S}^{(\zeta)}$.

For Property 4, recall that the interval $[\mathcal{T}, 2\mathcal{T})$ is partitioned into intervals of size w and the schedule $\mathcal{S}^{(\zeta)}$ is smoothed out within each w-interval. Therefore, if a packet is scheduled to cross an edge e_{2w} during $[\mathcal{T}, 2\mathcal{T})$ in the schedule $\mathcal{S}^{(\zeta)}$, it must also be scheduled to cross e during $[\mathcal{T}, 2\mathcal{T})$ in the schedule $\mathcal{S}_{\mathcal{N}}$. Hence Property 4 follows from Property 4 of $\mathcal{S}^{(\zeta)}$ given in Theorem 5.4.1. □

Property 4 of the above theorem implies that the intervals $[0, \mathcal{T}), [\mathcal{T}, 2\mathcal{T}), [2\mathcal{T}, 3\mathcal{T})$ etc. can be scheduled independently. To summarize we have,

Theorem 5.5.4 *Consider an arbitrary network in which sessions are defined. Each session i is associated with an injection rate r_i and a path length d_i. Packets are injected into the network along these sessions subject to the injection rates. If the total rate on each edge is at most $1 - \varepsilon$ for some constant $\varepsilon \in (0, 1)$, then there exists a schedule in which each session-i packet reaches its destination within $O(1/r_i + d_i)$ steps of its injection and at most one packet crosses an edge at each time step. The edge queues that result from this schedule have constant size.*

Part II

Load Balancing

Chapter 6

Load Balancing - Introduction

6.1 The Problem

In this part of the thesis we consider the on-line load balancing problem. A formal definition of our problem is as follows. A set of jobs arrive in and depart from a system of m machines on-line, i.e. the arrival and departure times become known only when these events occur. Each job j has a *weight*, $w_j > 0$, which measures the level of service needed for the job, and a *reassignment cost*, $r_j > 0$, which measures the cost of assigning (or reassigning) the job to a machine. Each machine i has an associated capacity, cap_i, which measures the ability of the machine to carry out work. If $\text{cap}_i = 1$ for all i then we say that the machines are *identical*, otherwise we say that the machines are *related*.

When job j arrives in the system, the on-line scheduler is informed of its weight and reassignment cost (but not of its departure time or, equivalently, its duration) and has to assign job j to some machine i at a cost of r_j. At any time, the scheduler is allowed to reassign some of the jobs currently in the system to other machines. The corresponding reassignment cost r_j is paid for each reassigned job. A job is said to be *active* at time t if it has arrived but not yet departed. The load on machine i at time t is defined to be,

$$\lambda_i(t) = \frac{\sum_{j \in U_i(t)} w_j}{\text{cap}_i},$$

where $U_i(t)$ is the set of active jobs on machine i at time t. (For notational convenience, we shall later drop the time dependence.) If the machines are identical then $\lambda_i(t)$ is simply the sum of the weights of the active jobs on machine i. The maximum load $\lambda(t)$ at time t is the maximum of $\lambda_i(t)$ over all i. Let S be the sum of the reassignment costs of all jobs that have arrived in the system (regardless of whether or not they have departed). We say that an on-line algorithm is *α-competitive against current load and has a reassignment factor of r* if, for any time t, the maximum load $\lambda(t)$ is at most α times the lowest possible load for the active jobs at time t and the total reassignment cost (including the initial assignments) is at most rS. A competitive ratio of 1 means that the load is distributed optimally and a reassignment factor of 1 means that no reassignment is performed. We shall consider the cases of identical and related machines separately.

Identical Machines

For the case in which the machines are identical and the reassignment costs are either 1 or proportional to the weights ($r_j = aw_j$ for some constant a), Westbrook [69] presents an on-line algorithm that has a constant competitive ratio against current load and a constant reassignment factor. The competitive ratio he proves is 6 and his reassignment factor is 2 for unit reassignments and 3 for proportional reassignments. Westbrook's algorithm is based on partitioning the jobs into *levels* according to their weights and then load balancing each level separately.

The results for identical machines presented in this thesis are the following.

- For unit reassignment costs, we give a different analysis showing that the competitive ratio of Westbrook's algorithm is in fact 5. We then present an on-line algorithm, ZIGZAG, that treats even and odd levels differently and achieves a competitive ratio of 11/3 against current load. This analysis is shown to be tight. By defining the levels differently we reduce the competitive ratio still further to $1+3\sqrt{3}/2 < 3.5981$. ZIGZAG and its variant have a reassignment factor of 2, the same as Westbrook's algorithm. We also present a different algorithm,

the MULTI-SNAKES algorithm, which for any $\varepsilon > 0$ achieves a competitive ratio of $3 + \varepsilon$ at the expense of a greater reassignment factor, $f(\varepsilon)$.

- When reassignment costs are proportional to the weights, ZIGZAG has a competitive ratio of 3.5981 against current load and a reassignment factor of 3, improving upon Westbrook's competitive ratio of 6. We also present an algorithm, SNAKE, which is $(2 + \varepsilon)$-competitive against current load and has a reassignment factor of $g(\varepsilon)$.

- For arbitrary reassignment costs, our ZIGZAG-SNAKES algorithm is 3.5981-competitive against current load and has a reassignment factor of 6.8285. This is the first algorithm that achieves a constant competitive ratio against current load while simultaneously having a constant reassignment factor. The ZIGZAG-SNAKES algorithm is obtained by combining the ZIGZAG algorithm with the SNAKE algorithm. Within each level of the ZIGZAG algorithm, the jobs are balanced with respect to both their weights and reassignment costs by using the SNAKE algorithm.

Related Machines

Our results for related machines are as follows. Both algorithms can be used for arbitrary reassignment costs.

- We first present an algorithm, GREEDY ZIGZIG that is similar in spirit to ZIGZAG-SNAKES. The competitive ratio (against current load) is logarithmic in the ratio of the largest machine capacity to the smallest machine capacity and the reassignment factor is 6.8285.

- We also describe an algorithm, BALANCE-RELATED, that is 32-competitive against current load and has a reassignment factor of 79.4. This is the first algorithm that has a constant competitive ratio and a constant reassignment factor. It is an adaptation of the algorithm of Westbrook [69] that considered the special case in which r_j is proportional to w_j. This condition is helpful

because if a job of large weight leaves then it has a large reassignment cost and so we can "charge" the reassignment which may be necessary to the job that is leaving. (Note that it is the departure of jobs of large weight that can cause the load to become unbalanced.) If we have arbitrary reassignment costs and a job of large weight leaves then it might have a small reassignment cost and so it is less clear how to "pay for" the reassignment that might have to be carried out.

We deal with arbitrary reassignment costs by dividing the jobs into classes according to their ratio r_j/w_j. When we wish to assign a job, our choice of machine is based solely on the location of jobs in the classes for which this ratio is at least as big as that of the job being assigned. If the load on a machine becomes too great we can then reassign jobs whose ratio r_j/w_j is small. Whenever any reassignment is carried out we are able to "charge" the costs incurred to jobs whose ratio r_j/w_j is large. The detailed analysis of the reassignment costs is carried out using the method of potential functions.

Our results for identical and related machines are contained in Chapters 7 and 8 respectively.

6.2 Previous Work

In order to understand many of the previous results for load balancing we must make some more definitions. Let $\lambda^*(t)$ be the optimal assignment of the jobs that are active at time t. We say that an on-line algorithm is α-competitive against *peak* load if, for any time t, the maximum load $\lambda(t)$ in the assignment produced by the algorithm satisfies $\lambda(t) \leq \alpha \max_{t' \leq t} \lambda^*(t')$. As discussed in Chapter 1, competitiveness against peak load is a much weaker requirement than competitiveness against current load since an algorithm does not need to take advantage of job departures. (Note that if there are no job departures then current load is the same as peak load.) We now describe the previous work on problems related to load balancing. If no reassignment information is given then the algorithm being discussed does not make use of reassignment.

6.2.1 Identical Machines

The first result that can be applied to on-line load balancing is due to Graham [28]. He showed that for the identical machines problem where jobs never depart, there is a simple greedy algorithm that has a competitive ratio of $2 - 1/m$. Azar, Broder and Karlin [5] noted that Graham's algorithm is still $(2 - 1/m)$-competitive against *peak* load even if jobs departures are allowed. Bartal, Fiat, Karloff and Vohra [9] and Karger, Phillips and Torng [36] gave algorithms for identical machines that are $(2-\varepsilon)$-competitive (ε a small constant) for the case where jobs never depart. As indicated above, Westbrook [69] gave an algorithm that is 6-competitive against *current* load and has a reassignment factor of 2 for unit reassignment costs and 3 for reassignment costs proportional to job weights.

6.2.2 Related Machines

For the related machines problem in which jobs never depart, Aspnes, Azar, Fiat, Plotkin and Waarts gave an 8 competitive algorithm. For the case in which jobs can depart, Azar, Kalyanasundaram, Plotkin, Pruhs and Waarts [6] presented an algorithm that is 20-competitive against *peak* load and showed that no algorithm can be better than $(3 - o(1))$-competitive if job reassignments are not allowed. Westbrook's algorithm [69] is $(8 + \varepsilon)$-competitive against current load and has an $O(1)$ reassignment factor for the case in which reassignment costs are proportional to job weights.

6.2.3 Restricted Assignment

In the restricted assignment problem all machines have capacity 1 but each job can only be serviced by an associated subset of the machines. (The identical machines problem is clearly the special case in which for all jobs this subset is the complete set of machines.)

Azar, Naor and Rom [7] studied Graham's greedy algorithm and showed that it is $O(\log m)$ competitive for the restricted assignment problem in which jobs never

depart. Moreover they showed that this is optimal up to constant factors when reassignment is not allowed, even for randomized algorithms. Azar, Broder and Karlin [5] then showed that if jobs are allowed to depart the problem becomes significantly harder. The greedy algorithm is $O(m^{2/3})$-competitive against *peak* load and no deterministic or randomized algorithm is better than $O(\sqrt{m})$-competitive if reassignment is not allowed. Azar, Kalyanasundaram, Plotkin, Pruhs and Waarts [6] closed the gap by obtaining an algorithm that is $O(\sqrt{m})$-competitive against *peak load* and does not reassign jobs.

Phillips and Westbrook [55] gave an algorithm that is $O((\log m)/\rho)$ competitive against *current* load and has a reassignment factor of $1 + \rho$ for reassignment costs proportional to job weights. Here ρ is a user-specified parameter that satisfies $0 < \rho < 1$. This was the first load balancing algorithm to be competitive against current load.

Awerbuch, Azar, Plotkin and Waarts [4] gave an algorithm for the case in which all jobs have unit weight and the peak load is $\Omega(\log m)$. Their algorithm is 4-competitive against *peak* load and has a reassignment factor of $O(\log m)$ for unit reassignment costs. Westbrook also considered the unit weight problem. His algorithm is $O(1)$-competitive against *current* load and has a reassignment factor of $O(\log m)$.

6.2.4 Unrelated Machines

In the unrelated machines problem each job has a *load vector* with m components. If the job is assigned to machine i then the load on machine i is increased by an amount equal to the ith component of the load vector. The restricted assignment problem can be viewed as a special case of the unrelated machines problem (if we allow some of the components of the load vector to be ∞). Hence the $\Omega(\sqrt{m})$ lower bound of Azar, Broder and Karlin [5] still applies for the case in which jobs depart and reassignment is not allowed. However, this lower bound assumes that the *duration* of a job (i.e. the length of time it spends in the system) is not known when the job arrives. If the duration *is* known when the job arrives then Azar, Kalyanasundaram, Plotkin, Pruhs and Waarts [6] gave an algorithm for the unrelated machines problem that is

$O(\log mT)$-competitive against *peak* load, where T is the ratio of the maximum to the minimum job duration.

Aspnes, Azar, Fiat, Plotkin and Waarts [2] considered the unrelated machines problem in which jobs do not depart and gave an $O(\log m)$-competitive algorithm. By the lower bound for restricted assignment this is asymptotically optimal if reassignment is not allowed. Aspnes et al. note that Graham's greedy algorithm is $\Theta(m)$-competitive for this problem.

Awerbuch, Azar, Plotkin and Waarts [4] considered the case in which jobs depart and reassignment is allowed. Their algorithm is $O(\log m)$-competitive against *peak* load and reassigns each job at most $O(\log m)$ times. Note that this immediately implies an $O(\log m)$ reassignment factor for arbitrary reassignment costs.

6.2.5 Virtual Circuit Routing – Congestion Minimization

In the virtual circuit routing problem we are given a network in which each edge has a capacity. *Calls* that require a certain amount of bandwidth between two nodes in the network arrive in the system. In the congestion minimization version of the problem, the goal is to route all the calls while minimizing the maximum edge *congestion*. The congestion on an edge is defined to be the percentage of capacity used. The related machines load balancing problem is the special case of this problem in which the network has two nodes and m edge-disjoint paths between these nodes.

Aspnes, Azar, Fiat, Plotkin and Waarts [2] gave an $O(\log n)$-competitive algorithm for virtual circuit routing for the case where calls never depart. Here n is the number of nodes in the graph. In [4], Awerbuch, Azar, Plotkin and Waarts considered the case in which calls can depart and gave an algorithm that is $O(\log n)$-competitive against *peak* load and reroutes each call $O(\log n)$ times. Westbrook [69] extended this to an algorithm that is $O(\log n)$-competitive against *current* load and has a reassignment factor of $O(\log n \log C/\mathrm{cap}_{\mathrm{min}})$, where C is the sum of the edge capacities and $\mathrm{cap}_{\mathrm{min}}$ is the minimum edge capacity. All of these algorithms are based on assigning a length to each edge that is an exponential function of its congestion and then routing along shortest paths with respect to these lengths.

6.2.6 Virtual Circuit Routing – Throughput Maximization

In the throughput maximization version of the virtual circuit routing problem, we assume that the total bandwidth assigned to an edge cannot exceed the capacity of the edge. This means that some calls may need to be rejected. The goal is to accept as much *profit* as possible, the profit of a call being a function of its bandwidth and duration. An algorithm is α-competitive if the total profit that it accepts is at least a $1/\alpha$ fraction of the total profit accepted by the optimal offline algorithm.

Awerbuch, Azar, Plotkin and Waarts [4] noted that an on-line algorithm cannot be competitive in this setting unless it knows the duration of calls in advance. Awerbuch, Azar and Plotkin [3] made this assumption and also assumed that the maximum bandwidth of a call is at most an $O(1/\log nT)$ fraction of the minimum edge capacity, where T is the ratio of the longest call duration to the shortest call duration. They then presented an algorithm that is $O(\log nT)$-competitive. Kamath, Palmon and Plotkin [35] later adapted this algorithm for the special case in which the calls are generated by Poisson processes and have durations that are drawn from an exponential distribution. In [25], Gawlick, Kamath, Plotkin and Ramakrishnan simulated the performance of the Awerbuch, Azar, Plotkin algorithm and modified its parameters so as to give good practical performance. Zhang, Andrews, Aiello, Bhatt and Krishnan [75] compared the resulting algorithm with algorithms developed by the queueing theory community.

Chapter 7

Identical Machines

In this chapter we consider the identical machines problem. Our competitive analyses against current load will be performed by comparing the load of our on-line algorithm at time t to a lower bound on the lowest achievable load for the active jobs. Let J be the set of active jobs and let w_{max} be the maximum weight of a job in J. Then clearly w_{max} is a lower bound on the current load. Since all jobs are distributed to m machines, some machine has to have a load which is at least $\sum_{j \in J} \frac{w_j}{m}$. Hence $LB := \max\{w_{max}, \sum_{j \in J} \frac{w_j}{m}\}$ is a lower bound on the current load.

7.1 Unit Reassignment Costs

In this section we assume that every job has unit reassignment cost (r_j=1). We start by examining the trivial case in which every job also has unit weight ($w_j = 1$). Consider any algorithm that maintains the invariant that the number of jobs on any two machines differs by at most 1, i.e. the number of jobs on every machine is either p or $p+1$ for some p, with possibly no machine with $p+1$ jobs. It is clear that any such algorithm distributes the load optimally. Moreover, the invariant is easy to maintain. If a job arrives, simply assign it to any machine with p jobs. If a job leaves from a machine with $p+1$ jobs, the invariant is already maintained. On the other hand, if a job leaves from a machine with p jobs, say machine i, the algorithm moves any job from a machine with $p+1$ jobs to machine i. We pay one unit of cost for each arrival

of a job and at most one unit for each departure. Hence the reassignment factor of this algorithm is at most 2.

For the case in which the jobs have arbitrary weights, Westbrook [69] gave the following generalization of this algorithm and showed that it has a competitive ratio of 6. The jobs are divided into *levels* according to their weights. Level ℓ consists of jobs whose weight w_j satisfies $2^\ell \leq w_j < 2^{\ell+1}$. Let J_ℓ be the set of active jobs in level ℓ. We treat each job in J_ℓ as if it had weight 2^ℓ. For level ℓ, let $n^{(\ell)} = |J_\ell|$ be the number of jobs, $w^{(\ell)}_{max}$ be the maximum weight of a job, $w^{(\ell)}_{avg} = \sum_{j \in J_l} w_j / n^{(\ell)}$ be the average weight of the jobs, and $\lambda^{(\ell)}_i$ be the load on machine i due to these jobs. Consider one particular level ℓ. The algorithm guarantees that every machine has either p or $p+1$ jobs in this level. Since the weights of all these jobs differ by a factor less than two, the load $\lambda^{(\ell)}_i$ on any machine of p jobs is at most twice the average load $\sum_{j \in J_\ell} \frac{w_j}{m}$, while the load of the potential $(p+1)$st job can be simply bounded by $w^{(\ell)}_{max}$. This implies that $\lambda^{(\ell)}_i \leq 2 \sum_{j \in J_\ell} \frac{w_j}{m} + w^{(\ell)}_{max}$ for any machine i. Let $L = \max\{\ell : J_\ell \neq \emptyset\}$. Summing over all levels, we derive that the load on any machine is at most,

$$\begin{aligned} & 2 \sum_{j \in J} \frac{w_j}{m} + w^{(L)}_{max} \left(1 + 1 + \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \ldots\right) \\ = \; & 2 \sum_{j \in J} \frac{w_j}{m} + 3 w^{(L)}_{max} \\ \leq \; & 5LB, \end{aligned}$$

showing that Westbrook's algorithm is in fact 5-competitive. The term in parentheses comes from the fact that for all $\ell' < \ell$, $w^{(\ell')}_{max} < w^{(\ell)}_{max} / 2^{\ell - \ell' - 1}$.

We now describe a more sophisticated variant of this algorithm with an improved performance guarantee. The idea is to take advantage of the flexibility one has in adding and reassigning jobs. The algorithm will be slightly different for even levels and odd levels. Assume that the machines are numbered from 1 to m. For even ℓ, we maintain that machine i has $p+1$ jobs for $i < z$ and p jobs for $z \leq i \leq m$ for some pointer $z \in \{1, \ldots, m\}$ (depending on ℓ). The initial value of the pointer is

1. When a job arrives we assign it to machine z and we increase z by 1. When a job leaves from machine i we reassign a job from machine $z-1$ to machine i and decrease z by 1. (We assume that $m+1=1$ and $1-1=m$.) If ℓ is odd, the value of the pointer is initialized to m, decreased when a job arrives and increased when a job leaves. This means that for odd levels the machines with $p+1$ jobs are those for which $i > z$. We refer to this algorithm as the ZIGZAG algorithm. See Figure 7-1 for an illustration. The ZIGZAG algorithm has the following important feature: If one combines two consecutive levels, the number of jobs on any two machines still differs by at most one.

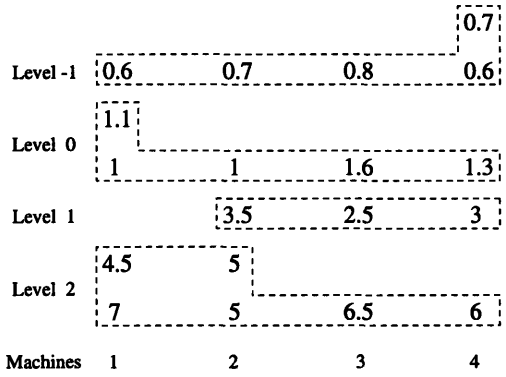

Figure 7-1: A typical state of the ZIGZAG algorithm for identical machines.

In Theorem 7.1.3 we show that the competitive ratio for the ZIGZAG algorithm is 11/3. To prove this result we need the following two lemmas. First, consider some arbitrary level ℓ (for notational simplicity, we drop the superscript). Assume that we have n jobs in level ℓ and every machine has p or $p+1$ jobs (from this level). Let $n = pm + k$, where $0 \leq k < m$.

Lemma 7.1.1 *If machine i has p jobs, then λ_i is at most $2\sum_{j\in J_\ell} \frac{w_j}{m} - w_{max}\frac{k}{m}$. If machine i has $p+1$ jobs, then λ_i is at most $2\sum_{j\in J_\ell} \frac{w_j}{m} + w_{max}(1-\frac{k}{m})$.*

Proof: The weights of all the jobs in the level differ by a factor less than 2. This means that the average job weight w_{avg} satisfies $w_{max} \leq 2w_{avg}$. If machine i has p jobs then,

$$\lambda_i \leq w_{max}p$$

$$
\begin{aligned}
&= w_{max}\left(p+\frac{k}{m}\right) - w_{max}\frac{k}{m} \\
&\leq 2w_{avg}\left(p+\frac{k}{m}\right) - w_{max}\frac{k}{m} \\
&= 2\left(\sum_{j\in J_\ell}\frac{w_j}{n}\right)\left(p+\frac{k}{m}\right) - w_{max}\frac{k}{m} \\
&= 2\left(\sum_{j\in J_\ell}\frac{w_j}{n}\right)\left(\frac{pm+k}{m}\right) - w_{max}\frac{k}{m} \\
&= 2\sum_{j\in J_\ell}\frac{w_j}{m} - w_{max}\frac{k}{m}.
\end{aligned}
$$

On the other hand, if machine i has $p+1$ jobs then $\lambda_i \leq w_{max}(p+1)$, and thus λ_i is upper bounded by the above expression increased by w_{max}. □

We now consider what happens in two consecutive levels. This is where the ZIGZAG algorithm gains.

Lemma 7.1.2 *The load on machine i due to jobs in J_ℓ and $J_{\ell+1}$ is at most $2\sum_{j\in J_\ell\cup J_{\ell+1}}\frac{w_j}{m}+ w_{max}^{(\ell+1)}$.*

Proof: We consider two cases. First, assume that machine i has simultaneously $p^{(\ell)}+1$ jobs of level ℓ and $p^{(\ell+1)}+1$ jobs of level $\ell+1$. Then it must be the case that $k^{(\ell+1)}+k^{(\ell)} > m$ since the pointer for level $\ell+1$ travels in the opposite direction to the pointer for level ℓ. Hence by Lemma 7.1.1 and the fact that $w_{max}^{(\ell+1)} \geq w_{max}^{(\ell)}$ we obtain:

$$
\begin{aligned}
\lambda_i^{(\ell+1)} + \lambda_i^{(\ell)} &\leq 2\sum_{j\in J_\ell\cup J_{\ell+1}}\frac{w_j}{m} + w_{max}^{(\ell+1)}\left(1-\frac{k^{(\ell+1)}}{m}\right) + w_{max}^{(\ell)}\left(1-\frac{k^{(\ell)}}{m}\right) \\
&\leq 2\sum_{j\in J_\ell\cup J_{\ell+1}}\frac{w_j}{m} + w_{max}^{(\ell+1)} + w_{max}^{(\ell)} - \frac{w_{max}^{(\ell)}\left(k^{(\ell)}+k^{(\ell+1)}\right)}{m} \\
&< 2\sum_{j\in J_\ell\cup J_{\ell+1}}\frac{w_j}{m} + w_{max}^{(\ell+1)} + w_{max}^{(\ell)} - w_{max}^{(\ell)} \\
&= 2\sum_{j\in J_\ell\cup J_{\ell+1}}\frac{w_j}{m} + w_{max}^{(\ell+1)}.
\end{aligned}
$$

In the second case, machine i has either $p^{(\ell)}$ jobs of level ℓ or $p^{(\ell+1)}$ jobs of level

$\ell + 1$. Lemma 7.1.1 implies that

$$\lambda_i^{(\ell)} + \lambda_i^{(\ell+1)} \leq 2 \sum_{j \in J_\ell \cup J_{\ell+1}} \frac{w_j}{m} + w_{max}^{(\ell+1)}.$$

□

Theorem 7.1.3 *The* ZIGZAG *algorithm is* (11/3)*-competitive against current load. For unit reassignment costs, the reassignment factor is* 2.

Proof: The bound on the reassignment cost holds because we pay at most one unit of cost for each job arrival or departure. Recall that $L = \max\{\ell : J_\ell \neq \emptyset\}$. Using Lemma 7.1.2, aggregating two levels at a time (starting from level L) and summing over these pairs of levels, we derive that the total load on any machine is at most,

$$\begin{aligned}
\lambda_i &\leq \sum_{\ell} 2 \sum_{j \in J_\ell} \frac{w_j}{m} + \sum_{q=0}^{\infty} w_{max}^{(L-2q)} \\
&\leq 2 \sum_{j \in J} \frac{w_j}{m} + w_{max}^{(L)} \left(1 + \frac{1}{2} + \frac{1}{2^3} + \frac{1}{2^5} + \ldots\right) \\
&= 2 \sum_{j \in J} \frac{w_j}{m} + \frac{5}{3} w_{max}^{(L)} \\
&\leq \frac{11}{3} LB.
\end{aligned}$$

The second inequality holds because $w_{max}^{(L)} \geq 2^L$ and $w_{max}^{(L-2q)} \leq 2^{L-2q+1}$. □

We give the following example to show that the ratio of 11/3 is tight for the ZIGZAG algorithm. Fix a parameter s (s even) and suppose that we have $m = 3^s$ machines. Consider the following sequence of arrivals:

$$\{2^s; (m-1)\times 2^{s-1}, (2^s)^-; (2^{s-1})^-, (m-1)\times 2^{s-2}, (2^{s-1})^-; \ldots (m-1)\times 2, 4^-; 2^-, (m-1)\times 1, 2^-\}.$$

By $(m-1) \times 2^{s-1}$ we mean $m-1$ separate jobs of weight 2^{s-1} and by $(2^s)^-$ we mean a job of weight slightly less than 2^s. The case $s = 4$ is illustrated in Figure 7-2. Using the ZIGZAG algorithm the load on machine 1 is,

$$2^s \left(2\left(1 + \frac{1}{2} + \frac{1}{2^3} + \frac{1}{2^5} + \ldots + \frac{1}{2^{s-1}}\right) + \left(\frac{1}{2^2} + \frac{1}{2^4} + \frac{1}{2^6} + \ldots + \frac{1}{2^{s-2}}\right)\right).$$

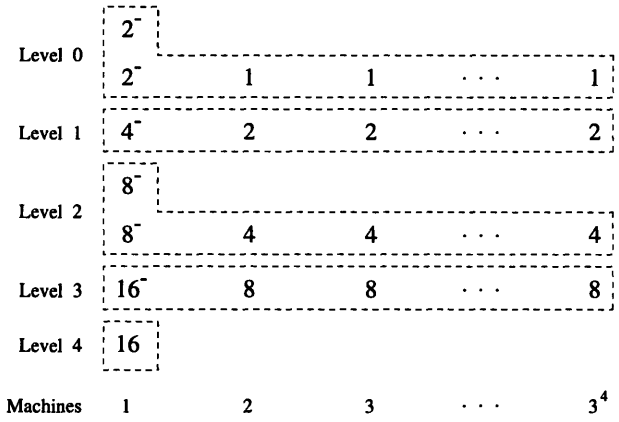

Figure 7-2: An example to demonstrate the tightness of the competitive ratio 11/3.

This expression approaches $\frac{11}{3}2^s$ as s tends to infinity. The jobs can however be placed on the machines so that the current load is 2^s.

We can actually get a competitive ratio better than $11/3 < 3.6667$ by defining the levels differently. Suppose level ℓ consists of jobs whose weight w_j satisfies $\alpha^\ell \leq w_j < \alpha^{\ell+1}$. By using an analysis identical to that used in proving Theorem 7.1.3, we obtain a competitive ratio of,

$$\alpha + 1 + \frac{1}{\alpha} + \frac{1}{\alpha^3} + \frac{1}{\alpha^5} + \ldots = 1 + \alpha + \frac{\alpha}{\alpha^2 - 1}.$$

We minimize this expression by setting $\alpha = \sqrt{3}$ to give a ratio of $1 + 3\sqrt{3}/2 < 3.5981$.

Theorem 7.1.4 *For $r_j = 1$, the* ZIGZAG *algorithm with $\alpha = \sqrt{3}$ is 3.5981-competitive against current load and has a reassignment factor of 2.*

7.2 Proportional Reassignment Costs

In the case of reassignment costs r_j of the form aw_j for some constant a, Westbrook's algorithm [69] as well as the ZIGZAG algorithm still apply. Since reassignments occur only within a level, the weight of a reassigned job is at most twice the weight of a departing job. As a result, both algorithms incur a reassignment cost upper bounded by $3S$. Theorem 7.1.4 thus implies:

Theorem 7.2.1 *When $r_j = aw_j$, the* ZIGZAG *algorithm with $\alpha = \sqrt{3}$ is 3.5981-competitive against current load and has a reassignment factor of 3.*

In this section we present an algorithm with an improved competitive ratio for the case of proportional reassignment costs, at the expense of a somewhat weaker bound on the reassignment cost. As for the ZIGZAG algorithm, level ℓ consists of jobs whose weight w_j satisfies $\alpha^\ell \leq w_j < \alpha^{\ell+1}$ for a fixed parameter $\alpha > 1$, and each level has a pointer, $z^{(\ell)}$. The difference is in the way assignments and reassignments are performed. The algorithm, which we will call the SNAKE algorithm, maintains the following property: for any ℓ, there exists a threshold y and a parameter q such that all machines whose index is less than y have a total of $q+1$ jobs from levels $\hat{\ell} > \ell$ while all machines whose index is at least y have a total of q jobs from levels greater than ℓ. This is maintained in the following way. (See Figures 7-3 and 7-4.) When a level ℓ job arrives, we assign it to machine $i = z^{(\ell)}$ and increase $z^{(\ell)}$ by 1. If level ℓ was empty before the arrival, the pointer $z^{(\ell)}$ would first be initialized to the value of the pointer $z^{(\hat{\ell})}$ of the next non-empty level $\hat{\ell} > \ell$, or to 1 if ℓ is greater than any existing level. Once the job has been assigned to machine i, we find a job of level $\max\{\ell' < \ell : J_{\ell'} \neq \emptyset\}$ on machine i and reassign it by treating it as a new arrival (it will thus be placed on machine $z^{(\ell')}$). Consequently, this will cause a job of level ℓ'' on machine $z^{(\ell')}$ to be reassigned, and so on. When a level ℓ job leaves machine i, we reassign a job of level ℓ on machine $z^{(\ell)} - 1$ to machine i and decrease $z^{(\ell)}$ by 1. We then reassign jobs in levels $\ell' < \ell$ in a manner that is the reverse of that carried out for job arrivals. Thus the arrival or the departure of a level ℓ job will cause all pointers $z^{(\ell')}$ of levels $\ell' \leq \ell$ to be incremented or decremented respectively.

Consider the example described in Figure 7-3. If a level 1 job arrives, it is added to machine 2; this causes a level 0 job to be reassigned from machine 2 to machine 3, and a level -1 job to be reassigned from machine 3 to machine 4. The result is shown in Figure 7-4. If a level 0 job on machine 1 now leaves, a level 0 job is moved from machine 3 to machine 1, and a level -1 job is moved from machine 4 to machine 3.

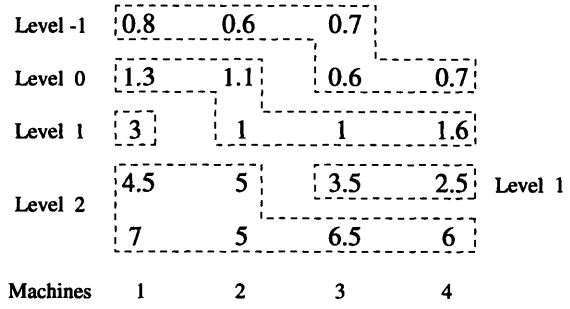

Figure 7-3: The SNAKE algorithm. The pointers are $z^{(-1)} = 4$, $z^{(0)} = 3$, $z^{(1)} = 2$ and $z^{(2)} = 3$.

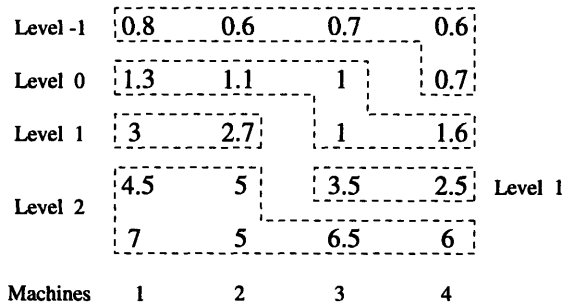

Figure 7-4: The configuration of the SNAKE algorithm after a level 1 job arrival. The pointers are updated to $z^{(-1)} = 1$, $z^{(0)} = 4$, $z^{(1)} = 3$ and $z^{(2)} = 3$.

Lemma 7.2.2 *When $r_j = aw_j$, the reassignment factor of the* SNAKE *algorithm is* $\frac{\alpha^2+2\alpha-1}{\alpha-1} = 3 + \alpha + \frac{2}{\alpha-1}$.

Proof: When a job of level ℓ arrives, say job j, a job from each level smaller than ℓ will be reassigned. Thus, the reassignment cost incurred when job j arrives is at most,

$$\begin{aligned} aw_j + a\alpha^{\ell} + a\alpha^{\ell-1} + a\alpha^{\ell-2} + \ldots &= aw_j + a\frac{\alpha^{\ell}}{1-1/\alpha} \\ &\leq aw_j + a\frac{w_j}{1-1/\alpha} \\ &= \frac{2\alpha-1}{\alpha-1} r_j. \end{aligned}$$

Similarly, the reassignment cost incurred when job j departs is at most,

$$a\alpha^{\ell+1} + a\alpha^{\ell} + a\alpha^{\ell-1} + \ldots = a\frac{\alpha^{\ell+1}}{1-1/\alpha} \leq \frac{\alpha^2}{\alpha-1} r_j.$$

Hence the reassignment factor is $\frac{\alpha^2+2\alpha-1}{\alpha-1} = 3+\alpha+\frac{2}{\alpha-1}$. □

Lemma 7.2.3 *The* SNAKE *algorithm is* $(\alpha+1)$*-competitive against current load.*

Proof: Given the way that reassignments are done by the SNAKE algorithm, there exists $p^{(\ell)}$ such that each machine contains either $p^{(\ell)}$ or $p^{(\ell)}+1$ jobs of J_ℓ. Moreover, the machines with $p^{(\ell)}+1$ jobs of level ℓ are precisely those with indices $z^{(\ell+1)}, z^{(\ell+1)}+1, \ldots, z^{(\ell)}-2, z^{(\ell)}-1$. If $z^{(\ell+1)} > z^{(\ell)}$, this sequence has to be interpreted as $z^{(\ell+1)}, \ldots, m, 1, \ldots, z^{(\ell)}-1$, while if $z^{(\ell+1)} = z^{(\ell)}$ there are no machines with $p^{(\ell)}+1$ jobs of level ℓ. Let C denote the set of "crossing" levels, i.e. the levels ℓ for which $z^{(\ell+1)} > z^{(\ell)}$. (In Figure 7-3 the only crossing level is level 1.) The key observation we will exploit is the fact that the sequence of machines with $p^{(\ell)}+1$ jobs of level ℓ is immediately followed by the corresponding sequence for level $\ell-1$ and so on. This implies that if machine i has $p^{(\ell)}+1$ jobs of level ℓ and $p^{(\ell')}+1$ jobs of level $\ell' < \ell$, then there must exist a crossing level among $\{\ell', \ell'+1, \cdots, \ell\}$.

We need some more notation. Assume $|J_\ell| = n^{(\ell)} = p^{(\ell)}m + k^{(\ell)}$, where $0 \le k^{(\ell)} < m$. We now analyze the load on machine i. Let $D_0 = \{\ell :$ machine i has $p^{(\ell)}$ jobs $\}$ and $D_1 = \{\ell :$ machine i has $p^{(\ell)}+1$ jobs $\}$. By Lemma 7.1.1, the load on machine i is at most

$$\begin{aligned}
\lambda_i &\le \alpha \sum_{j\in J} \frac{w_j}{m} - \sum_{\ell\in D_0} \frac{k^{(\ell)}}{m} w_{max}^{(\ell)} + \sum_{\ell\in D_1} \left(1 - \frac{k^{(\ell)}}{m}\right) w_{max}^{(\ell)} \\
&= \alpha \sum_{j\in J} \frac{w_j}{m} - \sum_{\ell} \frac{k^{(\ell)}}{m} w_{max}^{(\ell)} + \sum_{\ell\in D_1} w_{max}^{(\ell)}.
\end{aligned}$$

Observe that $k^{(\ell)} = z^{(\ell)} - z^{(\ell+1)}$ for $\ell \notin C$ and $k^{(\ell)} = z^{(\ell)} - z^{(\ell+1)} + m$ for $\ell \in C$. We can thus rewrite the upper bound on λ_i as

$$\begin{aligned}
\lambda_i &\le \alpha \sum_{j\in J} \frac{w_j}{m} - \sum_{\ell} \frac{z^{(\ell)} - z^{(\ell+1)}}{m} w_{max}^{(\ell)} - \sum_{\ell\in C} w_{max}^{(\ell)} + \sum_{\ell\in D_1} w_{max}^{(\ell)} \\
&= \alpha \sum_{j\in J} \frac{w_j}{m} - \frac{1}{m} \sum_{\ell} z^{(\ell)} (w_{max}^{(\ell)} - w_{max}^{(\ell-1)}) - \sum_{\ell\in C} w_{max}^{(\ell)} + \sum_{\ell\in D_1} w_{max}^{(\ell)} \\
&\le \alpha \sum_{j\in J} \frac{w_j}{m} - \sum_{\ell\in C} w_{max}^{(\ell)} + \sum_{\ell\in D_1} w_{max}^{(\ell)}.
\end{aligned}$$

The term involving $z^{(\ell)}$ has disappeared since $w_{max}^{(\ell)} - w_{max}^{(\ell-1)} > 0$. Now we use the crucial observation that the levels in C (the crossing levels) and in D_1 must alternate. Indeed, if we go through the sequences of machines with $p^{(\ell)}+1$ jobs of level ℓ in order of decreasing levels, we will alternate between machine i and crossings from machine m to machine 1. Therefore $\sum_{\ell \in D_1} w_{max}^{(\ell)} - \sum_{\ell \in C} w_{max}^{(\ell)} \leq w_{max}$ and thus,

$$\lambda_i \leq \alpha \sum_{j \in J} \frac{w_j}{m} + w_{max} \leq (\alpha + 1) LB.$$

□

Theorem 7.2.4 *When $r_j = aw_j$, the* SNAKE *algorithm with parameter $\alpha > 1$ is $(1+\alpha)$-competitive against current load and has a reassignment factor of $3 + \alpha + \frac{2}{\alpha - 1}$.*

We can see that there is a tradeoff between the competitive ratio and the reassignment factor. For example, for $\alpha = 2$, we have a competitive ratio of 3 (better than the ZIGZAG algorithm) and a reassignment factor of 7. The competitive ratio can be made arbitrarily close to two if we are willing to carry out a lot of reassignments. The best reassignment factor is $4 + 2\sqrt{2}$ and is obtained by setting $\alpha = 1 + \sqrt{2}$. Note that using our lower bound LB we cannot get a competitive ratio better than two. Consider an input which consists of $m + 1$ unit weight jobs arriving. In this case $LB = 1 + \frac{1}{m}$ but the optimum current load is 2.

We can also use a variation of the SNAKE algorithm for unit reassignment costs. Fix an integer parameter $\mu \geq 1$ and for all integers s form a snake out of levels $s\mu, \cdots, (s+1)\mu - 1$. The reassignment factor of the resulting MULTI-SNAKES algorithm now becomes 2μ. Using an analysis similar to that of Lemma 7.2.3 we obtain,

$$\lambda_i \leq \alpha \sum_{j \in J} \frac{w_j}{m} + w_{max} \left(1 + 1 + \frac{1}{\alpha^{\mu}} + \frac{1}{\alpha^{2\mu}} + \cdots\right) \leq \left(2 + \alpha + \frac{1}{\alpha^{\mu} - 1}\right) LB.$$

Theorem 7.2.5 *When $r_j = 1$, the* MULTI-SNAKES *algorithm with parameters μ and α is $(2 + \alpha + \frac{1}{\alpha^{\mu} - 1})$-competitive against current load and has a reassignment factor of 2μ.*

Notice that the competitive ratio can be made arbitrarily close to 3 but with a corresponding increase in the reassignment factor.

7.3 Arbitrary Reassignment Costs

In this section we generalize the 3.5981-competitive ZIGZAG algorithm from unit reassignment costs to arbitrary reassignment costs. Once again jobs are divided into levels according to their weights, level ℓ consisting of jobs whose weight w_j satisfies $\sqrt{3}^{\ell} \leq w_j < \sqrt{3}^{\ell+1}$. As for the ZIGZAG algorithm, two kinds of pointers, depending on the parity of the level number, are used to maintain the structure of Figure 7-1. However, what happens within a level is different. Indeed, we need to make sure that reassignments within a level are not too costly. For this purpose, within every level, we "load balance" the jobs according to their reassignment costs using the SNAKE algorithm. More precisely, within each level, jobs are divided into *blocks*. For a fixed parameter β, block b consists of jobs in level ℓ whose reassignment cost r_j satisfies $\beta^b \leq r_j < \beta^{b+1}$. We use a pointer for each block to maintain a block structure identical to the level structure of the SNAKE algorithm. (See Figure 7-5.)

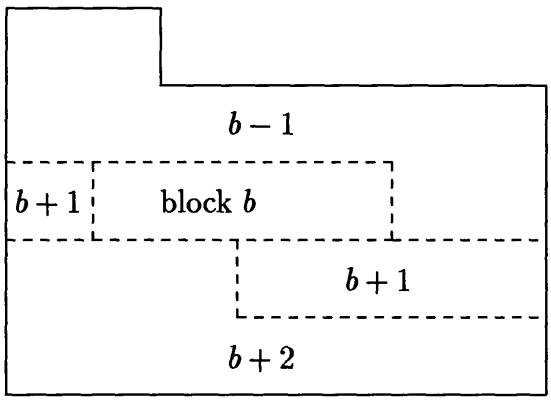

Figure 7-5: The block structure for level ℓ.

When a level ℓ job j arrives/departs, we use the SNAKE algorithm (using the reassignment costs as weights) to make appropriate reassignments between the blocks of level ℓ. To be more precise, we use the SNAKE algorithm for even levels, and a reverse SNAKE algorithm (in which the pointers go in the opposite direction) for

odd levels. The resulting algorithm, which we call the ZIGZAG-SNAKES algorithm, is still 3.5981-competitive. Indeed, the SNAKE algorithm has the important feature that the number of jobs on any two machines differs by at most one and that the machines with one additional job are the lowest indexed machines. Using the analysis of Lemma 7.2.2, we also obtain that the reassignment factor is $3 + \beta + \frac{2}{\beta - 1}$. The minimum value of this expression is $4 + 2\sqrt{2} < 6.8285$ and is achieved by setting $\beta = 1 + \sqrt{2}$.

Theorem 7.3.1 *For arbitrary reassignment costs and arbitrary weights, the* ZIGZAG-SNAKES *algorithm is* 3.5981*-competitive against current load and has a reassignment factor of* 6.8285.

Chapter 8

Related Machines

8.1 Greedy Algorithm for Related Machines

In this chapter we examine the related machines problem in which each machine i has a capacity cap_i and the load on machine i is equal to the total weight assigned to it divided by cap_i. By scaling and renumbering we can assume without loss of generality that $\text{cap}_1 \geq \text{cap}_2 \geq \cdots \geq \text{cap}_m = 1$. When the reassignment costs are proportional to the weights, Westbrook [69] has derived a 24-competitive algorithm against current load with a reassignment factor of 4. We shall consider arbitrary reassignment costs and propose two competitive algorithms against current load. The first has a competitive ratio that depends logarithmically on the largest capacity cap_1. Its reassignment factor is 6.8285. The second algorithm, described in Section 8.2, has a competitive ratio of 32 and a reassignment factor of 79.4.

We begin by presenting a GREEDY algorithm for the special case of unit weight jobs. When a job arrives, GREEDY assigns it to the machine whose resulting load is minimized. Thus, if there are n jobs present in the system and $T_i(n)$ jobs on machine i, the $(n+1)$st job will be assigned to the (or any) machine i minimizing $(T_i(n)+1)/\text{cap}_i$. This defines a sequence of machines $\{s_1, s_2, s_3, \cdots\}$ on which jobs are assigned; the ith job is assigned to machine i. If we have n jobs in the system and a job leaves from machine i, GREEDY then reassigns a job from machine s_n to machine i to fill in the gap. This ensures that the number of jobs on any machine is

as if $n-1$ jobs have arrived and no departure has occurred. Thus, the number $T_i(n)$ of jobs on machine i only depends on the total number n of active jobs but not on the sequence of arrivals and departures. Observe that for unit reassignment costs the reassignment factor of GREEDY is 2.

Theorem 8.1.1 *For unit weight jobs,* GREEDY *is optimal.*

Proof: Without loss of generality we assume that n jobs have arrived and no jobs have left. The GREEDY algorithm assigns them to machines $s_1, s_2, \cdots, s_n$. Let the number of jobs on machine i be $T_i(n)$ (or simply T_i for notational convenience). We show that the following two identities hold for $1 \leq i \leq m$:

$$\frac{T_i}{\text{cap}_i} \leq \frac{T_{s_n}}{\text{cap}_{s_n}}, \tag{8.1}$$

$$\frac{T_i+1}{\text{cap}_i} \geq \frac{T_{s_n}}{\text{cap}_{s_n}}. \tag{8.2}$$

To see that identity (8.1) holds, we consider the latest time at which a job, say j, was assigned to machine i. The construction of the sequence $\{s_1, s_2, \cdots\}$ guarantees that the resulting load (including j) on machine i is no bigger than that on machine s_n, i.e. $\frac{T_i}{\text{cap}_i} \leq \frac{T_{s_n}}{\text{cap}_{s_n}}$, otherwise j would have been assigned to machine s_n. If identity (8.2) did not hold for some i, i.e. $\frac{T_i+1}{\text{cap}_i} < \frac{T_{s_n}}{\text{cap}_{s_n}}$, then the nth job would have been assigned to machine i by the GREEDY algorithm. Hence identity (8.2) holds for all i.

We now show that the optimum load λ_{opt} is equal to $\frac{T_{s_n}}{\text{cap}_{s_n}}$. This is all that we need since identity (8.1) implies that the load due to GREEDY is $\frac{T_{s_n}}{\text{cap}_{s_n}}$. If $\lambda_{opt} < \frac{T_{s_n}}{\text{cap}_{s_n}}$, then in an optimum assignment machine s_n must have fewer than T_{s_n} jobs and so some machine $i \neq s_n$ must have at least $T_i + 1$ jobs. But identity (8.2) implies that $\frac{T_i+1}{\text{cap}_i} \geq \frac{T_{s_n}}{\text{cap}_{s_n}} > \lambda_{opt}$. Hence we have a contradiction. □

In the case of arbitrary job weights, Aspnes, Azar, Fiat, Plotkin and Waarts [2] have shown that the generalization of GREEDY is $\Theta(\log m)$-competitive if jobs never depart and reassignment is not allowed. Considering our model in which jobs depart, we divide the jobs into levels according to their weights. Level ℓ consists of jobs whose weight w_j satisfies $2^\ell \leq w_j < 2^{\ell+1}$. Let J_ℓ be the set of jobs in level ℓ,

$n^{(\ell)} = |J_\ell|$ and $L = \max\{\ell : J_\ell \neq \emptyset\}$. We treat jobs in J_ℓ as if they had weight 2^ℓ and assign them to machines using GREEDY. We call this algorithm the GREEDY ZIGZIG algorithm. The reassignment factor of GREEDY ZIGZIG is 2 if $r_j = 1$ for all j, and 3 if r_j is proportional to w_j. If we have arbitrary reassignment costs then we can modify the algorithm by dividing each level into blocks based on reassignment cost as for the ZIGZAG-SNAKES algorithm. The highest block will have jobs on machines $s_1, s_2, \ldots, s_k$, the next block on machines $s_{k+1}, \ldots$, and so on. We omit the details. We now bound the competitive ratio (of either version of the algorithm). Let $\tau = \lceil \log \text{cap}_1 \rceil + 1$. (If we did not assume $\text{cap}_m = 1$ we would set $\tau = \lceil \log(\text{cap}_{\max}/\text{cap}_{\min}) \rceil + 1$, where $\text{cap}_{\max}$ is the largest machine capacity and $\text{cap}_{\min}$ is the smallest.)

Theorem 8.1.2 GREEDY ZIGZIG *is* $(6 + 2\tau)$*-competitive against current load. Its reassignment factor is* 2 *for unit reassignments,* 3 *for proportional reassignments, and* 6.8285 *for arbitrary reassignments.*

Proof: The reassignment factors follow from the identical machine case. Let $C = \sum_{1 \leq i \leq m} \lfloor \text{cap}_i \rfloor$ and $n^{(\ell)} = p^{(\ell)}C + k^{(\ell)}$, where $0 \leq k^{(\ell)} \leq C - 1$. Let $\lambda_{opt}(k)$ be the (optimum) load resulting from the assignment of k unit weight jobs by GREEDY. Since the weight of any level ℓ job is at least 2^ℓ, $\max_{\ell \leq L}\{2^\ell \lambda_{opt}(k^{(\ell)})\}$ is a lower bound on the optimum load. Two other valid lower bounds are $(\sum_j w_j)/(\sum_i \text{cap}_i)$ and $2^L/\text{cap}_1$ (since the jobs in the last level must be assigned). Hence

$$LB := \max\left\{\max_{\ell \leq L}\{2^\ell \lambda_{opt}(k^{(\ell)})\}, \frac{\sum_j w_j}{\sum_i \text{cap}_i}, \frac{2^L}{\text{cap}_1}\right\}$$

is a lower bound on the current optimum load.

The load on each machine due to the first $p^{(\ell)}C$ jobs is at most $w_{max}^{(\ell)} p^{(\ell)}$ and the load due to the remaining $k^{(\ell)}$ jobs is at most $2^{\ell+1}\lambda_{opt}(k^{(\ell)})$. Observe that $\lambda_{opt}(k) \leq 1$ for $k \leq C$. Summing over all levels, we obtain that the load on machine i, λ_i, satisfies,

$$\lambda_i \leq \sum_{\ell \leq L} \left(w_{max}^{(\ell)} p^{(\ell)} + 2^{\ell+1} \lambda_{opt}(k^{(\ell)}) \right)$$

$$
\begin{aligned}
&\leq \sum_{\ell \leq L} 2w_{avg}^{(\ell)} \frac{n^{(\ell)}}{C} + \sum_{\ell=L-\tau+1}^{L} 2^{\ell+1} \lambda_{opt}(k^{(\ell)}) + \sum_{\ell \leq L-\tau} 2^{\ell+1} \lambda_{opt}(k^{(\ell)}) \\
&\leq 2\frac{\sum_j w_j}{C} + 2\tau LB + \sum_{\ell \leq L-\tau} 2^{\ell+1} \\
&\leq 4LB + 2\tau LB + \frac{2^L}{\mathrm{cap}_1}\left(1 + \frac{1}{2} + \frac{1}{2^2} + \ldots\right) \\
&\leq 4LB + 2\tau LB + 2LB \\
&= (6 + 2\tau)LB,
\end{aligned}
$$

where we have repeatedly used the definition of the lower bound LB, and the fact that $\sum_i \mathrm{cap}_i < 2C$ since $\mathrm{cap}_i < 2\lfloor \mathrm{cap}_i \rfloor$. □

8.2 Constant Factor Bounds for Related Machines

We now present an algorithm, BALANCE-RELATED, that is 32-competitive against current load and has a reassignment factor of 79.4. Note that the competitive ratio is only superior to that of GREEDY ZIGZIG when $\mathrm{cap}_{\mathbf{max}}/\mathrm{cap}_{\mathbf{min}} > 2^{12}$. To avoid complicated numerical calculations we shall initially describe an algorithm that has a reassignment factor of 127. Once again we assume that $\mathrm{cap}_1 \geq \mathrm{cap}_2 \geq \ldots \geq \mathrm{cap}_m$. Let λ be some constant. We start by considering an algorithm, BALANCE(λ), which keeps the load on each machine bounded by 8λ but which has the option to reject jobs. The framework of the algorithm is based on [69] which in turn is based on [2]. For simplicity we assume that for all jobs j, the ratio $\frac{r_j}{w_j}$ is a power of 8. This assumption is removed later. (See Theorem 8.3.2.) We divide the jobs into *classes* according to their ratio of reassignment cost to weight. Job j is in class c if $\frac{r_j}{w_j} = 8^c$. The algorithm of [69] dealt with the special case in which all of the jobs are in one class. We say that class c_1 is smaller than class c_2 if $c_1 < c_2$. Let $W_i^c(t)$ be the sum of the weights of class c jobs assigned to machine i. We shall also define a quantity $M_i^c(t)$ which is an upper bound on $W_i^c(t)$. The difference between $M_i^c(t)$ and $W_i^c(t)$ is a measure of how much weight from class c has left machine i. For clarity we normally drop the dependence on t. The aim of the algorithm is to maintain the following two

invariants.

- **Invariant 1:** $\sum_c W_i^c \leq 8\lambda \text{cap}_i$, for all $1 \leq i \leq m$.
- **Invariant 2:** $\sum_{i=1}^{k} \sum_{c' \leq c} (M_i^{c'} - 2W_i^{c'}) \leq 0$ for all $1 \leq k \leq m$ and for all c.

Invariant 1 provides a bound on the load on each machine. Invariant 2 ensures that not too much weight has left the system and so the optimum load must be high. If either of the invariants is ever violated then we carry out a rebalancing step in which some of the jobs are reassigned.

We maintain a list which consists of jobs that need to be assigned. Let A be the set of jobs in the list. Initially the list is empty. We say that we are in a *stable* state if the list is empty and the invariants are satisfied. The basic structure of BALANCE(λ) is as follows. The details are given in section 8.2.2. When a job arrives in the system it is placed in the list. The scheduler then takes the job from the list and attempts to insert it onto a machine. If the attempt is successful then the invariants may be violated and so rebalancing is carried out. When a job leaves the system it is simply removed from the machine which was processing it. Rebalancing is then performed.

A rebalancing step will consist of removing jobs from machines and inserting them into the list until the invariants are satisfied. We then remove a job from the list and reassign it to a machine. This rebalancing is repeated until we reach a stable state. We assume that after each job arrival or departure we reach a stable state before the next arrival or departure. We now describe the rebalancing procedures in detail. Procedure INSERTION-REBALANCE(i) is used to make sure that invariant 1 is not violated for machine i. Procedure DELETION-REBALANCE is used to make sure that invariant 2 is not violated.

8.2.1 Rebalancing Procedures

procedure INSERTION-REBALANCE(i)

1. If $\sum_c W_i^c \leq 8\lambda \text{cap}_i$ then stop. (Invariant 1 is satisfied.)
2. Let c' be the smallest class that has jobs on machine i.

3. Set $M_i^{c'} = W_i^{c'} = 0$, remove all of the class c' jobs from machine i and add them to the list.
4. Repeat.

procedure DELETION-REBALANCE

1. Let $\tilde{c} = \min\left\{c : \exists k \sum_{i=1}^{k} \sum_{c' \leq c} \left(M_i^{c'} - 2W_i^{c'}\right) > 0\right\}$.
2. If there is no such $\tilde{c}$ then stop. (Invariant 2 is satisfied.)
3. Let $\tilde{i} = \max\left\{k : \sum_{i=1}^{k} \sum_{c' \leq \tilde{c}} \left(M_i^{c'} - 2W_i^{c'}\right) > 0\right\}$.
4. For all $i \leq \tilde{i}$ and for all $c' \leq \tilde{c}$, set $M_i^{c'} = 0$ and $W_i^{c'} = 0$. Remove each job from a class $c' \leq \tilde{c}$ currently on a machine $i \leq \tilde{i}$ and add it to the list.
5. Repeat.

8.2.2 The Algorithm BALANCE(λ)

We now define the algorithm BALANCE(λ). It consists of three procedures: one for insertion, one for deletion and one for reinsertion from the list. When a job arrives in the system it is placed in the list and then inserted using the insertion procedure. When a job leaves the system the deletion procedure is called.

Insertion: Suppose that we wish to insert job j from class c. Say that machine i *accepts* job j if,

1. $\frac{w_j}{\text{cap}_i} \leq \lambda$, and
2. $\frac{1}{\text{cap}_i}\left(\sum_{\hat{c} \geq c} W_i^{\hat{c}} + w_j\right) \leq 4\lambda$.

If there is no such i, then BALANCE(λ) *rejects* job j. Otherwise we find the maximum such i, assign job j to machine i, increase W_i^c by w_j and set $M_i^c = \max\{W_i^c, M_i^c\}$.

It is possible that machine i now has load greater than 8λ. To deal with this we run INSERTION-REBALANCE(i). This rebalancing may remove some jobs from machine i and so we run DELETION-REBALANCE. Both rebalancing procedures may add some jobs to the list. These jobs need to be reassigned and so we run the reinsertion procedure (see below).

Deletion: Suppose that job j of class c is deleted from machine i and leaves the system. We decrease W_i^c by w_j and run DELETION-REBALANCE. We then run the reinsertion procedure since jobs may have been added to the list.

Reinsertion: Suppose that the list is non-empty. We find $\hat{c}$, the maximum class that contains a job in the list. We select any class $\hat{c}$ job from the list and insert it using the insertion procedure. If the list is empty then we are in a stable state and so we terminate.

Since the insertion of a job may require the reinsertion of another job it is not clear that we will ever terminate. We will however be able to use the bounds on reassignment cost to show that in fact we do terminate in a stable state. (See Corollary 8.3.3.)

Proposition 8.2.1 *Suppose that* INSERTION-REBALANCE(i) *adds jobs to the list. Then, immediately after the termination of* INSERTION-REBALANCE(i), $4\lambda cap_i < \sum_c W_i^c \leq 8\lambda cap_i$.

Proof: The upper bound is immediate from the definition of INSERTION-REBALANCE(i). To prove the lower bound first note that $W_i^c \leq 4\lambda \text{cap}_i$ for all c. This follows from the criteria for accepting a job. Since INSERTION-REBALANCE(i) removes jobs one class at a time, $\sum_c W_i^c$ cannot become smaller than or equal to $8\lambda \text{cap}_i - 4\lambda \text{cap}_i = 4\lambda \text{cap}_i$.

□

Proposition 8.2.2 *Suppose job j is rejected by the insertion procedure. Then $\lambda^* \geq \lambda$, where λ^* is the maximum load in the optimum assignment of the active jobs (including j).*

Proof: Suppose that j is a job from class c. Let $M_i = \sum_{c'} M_i^{c'}$ and let $W_i = \sum_{c'} W_i^{c'}$. Let k be minimal such that $M_k/\text{cap}_k < 2\lambda$. If there is no such k, define $k = m + 1$.

Suppose $k = 1$. Since j is rejected either $w_j > \lambda \text{cap}_1$ or $\frac{1}{\text{cap}_1}\left(\sum_{\hat{c} \geq c} W_1^{\hat{c}} + w_j\right) > 4\lambda$. Assume that the second condition holds. Then since $M_1 < 2\lambda \text{cap}_1$, we have that

$\sum_{\hat{c} \geq c} W_1^{\hat{c}} < 2\lambda \text{cap}_1$, implying that $w_j > 2\lambda \text{cap}_1$. Thus in either case, $\lambda^* \geq w_j/\text{cap}_1 > \lambda$.

Suppose $k > 1$. For all machines $i < k$, $M_i/\text{cap}_i \geq 2\lambda$. Since invariant 2 is maintained we know that $\sum_{h=1}^{i}(M_h - 2W_h) \leq 0$, for all $1 \leq i \leq k$. Let X be the set of jobs that are currently assigned to a machine $i < k$. Now consider an optimal assignment of jobs to machines.

- **Case 1:** In the optimal assignment all the jobs in X are assigned to machines $i < k$. Let W_i^* be the weight on i in the optimal assignment. It is clear that $\lambda^* \geq W_i^*/\text{cap}_i$ for all $i \leq m$. Hence

$$\lambda^* \sum_{h=1}^{k-1} \text{cap}_h \geq \sum_{h=1}^{k-1} W_h^* \geq \sum_{h=1}^{k-1} W_h \geq \sum_{h=1}^{k-1} M_h/2 \geq \lambda \sum_{h=1}^{k-1} \text{cap}_h.$$

- **Case 2:** In the optimal assignment there is a job $x \in X$ that is assigned to a machine $i \geq k$. Suppose that x is in class $\tilde{c}$. Let t' be the time that x was last reassigned by BALANCE(λ) and let t be the current time. At time t' job x was assigned by BALANCE(λ) to a machine $\ell < k$ and so either $w_x/\text{cap}_k > \lambda$ or $\left(\sum_{\hat{c} \geq \tilde{c}} M_k^{\hat{c}}(t') + w_x\right)/\text{cap}_k > 4\lambda$.

  We claim that in any case the first condition holds ($w_x/\text{cap}_k > \lambda$). Assume for the moment that the second condition holds. If $\sum_{\hat{c} \geq \tilde{c}} M_k^{\hat{c}}$ has not decreased between t' and t then $\sum_{\hat{c} \geq \tilde{c}} M_k^{\hat{c}}(t') \leq \sum_{\hat{c} \geq \tilde{c}} M_k^{\hat{c}}(t) \leq M_k(t) < 2\lambda \text{cap}_k$, by definition of k. This implies that $w_x > 2\lambda \text{cap}_k$. On the other hand, if $\sum_{\hat{c} \geq \tilde{c}} M_k^{\hat{c}}$ has decreased between t' and t, part of the decrease must have been caused by DELETION-REBALANCE because of Proposition 8.2.1 and the fact that $\sum_{\hat{c} \geq \tilde{c}} M_k^{\hat{c}}(t) \leq M_k(t) < 2\lambda \text{cap}_k$. But if DELETION-REBALANCE affects the value of $\sum_{\hat{c} \geq \tilde{c}} M_k^{\hat{c}}$ then all of the class $\tilde{c}$ jobs on machines $\ell \leq k$ are reassigned. Hence x is reassigned at a time between t' and t, contradicting the definition of t'. Thus, in any case, $w_x/\text{cap}_k > \lambda$. This implies that $\lambda^* \geq w_x/\text{cap}_j \geq w_x/\text{cap}_k > \lambda$.

□

Observe that the above proposition would still hold if we were to strengthen the second condition for machine i to accept job j to $\frac{1}{\mathbf{cap}_i}\left(\sum_{\hat{c}\geq c} W_i^{\hat{c}} + w_j\right) \leq 3\lambda$. This observation will be useful in the next section.

8.2.3 The Load Balancing Algorithm

To obtain an algorithm competitive against current load that does not reject jobs we use a system of levels. Level ℓ is parameterized by $\lambda_\ell = 2^\ell$ and uses the algorithm BALANCE(λ_ℓ). We say that a job is in level ℓ if it was assigned to its current machine by BALANCE(λ_ℓ). In order to make sure that the algorithm is competitive we need to maintain a third invariant, the idea of which is due to Westbrook [69]. Let L be the maximum occupied level.

- **Invariant 3:** There exists a job in level L that would be rejected by BALANCE(λ_{L-1}) if an attempt were made to insert it into level $L-1$.

Invariant 3 is maintained using the following procedure.

procedure GLOBAL-REBALANCE

1. Let L be the maximum occupied level, let i be the minimum indexed machine containing a job in level L and let c be the minimum occupied class on machine i in level L.
2. Select any job j of class c assigned to machine i in level L.
3. Attempt to insert j into level $L-1$ using the following small modification of BALANCE(λ_{L-1}). Replace the bound "$\leq 4\lambda_{L-1}$" in the criteria for acceptance by "$\leq 3\lambda_{L-1}$".
4. If job j is rejected then stop. (Invariant 3 is satisfied.)
5. Remove job j from machine i in level L. Decrease $M_{L,i}^c$ and $W_{L,i}^c$ by w_j where $M_{L,i}^c$ and $W_{L,i}^c$ are the values of M_i^c and W_i^c respectively in level L.
6. Insert job j into level $L-1$ using the modified version of BALANCE(λ_{L-1}).
7. Run DELETION-REBALANCE on level L.
8. Repeat.

Observe that invariants 1 and 2 are still satisfied in level L after GLOBAL-REBALANCE has terminated. Invariant 1 is satisfied because W values are only decreased. Invariant 2 is satisfied because DELETION-REBALANCE is run on level L in step 7. The complete algorithm can now be described. We shall call the algorithm BALANCE-RELATED. Suppose that job j needs to be inserted from the list. Let $\hat{\ell} = \min\{\ell :$ BALANCE(λ_ℓ) accepts job $j\}$. Job j is inserted using algorithm BALANCE$(\lambda_{\hat{\ell}})$. Now suppose that job j' is removed from level ℓ. The algorithm BALANCE(λ_ℓ) is used to carry out rebalancing within level ℓ. To ensure that invariant 3 is maintained the procedure GLOBAL-REBALANCE is now applied.

We say that algorithm BALANCE-RELATED is in a stable state if all of the algorithms BALANCE(λ_ℓ) are in a stable state. We assume that no jobs arrive in or depart from the system unless BALANCE-RELATED is in a stable state.

Theorem 8.2.3 *Algorithm* BALANCE-RELATED *is 32-competitive against current load.*

Proof: Since invariant 3 is maintained there is always a job in level L that cannot be inserted into level $L-1$. By the observation following the proof of Proposition 8.2.2, this proposition still holds for the modified version of BALANCE(λ_{L-1}) used in GLOBAL-REBALANCE. Furthermore, Proposition 8.2.2 also holds for BALANCE(λ_L) even though an M value is decreased in step 5 of GLOBAL-REBALANCE. To show this, we need to look carefully at what happens in Case 2 of the proof. There, between t' and t, the decrease of $\sum_{\hat{c}\geq\tilde{c}} M_k^{\hat{c}}$ could not have been caused by GLOBAL-REBALANCE. Indeed, GLOBAL-REBALANCE only moves a job from the least indexed machine in level L which, in this case, cannot be k since job x is occupying machine $\ell < k$. Thus the decrease is still caused by DELETION-REBALANCE, and the proof follows. Hence Proposition 8.2.2 implies that the optimum load is at least $\lambda_{L-1} = \frac{1}{2}\lambda_L$. Since invariant 1 is maintained for all levels the load on machine j in level ℓ is bounded by $8\lambda_\ell$. This implies that the total load on machine i is bounded by $8\lambda_L(1+\frac{1}{2}+\frac{1}{4}+\ldots) = 16\lambda_L$. Thus the competitive ratio is $16\lambda_L/\frac{1}{2}\lambda_L = 32$. □

8.3 Reassignment Analysis for Algorithm BALANCE-RELATED

8.3.1 Definitions

We need to bound the total reassignment cost incurred by BALANCE-RELATED. This will be achieved by using four potential functions, Υ, Φ, Ψ and Ω. Each is used to pay for a particular type of reassignment. The function Υ is associated with INSERTION-REBALANCE, Φ with DELETION-REBALANCE and Ψ with GLOBAL-REBALANCE. The function Ω is used to pay for the reinsertion of a job after it has been in the list.

Let $M^c_{\ell,i}$ and $W^c_{\ell,i}$ be the values of M^c_i and W^c_i respectively in level ℓ. In order to bound the reassignment performed by GLOBAL-REBALANCE and INSERTION-REBALANCE we need to define two more values, $N^c_{\ell,i}$ and $P^c_{\ell,i}$. The difference between $N^c_{\ell,i}$ and $\sum_{\hat{c}\geq c} W^{\hat{c}}_{\ell,i}$ is a measure of how much weight from classes $\hat{c} \geq c$ has left machine i in level ℓ since ℓ was last the top level. The value of $P^c_{\ell,i}$ is a measure of how much weight from classes strictly larger that c has been added to machine i in level ℓ. Suppose that j is a class c job which is inserted onto machine i in level ℓ. For all $c' \leq c$ we set $N^{c'}_{\ell,i} = \max\{N^{c'}_{\ell,i}, \sum_{\hat{c}\geq c'} W^{\hat{c}}_{\ell,i}\}$. Then for all $c' < c$ we update $P^{c'}_{\ell,i}$ as follows,

$$P^{c'}_{\ell,i} \leftarrow P^{c'}_{\ell,i} + 8^{-(c-c')} r_j.$$

If GLOBAL-REBALANCE removes the class c job j from machine i in the top level L then we set $N^{c'}_{L,i} = N^{c'}_{L,i} - w_j$ for all $c' \leq c$. If INSERTION-REBALANCE removes all the class c jobs from machine i in level ℓ then we set $P^c_{\ell,i} = 0$.

Let,

$$\begin{aligned}
\Upsilon &= \sum_{\ell,i,c} 16 P^c_{\ell,i}, \\
\Phi &= \sum_{\ell,i,c} 8 \cdot 8^c M^c_{\ell,i}, \\
\Psi &= \sum_{\ell,i,c} 4 \cdot 8^c N^c_{\ell,i},
\end{aligned}$$

$$\Omega = \sum_{j \in A} \frac{111}{7} r_j.$$

(Recall that A is the set of jobs in the list.)

8.3.2 Amortized Costs for Job Arrival, Job Departure and Job Insertion

When job j arrives in the system it is placed in the list. We shall use Δ to represent change in value. It is clear that $\Delta\Omega = 111r_j/7$ and none of the other potential functions change. Hence the total amortized cost of the arrival is $111r_j/7 < 15.86r_j$. When a job departs from the system none of the potential functions increase and so the amortized cost is at most 0.

Now suppose that j is a class c job which is taken from the list and inserted onto machine i in level ℓ. The actual cost of the insertion is r_j. From the definitions we know that $\Delta M^c_{\ell,i} \leq w_j$, $\Delta N^{c'}_{\ell,i} \leq w_j$ for all $c' \leq c$ and $\Delta P^{c'}_{\ell,i} = 8^{-(c-c')}r_j$ for all $c' < c$. No other M, N or P values are affected. Since job j is removed from the list we also know that $\Delta\Omega = -111r_j/7$. Therefore,

$$\begin{aligned}
\Delta\Upsilon &= 16\left(\frac{1}{8} + \frac{1}{64} + \ldots\right) r_j = \frac{16}{7} r_j \\
\Delta\Phi &\leq 8 \cdot 8^c w_j = 8r_j \\
\Delta\Psi &\leq 4 \cdot 8^c w_j \left(1 + \frac{1}{8} + \frac{1}{64} + \ldots\right) = \frac{32}{7} r_j \\
\Delta\Omega &= -\frac{111}{7} r_j.
\end{aligned}$$

Hence the total amortized cost of the insertion is at most $r_j(1 + \frac{16}{7} + 8 + \frac{32}{7} - \frac{111}{7}) = 0$.

8.3.3 Amortized Cost of GLOBAL-REBALANCE.

Let L be the top level. Suppose that j is a class c job which is reassigned from machine i in level L to machine k in level $L-1$. Recall that i is the minimum indexed machine in level L and c is the minimum occupied class on machine i in level L. The definitions imply that $\Delta M^c_{L,i} = -w_j$, $\Delta M^c_{L-1,k} \leq w_j$, $\Delta N^{c'}_{L,i} = -w_j$ for all

$c' \leq c$, $\Delta N^{c'}_{L-1,k} \leq w_j$ for all $c' < c$ and $\Delta P^{c'}_{L-1,k} = 8^{-(c-c')} r_j$ for all $c' < c$.

Lemma 8.3.1 $\Delta N^{c}_{L-1,k} = 0$.

Proof: Let t be the current time and let t' be the last time that j was inserted from the list. Between times t' and t job j has been in levels L or higher. At time t job j is inserted onto machine k by GLOBAL-REBALANCE. Hence $w_j \leq \lambda_{L-1}\mathrm{cap}_k$ and $\sum_{\hat{c}\geq c} W^{\hat{c}}_{L-1,k}(t) + w_j \leq 3\lambda_{L-1}\mathrm{cap}_k$. But at time t' job j was rejected by machine k in level $L-1$ when it was inserted from the list. Hence $\sum_{\hat{c}\geq c} W^{\hat{c}}_{L-1,k}(t') + w_j > 4\lambda_{L-1}\mathrm{cap}_k$. This means that between times t' and t the value of $\sum_{\hat{c}\geq c} W^{\hat{c}}_{L-1,k}$ has decreased by at least $\lambda_{L-1}\mathrm{cap}_k \geq w_j$. The value of $N^{c}_{L-1,k}$ has not decreased between times t' and t because level $L-1$ has never been the top level during this time interval. Note that $N^{c}_{L-1,k}$ is never smaller than $\sum_{\hat{c}\geq c} W^{\hat{c}}_{L-1,k}$. Thus just before job j is inserted by GLOBAL-REBALANCE, $N^{c}_{L-1,k} - \sum_{\hat{c}\geq c} W^{\hat{c}}_{L-1,k} \geq w_j$. Hence $N^{c}_{L-1,k}$ does not increase. □

The M, N and P values that were not considered above do not change. No jobs are added to the list. Therefore,

$$\begin{aligned}
\Delta\Upsilon &= 16(\frac{1}{8} + \frac{1}{64} + \ldots)r_j = \frac{16}{7} r_j \\
\Delta\Phi &\leq 0 \\
\Delta\Psi &\leq -4 \cdot 8^{c} w_j = -4 r_j \\
\Delta\Omega &= 0.
\end{aligned}$$

Hence the total amortized cost of the insertion is at most $r_j(1 + \frac{16}{7} - 4) < 0$.

8.3.4 Amortized Cost of DELETION-REBALANCE

Suppose that DELETION-REBALANCE removes all the jobs from classes $c, c-1, c-2\ldots$ that are on machines $1, 2, \ldots, \tilde{i}$ in level ℓ. (For simplicity we shall drop the dependence on ℓ.) Then,

$$\sum_{c'\leq c} \sum_{i=1}^{\tilde{i}} \left(M_i^{c'} - 2W_i^{c'} \right) > 0 \tag{8.3}$$

and,

$$\sum_{c' \leq \tilde{c}} \sum_{i=1}^{\tilde{i}} \left(M_i^{c'} - 2W_i^{c'} \right) \leq 0, \tag{8.4}$$

for all $\tilde{c} < c$. The second formula holds because we always choose the smallest class for which invariant 2 is violated. Hence, multiplying (8.3) by 8^c and subtracting appropriate multiples of (8.4) for all $\tilde{c} < c$, we derive that

$$\begin{aligned} 8^c \sum_{i=1}^{\tilde{i}} (M_i^c - 2W_i^c) &\geq -\sum_{c'<c} 8^{c'} \sum_{i=1}^{\tilde{i}} \left(M_i^{c'} - 2W_i^{c'} \right) \\ \Rightarrow \sum_{c' \leq c} 8^{c'} \sum_{i=1}^{\tilde{i}} M_i^{c'} &\geq 2 \sum_{c' \leq c} 8^{c'} \sum_{i=1}^{\tilde{i}} W_i^{c'} \\ &= 2 \sum_{c' \leq c} \sum_{i=1}^{\tilde{i}} R_i^{c'}, \end{aligned}$$

where R_i^c is the total reassignment cost of class c jobs on machine i. The procedure DELETION-REBALANCE sets $M_i^{c'} = 0$ for all $i \leq \tilde{i}$ and for all $c' \leq c$. The other M values do not change and the N and P values are unaffected. The jobs that are removed are added to the list. Therefore,

$$\begin{aligned} \Delta\Upsilon &= 0 \\ \Delta\Phi &\leq -8 \cdot 2 \sum_{c' \leq c} \sum_{i=1}^{\tilde{i}} R_i^{c'} \\ \Delta\Psi &= 0 \\ \Delta\Omega &= \frac{111}{7} \sum_{c' \leq c} \sum_{i=1}^{\tilde{i}} R_i^{c'}. \end{aligned}$$

Hence the amortized cost of the rebalancing is at most $(-16 + \frac{111}{7}) \sum_{c' \leq c} \sum_{i=1}^{\tilde{i}} R_i^{c'} < 0$.

8.3.5 Amortized Cost of INSERTION-REBALANCE

Suppose that INSERTION-REBALANCE(i) removes all the jobs from class c on machine i in level ℓ. (We shall drop the dependence on ℓ.) Let these jobs have total reassignment cost R_i^c and total weight W_i^c. Let t be the current time and let t' be the time at which the algorithm last assigned a class c job to machine i. We know

that $\sum_{\hat{c}\geq c} W_i^{\hat{c}}(t') \leq 4\lambda \text{cap}_i$ since a class c job was assigned to machine i at time t'. This implies that $W_i^c(t) \leq 4\lambda \text{cap}_i$. We also know that $\sum_{\hat{c}\geq c} W_i^{\hat{c}}(t) \geq 8\lambda \text{cap}_i$. Hence between times t' and t the algorithm must have inserted at least $4\lambda \text{cap}_i$ weight from classes $\hat{c} > c$ onto machine i. When a job j from a class $\hat{c} > c$ is inserted onto machine i, P_i^c increases by $8^{-(\hat{c}-c)} r_j = 8^c w_j$. P_i^c has not been set to 0 since time t' and so at time t,

$$\begin{aligned} P_i^c &\geq 8^c \cdot 4\lambda \text{cap}_i \\ &\geq 8^c W_i^c \\ &= R_i^c. \end{aligned}$$

Thus $\Delta P_i^c \leq -R_i^c$. The other P values do not change and the M and N values are unaffected. The jobs that are removed are added to the list. Therefore,

$$\begin{aligned} \Delta\Upsilon &\leq -16R_i^c \\ \Delta\Phi &= 0 \\ \Delta\Psi &= 0 \\ \Delta\Omega &= \frac{111}{7} R_i^c. \end{aligned}$$

Hence the amortized cost of the rebalancing is at most $(-16 + 111/7)R_i^c < 0$.

Theorem 8.3.2 *The reassignment factor of the algorithm is* $\frac{111}{7} < 15.86$. *If we allow class c jobs to satisfy* $8^c \leq r_j/w_j < 8^{c+1}$ *then the reassignment factor becomes* $\frac{888}{7} < 127$.

Proof: Let $C(t)$ be the total reassignment cost that the algorithm has incurred by time t. Let $S(t)$ be the sum of the reassignment costs of jobs that have arrived in the system at time t (regardless of whether or not they have departed). Let $\Upsilon(t)$, $\Phi(t)$, $\Psi(t)$ and $\Omega(t)$ be the values of Υ, Φ, Ψ and Ω respectively at time t. The above results show that the amortized cost incurred when job j arrives is at most $111r_j/7$ and the amortized cost for deleting job j, reinserting job j or carrying out a

rebalancing procedure is at most 0. In other words,

$$C(t) + \Upsilon(t) + \Phi(t) + \Psi(t) + \Omega(t) \leq \frac{111}{7} S(t)$$

for all times t. It is easy to see that the potential functions are always non-negative. The reassignment factor of 111/7 follows. If jobs in class c satisfy $8^c \leq r_j/w_j < 8^{c+1}$ then for every step of the algorithm the reassignment cost incurred is at most 8 times the cost in the previous case. Hence the reassignment factor is 888/7. □

Corollary 8.3.3 *After each job arrival and job departure the algorithm* BALANCE-RELATED *reaches a stable state.*

Proof: Suppose that BALANCE-RELATED does not reach a stable state. Since there are finitely many jobs in the system there must be a job j that is reassigned arbitrarily many times. We are assuming that no jobs arrive if BALANCE-RELATED is not in a stable state. Hence $C(t)$ increases to an arbitrarily high value whereas $S(t)$ does not increase. (Recall that we assumed that $r_j > 0$ for all j.) Therefore the condition of Theorem 8.3.2 will eventually be violated. □

8.3.6 Improving the Reassignment Factor

By altering the definition of class we can obtain an improved reassignment factor. Let $\varepsilon = 10.4$ and say that job j is in class c if $\varepsilon^c \leq r_j/w_j < \varepsilon^{c+1}$. Let,

$$\begin{aligned}
\Upsilon &= \sum_{\ell,i,c} a_1 P^c_{\ell,i}, \\
\Phi &= \sum_{\ell,i,c} a_2 \varepsilon^c M^c_{\ell,i}, \\
\Psi &= \sum_{\ell,i,c} a_3 \varepsilon^c N^c_{\ell,i}, \\
\Omega &= \sum_{j \in A} a_4 r_j,
\end{aligned}$$

where $a_1 = \frac{4\varepsilon^2 - 6\varepsilon + 2}{\varepsilon^2 - 6\varepsilon + 3}$, $a_2 = \frac{a_1}{2}$, $a_3 = 1 + \frac{a_1}{\varepsilon - 1}$ and $a_4 = a_1$.

The proof of the following theorem is almost identical to the proofs of Theorem 8.2.3 and Theorem 8.3.2. Details are omitted.

Theorem 8.3.4 *Algorithm* BALANCE-RELATED *with the revised definition of class is* 32 *competitive against current load and has a reassignment factor of* 79.4.

Remarks. For ease of presentation we have assumed that BALANCE-RELATED is able to work with infinitely many job classes and infinitely many levels. Of course, in an implementation, the algorithm would only maintain the finitely many classes and levels that can actually be occupied by jobs. We omit the details since they are simple but cumbersome to describe.

We have also assumed that all reassignment costs are strictly positive. If some reassignment costs are zero then we can reassign all such jobs after each arrival or departure without affecting the reassignment cost incurred.

Part III

Disk Scheduling

Chapter 9

Disk Scheduling

9.1 Introduction

We begin our study of the disk scheduling problem with a formal definition of the problem. We model a computer disk as an annulus whose radial distance between the inner and outer circles is 1. The disk rotates at a constant rate. A movable *disk head* travels in and out radially in order to access locations on the disk. (In our presentation we consider the motion of the head relative to the disk and so the head not only moves radially but also moves around the disk at a constant rate.) An instance of the disk scheduling problem consists of a set of n locations on the disk. These n locations represent *requests* to be serviced by the disk head. To *service* a request, the disk head must be at the request and have no radial movement. A solution to the disk problem is a path of the disk head that services all n requests. An optimal solution is one that requires the minimum number of rotations.

The Reachability Function Associated with a disk drive is a function $f(\theta)$, which we call the *reachability function.* In a rotation through angle θ, the function $f(\theta)$ represents the maximum radial distance the head can travel when it starts and ends with no radial movement. Since the annulus has thickness 1, we have $0 \leq f(\theta) \leq 1$. For convenience, we let $f(\theta) = 0$ for $\theta \leq 0$. Thus, from any starting-point the function $f(\theta)$ defines the reachable region in the θ-r plane; we call this region the *reachability*

cone. The reachability function f has the following properties.

1. Function f is nondecreasing since given more time the disk head can visit a larger fraction of the disk. That is, $f' \geq 0$ where f' is the first derivative of f. (We assume throughout that the reachability function is differentiable.)

2. Function f is convex, implying that the slope of f is nondecreasing. The intuition for convexity is as follows. The head accelerates as much as possible and stays at the maximum radial speed as long as possible (if the maximum speed is reached) before decelerating.

3. Properties 1 and 2 imply that $f(\theta + \theta') \geq f(\theta) + f(\theta')$ for $\theta, \theta' \geq 0$.

We define t_{fullseek} to be the minimum number of rotations (not necessarily integral) required for the head to travel the entire radial distance. That is, $2\pi t_{\text{fullseek}} = \text{argmin}_\theta f(\theta) = 1$. On modern disks $1 \leq t_{\text{fullseek}} < 3$, and usually $t_{\text{fullseek}} < 2$.

The Representation of the Disk and the Requests For ease of presentation we view the disk as a $2\pi \times 1$ rectangle. Each request R_i is specified by coordinates (θ_i, r_i), where $0 \leq r_i \leq 1$ and $0 \leq \theta_i < 2\pi$. The *distance* between two requests $R_i = (\theta_i, r_i)$ and $R_j = (\theta_j, r_j)$ is defined by,

$$d(R_i, R_j) = d((\theta_i, r_i), (\theta_j, r_j)) = \min\{\text{integers } k : f(\theta_j - \theta_i + 2k\pi) \geq |r_j - r_i|\}.$$

In other words, the distance from request R_i to R_j is equal to the number of times that the head must cross the line $\theta = 0$ when traveling from R_i to R_j.[1] Note that this distance is *asymmetric*. To reflect the "rotational nature" of the disk we also use $(\theta_i + 2k\pi, r_i)$ to denote request R_i, and we sometimes represent multiple copies of the disk by a $2k\pi \times 1$ rectangle.

[1]The distance between R_i and R_j could be defined as the angular distance through which the head must travel in order to service R_i and then R_j. However, our integral definition of distance facilitates many of our later proofs.

The *disk graph* is a directed graph whose vertices are the requests and whose directed edges are the ordered pairs of vertices. The weight on the directed edge R_iR_j is $d(R_i, R_j)$.

9.1.1 Our Results

The results for disk scheduling that we present in this thesis are as follows.

- Let T_{opt} be the minimum number of rotations in an optimal schedule in which the disk head starts and ends at the same place. For general reachability functions we show how to service all of the requests in at most $\frac{3}{2}T_{\text{opt}} + a$ rotations, where a is a term that depends solely on the reachability function, not on the number of requests. (See Section 9.2.)

- For general reachability functions we show that the disk-scheduling problem is NP-hard. (See Section 9.3.)

- Now suppose that the reachability function is linear. Let T_{opt} be the minimum number of rotations in an optimal schedule in which the disk head must start at $(0,0)$ and end at $(0,1)$. We show how to construct an optimal schedule. (See Section 9.4.)

- We provide an optimal solution to the Asymmetric Traveling Salesman Problem with the triangle inequality (ATSP-Δ) in which all distances are either 0 or α for some value $\alpha > 0$. This extends to a $\frac{\beta}{\alpha}$-approximation algorithm for the case in which all distances are either 0 or else lie between α and β for some values $0 < \alpha < \beta$. This latter result leads to another approximation algorithm for disk scheduling with general reachability functions. (See Section 9.5.)

- For the on-line problem in which we wish to service requests at a high rate (i.e. maximize the throughput), we present heuristics with good look-ahead properties. (See Section 9.6.)

9.1.2 Related Work

Since disks have been used for many years as secondary storage devices, the problem of disk scheduling has received a great deal of attention. Most early papers, (e.g., [32, 16, 67, 27, 50, 70]) study the on-line problem and focus primarily on the algorithms first-come-first-served (FCFS), CSCAN, shortest-seek-first (SSF) and modifications and generalizations of these algorithms.

As its name suggests, FCFS always serves the request that has been in the system the longest. Although it treats the requests fairly, numerous studies have shown that it has low throughput.

The algorithm CSCAN is the disk scheduling algorithm most widely used in practice. In CSCAN, the head starts at one side of the disk and travels to the other, servicing all the requests in a track as the head passes over it. Once it has completed this pass it performs one full seek back to its starting position and repeats. The throughput of CSCAN is usually higher than that of FCFS and it has good fairness properties. Also, if a subset of the requests are arranged on the disk sequentially then CSCAN services them efficiently. (This situation is not uncommon in practice.) A close relative of CSCAN is the SCAN algorithm in which the head services requests as it travels in *both* directions across the disk. (We can regard CSCAN as a unidirectional version of SCAN.) The SCAN algorithm is often thought to be inferior to CSCAN since the times at which it visits the inner and outer tracks are less evenly spaced than the times at which it visits the middle tracks.

The shortest-seek-first (SSF) algorithm always moves the head to the request whose track is closest to the track over which the head is currently positioned. If SSF is used then some requests may be treated unfairly. For instance, under heavy workloads the head remains over one portion of the disk. The requests in the other regions are said to *starve*.

A continuum of algorithms, $V(R)$, that have elements of both SSF and SCAN were proposed by Geist and Daniel [26]. Here, the distance to a request is equal to the seek distance if the head can move there while maintaining its current radial direction.

However, if the head must change direction then the distance is the seek distance plus (full radial distance) $\times R$. The head moves to the request that is at the smallest such distance from its current position. It is clear that $V(0)$ =SSF and $V(1)$ =SCAN. Geist and Daniel proposed $V(0.2)$ as an algorithm that performs well.

Note that for each of the above algorithms, the scheduler does not take into account the rotational position of the request, only its track number. Although useful in the past, this design principle currently makes less sense. In older disks the seek time was the dominating factor limiting performance. In modern disks, however, the rotational latency also plays a significant role since seek times are decreasing at a higher rate than rotational latency. Jacobson and Wilkes [33] and Seltzer, Chen and Ousterhout [60] simulated the algorithm shortest-time-first (STF) which always services the request that can be reached in the shortest amount of time (i.e., the time to seek to the correct track plus the time for the request to rotate underneath the disk head). The results of [33] and [60] indicate that for randomly generated requests, STF has better throughput than algorithms that do not take rotational position into account. Although the algorithm STF is prone to starvation, the effects can be lessened if older requests are given higher priority or if the disk head is sometimes forcibly moved to a new region of the disk.

A number of recent papers study disk scheduling from a more "real-world" perspective. Ruemmler and Wilkes [59] and Kotz, Toh and Radhakrishnan [40] developed detailed models of Hewlett-Packard disks. In a separate paper Ruemmler and Wilkes [58] describe disk activity in various UNIX[2] systems. The traces they obtained were later used by Worthington, Ganger and Patt [71] to evaluate many disk scheduling algorithms, including those described above. Methods for obtaining exact disk drive specifications were given by Worthington, Ganger, Patt and Wilkes in [72].

The increasing significance of rotational latency is not the only technological change that has altered the nature of the disk scheduling problem. For instance, preventing starvation is becoming less important since nonvolatile memory (NVRAM, i.e. memory that retains its stored values during a system power loss) is emerging

[2]UNIX is a registered trademark of X/Open Company Limited.

as a viable technology [8, 31]. If the disk buffer (which stores data before it is written to disk) consists of NVRAM then it is not essential for *every* write to get to disk fast. Hence for writes, throughput becomes the only important performance measure. (Servicing read requests is not considered as much of a potential bottleneck, because many reads can be avoided as cache sizes increase.)

Another relevant technological change is that a modern processor, dedicated to the task of disk scheduling, can execute algorithms that are more computationally expensive. For example, the algorithm CHAIN of Section 9.6 has a higher time complexity than STF, but for a given disk configuration the better look-ahead properties of CHAIN mean that it is likely to serve more requests in the next rotation.

9.2 A 3/2-Approximation Algorithm

In this section we present an algorithm that services all of the requests on the disk in at most $\frac{3}{2}T_{\text{opt}} + a$ rotations, where T_{opt} is the number of rotations required by an optimal algorithm that returns the disk head to its starting position. The additive term a depends solely on the reachability function. It is not a function of the number of requests. We do not try to look for an optimal solution since we show in Section 9.3 that the problem is NP-hard.

The Minimum-cost Cycle Cover and a Lower Bound

We first use a minimum-cost cycle cover of the disk graph to derive a lower bound LB for T_{opt}, and then present an algorithm that services all the requests in $\frac{3}{2}LB + a$ rotations. For a graph G, let $\mathcal{C}$ denote a collection of cycles in G. If every node of G is contained in exactly one cycle, then $\mathcal{C}$ is called a *cycle cover* of G. For edge-weighted graphs the cost of a cycle cover, $\mathcal{C}$, is the sum of the weights of the edges in $\mathcal{C}$. A *minimum-cost cycle cover* of G has the minimum cost among all the cycle covers of G. Recall that in the disk graph, the length of an edge R_iR_j is equal to the number of times that the head must cross the line $\theta = 0$ when traveling from R_i to R_j.

The problem of finding a minimum-cost cycle cover is equivalent to solving an

assignment problem derived from the edge weights [22]. In an assignment problem we have a weighted bipartite graph (L, R). The goal is to find a minimum-cost matching in which all vertices in L are matched. Given an n-vertex graph G with vertex set $\{v_0, \ldots, v_{n-1}\}$, we construct a $2n$-node bipartite graph with vertex sets $L = \{\ell_0, \ldots, \ell_{n-1}\}$ and $R = \{r_0, \ldots, r_{n-1}\}$. There is an edge of weight w between ℓ_i and r_j if and only if there is an edge of weight w between v_i and v_j in G. A matching in which all nodes in L (and hence all nodes in R) are matched defines a permutation of the nodes in G. By elementary results in algebra this permutation can be decomposed into disjoint cyclic permutations, each of which corresponds to a cycle in G. Hence the matching in (L, R) gives a cycle cover in G. It is easy to see that the weight of the matching is equal to the weight of the cycle cover. This demonstrates the equivalence of solving the assignment problem and finding a minimum-cost cycle cover. We can solve the assignment problem in $O(n^3)$ time using the Hungarian method of Kuhn [41, 52].

Let $\mathcal{C}$ denote a minimum-cost cycle cover of the disk graph, and let $C^{(i)}$ denote the set of cycles in $\mathcal{C}$ with cost i. Let p be the maximum cost of a cycle in $\mathcal{C}$. Then $\mathcal{C} = C^{(1)} \cup C^{(2)} \cup \ldots \cup C^{(p)}$. Let K be the total cost of $\mathcal{C}$, i.e. $K = \sum_{i=1}^{p} i|C^{(i)}|$, where $|C^{(i)}|$ is the number of cycles in $C^{(i)}$. Note that K is a lower bound on T_{opt}, since an optimal solution to the disk scheduling problem is a cycle cover. Our algorithm finds an order in which to service the cycles in $\mathcal{C}$ such that the disk head can move between the cycles without using "too many" rotations.[3] An approach based on finding a minimum-cost cycle cover was independently proposed by [23], but no performance guarantees were provided for the resulting algorithms.

The Virtual Trace

Before describing the algorithm in detail, we need to define a *virtual trace* to connect neighboring requests on a cycle. A virtual trace does not describe the actual trace

[3]Note that the head can travel between an arbitrary pair of cycles in $t_{\text{fullseek}} + 1$ rotations. This immediately gives us a $t_{\text{fullseek}} + 1$-approximation algorithm. However, by being more careful about the order in which the cycles are serviced, we shall reduce the time taken to travel between cycles and hence reduce the approximation ratio.

of the disk head, but rather an imaginary path defined by the reachability function f. Consider a cycle $c \in C^{(i)}$. Let $R_j = (\theta_j, r_j)$, for $1 \leq j \leq m$, be the requests on c, numbered such that request $R_1 = (\theta_1, r_1)$ satisfies $\theta_1 = \min_{1 \leq j \leq m} \theta_j$, and $R_j R_{j+1}$ and $R_m R_1$ are directed edges in the cycle cover. For each $c \in C^{(i)}$ we shall view the disk as a $2i\pi \times 1$ rectangle, $T^{(i)}$, i.e. i copies of the disk are joined end to end. As demonstrated in Figure 9-1, we represent the requests on cycle c so that every request appears exactly once in rectangle $T^{(i)}$. Formally, R_1 appears at location (ϕ_1, r_1) for $\phi_1 = \theta_1$. Request R_j for $2 \leq j \leq m$ appears at (ϕ_j, r_j), where $\phi_j = \theta_j - \theta_{j-1} + \phi_{j-1} + 2k_j\pi$ and $k_j = d(R_{j-1}, R_j)$.

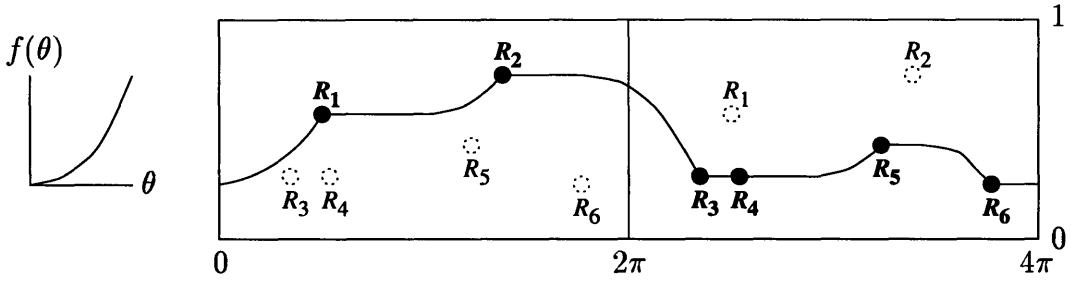

Figure 9-1: (Left) The reachability function f. (Right) A cycle $c \in C^{(2)}$, whose neighboring requests are connected by the virtual trace. The disk is viewed as a $4\pi \times 1$ rectangle $T^{(2)}$.

Consider two neighboring requests R_j and R_{j+1}, which appear at locations (ϕ_j, r_j) and (ϕ_{j+1}, r_{j+1}) respectively. The trace connecting them is composed of a horizontal line (of possibly zero length) followed by a curve defined by f or $-f$. (See Figure 9-1.) Formally, the virtual trace is defined as follows. By the definition of ϕ_j and ϕ_{j+1}, one can verify that there exists $\phi' \in [\phi_j, \phi_{j+1}]$ such that $f(\phi_{j+1} - \phi') = |r_j - r_{j+1}|$. For $r_{j+1} \geq r_j$ let,

$$g_c(\theta) = \begin{cases} r_j & \text{for } \phi_j \leq \theta \leq \phi' \\ r_j + f(\theta - \phi') & \text{for } \phi' < \theta \leq \phi_{j+1} \end{cases}.$$

The virtual trace between R_j and R_{j+1} is defined parametrically by $(\theta, g_c(\theta))$ for $\phi_j \leq \theta \leq \phi_{j+1}$. The case in which $r_{j+1} < r_j$ is analogous. The trace from R_m to R_1 is obtained from the trace connecting (ϕ_m, r_m) and $(2i\pi + \phi_1, r_1)$. If R_1 is the only request on c then the trace is the horizontal line $r = r_1$. The next four lemmas describe some properties of the virtual trace of cycle $c \in C^{(i)}$.

Lemma 9.2.1 *If the virtual trace can connect two requests within an angle θ then the disk head can service both of them within a rotation through angle θ.*

Proof: The result follows from the definitions of the virtual trace and the reachability function f. □

Lemma 9.2.2 *If R_j and R_k are two requests on cycle $c \in C^{(i)}$ and they appear at (ϕ_j, r_j) and (ϕ_k, r_k) respectively, then $|r_j - r_k| \leq f(i\pi)$.*

Proof: Without loss of generality, we assume $j \leq k$. Either $\phi_k - \phi_j \leq i\pi$ or $(2i\pi + \phi_j) - \phi_k \leq i\pi$. If the former case holds then,

$$\begin{aligned} |r_k - r_j| &\leq \sum_{\ell=j+1}^{k} f(\theta_\ell - \theta_{\ell-1}) \\ &\leq f(\sum_{\ell=j+1}^{k} (\theta_\ell - \theta_{\ell-1})) \leq f(i\pi). \end{aligned}$$

The first inequality follows the definition of the reachability function. The second and third inequalities follow from properties 3 and 1 of f respectively. A similar argument applies for the case in which $(2i\pi + \phi_j) - \phi_k \leq i\pi$. □

Lemma 9.2.3 *For a cycle $c \in C^{(i)}$, the slope of the virtual trace $(\theta, g_c(\theta))$ is between $-f'(i\pi)$ and $f'(i\pi)$ for $0 \leq \theta \leq 2i\pi$.*

Proof: By construction, the virtual trace g_c for cycle c is composed of the curves defined by f and $-f$. In particular, $g_c(\theta) = r_k \pm f(\theta - \phi')$ for $\phi_k \leq \theta \leq \phi_{k+1}$ and $\phi' \in [\phi_k, \phi_{k+1}]$. (Recall that $f(\theta) = 0$ for $\theta < 0$.) Lemma 9.2.2 implies that $\phi_{k+1} - \phi' \leq i\pi$. Property 2 of f therefore implies the result. □

Recall that $f(i\pi)$ is the radial distance that the head can travel after $i/2$ rotations of the disk, given that the head starts and ends at rest. Let $q_i = \lceil 1/f(i\pi) \rceil$. For each $C^{(i)}$ the rectangle $T^{(i)}$ is divided into smaller rectangles $T_1^{(i)}, T_2^{(i)}, \ldots, T_{q_i}^{(i)}$, each of size at most $2i\pi \times f(i\pi)$. A request $R = (\theta, r)$ is *in* rectangle $T_j^{(i)}$ if and only if $(j-1)f(i\pi) \leq r < j \cdot f(i\pi)$. We have,

Lemma 9.2.4 *If $(0, g_c(0))$ is in rectangle $T_j^{(i)}$ then the trace of cycle c either stays in rectangles $T_j^{(i)}$ and $T_{j+1}^{(i)}$ or else it stays in rectangles $T_j^{(i)}$ and $T_{j-1}^{(i)}$.*

Proof: If (θ_k, r_k) is some request on $c \in C^{(i)}$ then by Lemma 9.2.2 the trace of c stays in the horizontal stripe defined by $r_k - f(i\pi) \leq r \leq r_k + f(i\pi)$. Since rectangles $T_j^{(i)}$ are of size $2i\pi \times f(i\pi)$, the result follows. □

For a cycle $c \in C^{(i)}$ let the *centerpoint* of cycle c, denoted by centerpoint(c), be the point $(i\pi, g_c(i\pi))$ on the virtual trace, and let the *leftpoint* of cycle c, denoted by leftpoint(c), be the point $(0, g_c(0)) = (2i\pi, g_c(2i\pi))$ on the trace. For an angle $\rho \in [0, 2i\pi)$, let α_c be the first request on the virtual trace of c that appears *after* the line $\theta = \rho$. We use the phrase *cycle c is serviced starting at angle ρ* to mean that α_c is the first request on c to be serviced and the other requests on c are serviced in the order of the cycle.

The Algorithm

The algorithm HEADSCHEDULE is shown in Figure 9-2. It proceeds by finding a min-cost cycle cover of the disk graph and then determining a particular order in which to service the cycles. For each cycle, HEADSCHEDULE identifies the first request to visit and then travels around the cycle servicing all of the requests. Our goal is to show that the disk head can connect the last request of a cycle to the first request of the next cycle without using "too many" rotations.

A cycle $c \in C^{(i)}$ is *long* if $i \geq 2L$ where $L = \lceil t_{\text{fullseek}} \rceil + 1$; otherwise c is *short*. Note that in L rotations, the head can travel from any request to any other request. The long cycles can therefore be serviced in any order and the number of rotations required is at most

$$\sum_{i=2L}^{p} i|C^{(i)}| + \sum_{i=2L}^{p} L|C^{(i)}| \leq \frac{3}{2} \sum_{i=2L}^{p} i|C^{(i)}|. \tag{9.1}$$

Hence, the approximation ratio of 3/2 is achieved for servicing the long cycles.

We therefore focus on the order in which HEADSCHEDULE services the short cycles. The algorithm first services the cycles in $C^{(1)}$ and then the cycles in $C^{(2)}$, etc.

HEADSCHEDULE

1 find a min-cost cycle cover $\mathcal{C} = C^{(1)} \cup \ldots \cup C^{(p)}$

service short cycles

2 **for** $i = 1$ to $2\lceil t_{\text{fullseek}} \rceil + 1$ **do**

3 **if** i is odd **then**

4 **for** $j = 1$ to q_i **do**

5 CYCLECONNECT $(H_j^{(i)}, V_j^{(i)})$

6 **if** i is even **then**

7 **for** $j = q_i$ down to 1 **do**

8 CYCLECONNECT $(H_j^{(i)}, V_j^{(i)})$

service long cycles

9 **while** there exist unserviced long cycles c

service cycle c starting at angle 0

CYCLECONNECT $(H_j^{(i)}, V_j^{(i)})$

1 **while** there exist unserviced cycles in $H_j^{(i)}$, alternate between the following.

2 • Let c be the unserviced cycle in $H_j^{(i)}$ which has the highest centerpoint. Service c starting at angle 0.

• Let c be the unserviced cycle in $H_j^{(i)}$ which has the lowest leftpoint. Service c starting at angle $i\pi$.

3 **while** there exist unserviced cycles in $V_j^{(i)}$, alternate between the following.

4 • Let c be the unserviced cycle in $V_j^{(i)}$ which has the lowest centerpoint. Service c starting at angle 0.

• Let c be the unserviced cycle in $V_j^{(i)}$ which has the highest leftpoint. Service c starting at angle $i\pi$.

Figure 9-2: The HEADSCHEDULE algorithm.

By Lemma 9.2.4, cycles in $C^{(i)}$ can be divided into the following groups.

1. The *hill group* $H_j^{(i)}$ consists of cycles c whose leftpoint $(0, g_c(0))$ is in rectangle $T_j^{(i)}$ and whose centerpoint $(i\pi, g_c(i\pi))$ is in rectangle $T_j^{(i)}$ or $T_{j+1}^{(i)}$.

2. The *valley group* $V_j^{(i)}$ consists of cycles c whose leftpoint is in rectangle $T_j^{(i)}$ and whose centerpoint is in rectangle $T_{j-1}^{(i)}$.

To service the cycles in $C^{(i)}$, HEADSCHEDULE services the requests in rectangle $T^{(i)}$ from bottom to top when i is odd and services requests in $T^{(i)}$ from top to bottom when i is even. To be more precise, for odd i HEADSCHEDULE first services the cycles in $H_1^{(i)}$ and then the cycles in $V_1^{(i)}$, $H_2^{(i)}$ and $V_2^{(i)}$, etc; for even i HEADSCHEDULE first services cycles in $H_{q_i}^{(i)}$ and then cycles in $V_{q_i}^{(i)}$, $H_{q_{i-1}}^{(i-1)}$ and $V_{q_{i-1}}^{(i-1)}$, etc. The subroutine CYCLECONNECT specifies the order in which HEADSCHEDULE services the cycles in $H_j^{(i)}$ and $V_j^{(i)}$ and also identifies the first request to be serviced on each cycle. (See Figures 9-2 and 9-3.) Hence, the order in which HEADSCHEDULE services all the requests on the disk is fully determined.

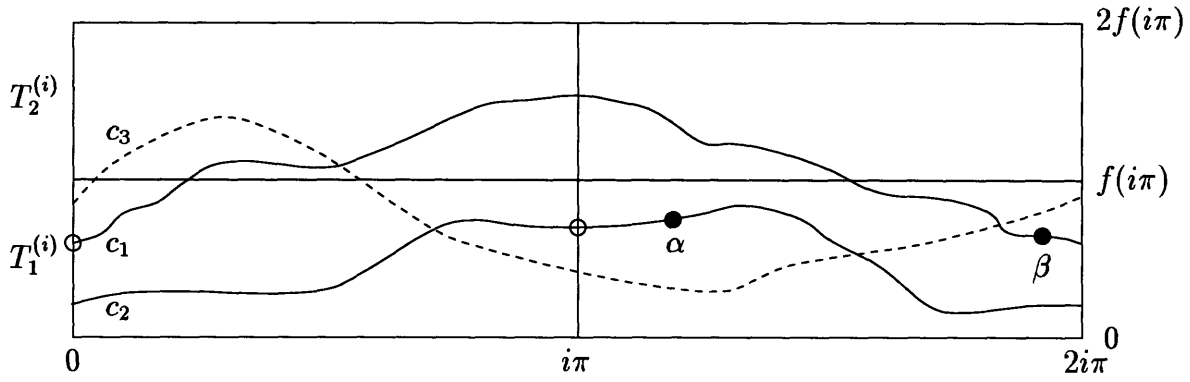

Figure 9-3: CYCLECONNECT services the cycles in $H_1^{(i)}$ in the order c_1, c_2, c_3, since c_1 has the highest centerpoint and c_2 has the lowest leftpoint. To illustrate Lemma 9.2.6, β is serviced last on cycle c_1 and α is serviced first on c_2. Request α is reachable from leftpoint(c_1). CYCLECONNECT therefore uses $i/2$ rotations to travel from c_1 to c_2.

To complete the analysis, we shall show that the approximation ratio of 3/2 is achieved for servicing short cycles. In particular, we show in Lemma 9.2.6 that HEADSCHEDULE uses $\frac{3}{2}i|H_j^{(i)}|$ rotations (resp. $\frac{3}{2}i|V_j^{(i)}|$ rotations) to service all the cycles in $H_j^{(i)}$ (resp. $V_j^{(i)}$). Then in the proof of Lemma 9.2.7 we show that the head can travel between $H_j^{(i)}$ and $V_j^{(i)}$ etc. in a small number of rotations.

Let γ be a point of the form $(0, r_\gamma)$ and consider the reachability cone rooted at γ. Let functions $h_1(\theta)$ and $h_2(\theta)$ define the upper and lower boundaries of the cone, i.e. $h_1(\theta) = r_\gamma + f(\theta)$ and $h_2(\theta) = r_\gamma - f(\theta)$. For a cycle $c \in C^{(i)}$ let $\alpha_c = (\phi^{(c)}, r^{(c)})$ be the first request on c that appears after the line $\theta = i\pi$. For simplicity we assume that $\phi^{(c)} > i\pi$. (For the case in which $\phi^{(c)} \leq i\pi$, the location of α_c can be taken to be at $(2i\pi + \phi^{(c)}, r^{(c)})$ for the analysis.) Figure 9-4 illustrates Lemma 9.2.5 and its proof.

Lemma 9.2.5 *If the centerpoint of cycle c is in the reachability cone rooted at γ, then α_c is in the reachability cone rooted at γ. That is, if $h_1(i\pi) \leq g_c(i\pi) \leq h_2(i\pi)$, then $h_1(\phi^{(c)}) \leq g_c(\phi^{(c)}) \leq h_2(\phi^{(c)})$.*

Proof: The definition of h_1 and h_2 implies that $h_1'(\theta) = f'(\theta)$ and $h_2'(\theta) = -f'(\theta)$. By Lemma 9.2.3, the virtual trace of cycle c never has a slope whose absolute value is greater than $f'(i\pi)$. Since f' is nondecreasing by Property 2 of f, we have $h_2'(\theta) \leq g_c'(\theta) \leq h_1'(\theta)$ for $\theta \geq i\pi$. Therefore, if $h_1(i\pi) \leq g_c(i\pi) \leq h_2(i\pi)$ then $h_1(\phi^{(c)}) \leq g_c(\phi^{(c)}) \leq h_2(\phi^{(c)})$ for $\phi^{(c)} \geq i\pi$. Stated differently, if the centerpoint of the virtual trace is in the reachability cone, then the trace can never leave the cone after the centerpoint, i.e. the point $(\theta, g_c(\theta))$ is in the cone for any $\theta \geq i\pi$. □

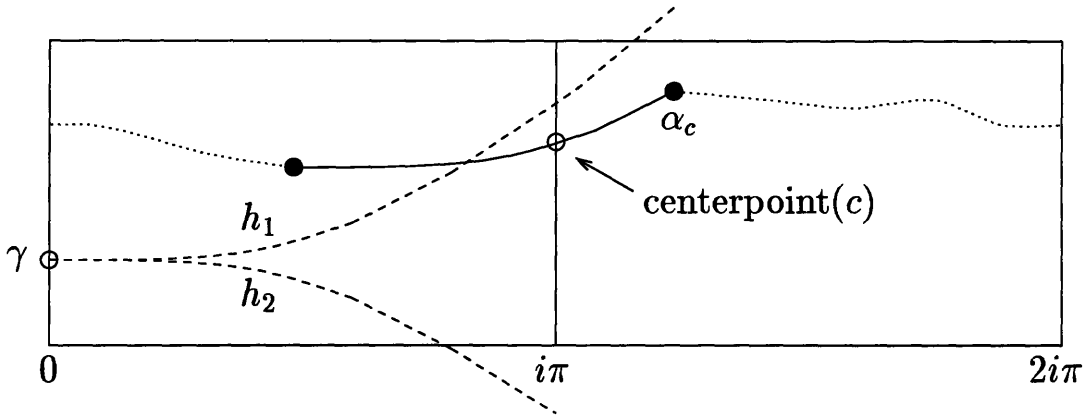

Figure 9-4: The two dashed curves represent functions h_1 and h_2, which define the upper and lower boundaries of the reachability cone rooted at the point γ. If centerpoint(c) is in the reachability cone then α_c, the first request on c after centerpoint(c), is also in the cone.

Lemma 9.2.6 *Subroutine* CYCLECONNECT *services all the cycles in* $H_j^{(i)}$ *(resp.* $V_j^{(i)}$*) in* $\frac{3}{2}i|H_j^{(i)}|$ *rotations (resp.* $\frac{3}{2}i|V_j^{(i)}|$ *rotations).*

Proof: Stated intuitively, we show that CYCLECONNECT uses $i/2$ rotations to travel to the next cycle and then uses i rotations to service all the requests on this cycle. Let $c_h \in H_j^{(i)}$ and $c_\ell \in H_j^{(i)}$ be the unserviced cycles that have the highest centerpoint and lowest leftpoint, respectively. Suppose that CYCLECONNECT services c_h followed by c_ℓ. Let β_h be the last request on cycle c_h before leftpoint(c_h) and let α_ℓ be the first request on cycle c_l after centerpoint(c_ℓ). (Note that β_h is serviced last on cycle c_h and α_ℓ is serviced first on c_ℓ. See Figure 9-3.) By the definition of the virtual trace it is clear that leftpoint(c_h) is *reachable* from β_h, i.e. leftpoint(c_h) is in the reachability cone rooted at β_h. The following two arguments show that centerpoint(c_ℓ) is in the reachability cone rooted at leftpoint(c_h).

Case 1: *Centerpoint(c_ℓ) is in* $T_j^{(i)}$. Since the height of $T_j^{(i)}$ is $f(i\pi)$, centerpoint(c_ℓ) is reachable from leftpoint(c_h).

Case 2: *Centerpoint(c_ℓ) is in* $T_{j+1}^{(i)}$. Centerpoint(c_h) is higher than centerpoint(c_ℓ) by the definition of c_h. Since centerpoint(c_h) is reachable from leftpoint(c_h), centerpoint(c_ℓ) is reachable from leftpoint(c_h).

Lemma 9.2.5 implies that α_ℓ is in this reachability cone and is hence reachable from β_h. Therefore the head services the requests in c_h, moves to cycle c_ℓ and services the requests in c_ℓ in $i+i/2+i = 5i/2$ rotations. An analogous argument can be applied to show that after servicing any cycle in $H_j^{(i)}$ (resp. $V_j^{(i)}$) the next cycle can be serviced in $3i/2$ rotations. For example, after servicing c_ℓ, the head can travel to the unserviced cycle with the highest centerpoint in $i/2$ rotations. The result follows. □

Lemma 9.2.7 HEADSCHEDULE *services all the short cycles in,*

$$\sum_{i=1}^{2L-1} \frac{3}{2} i|C^{(i)}| + \sum_{i=1}^{2L-1} \frac{3}{2} i q_i$$

rotations, where $L = \lceil t_{\text{fullseek}} \rceil + 1$.

Proof: Let i be an odd integer that satisfies $1 \leq i \leq 2L-1$. (The case when i

is even is similar.) HEADSCHEDULE can finish serving the requests in $C^{(i)} - 1$ with the head positioned at a point in rectangle $T_1^{(i-1)}$ with angular coordinate 0. Since rectangle $T_1^{(i)}$ contains $T_1^{(i-1)}$ the head can move to its first request in $C^{(i)}$ in $i/2$ rotations.

Now consider a call to the subroutine CYCLECONNECT $(H_j^{(i)}, V_j^{(i)})$. By Lemma 9.2.6, in $\frac{3}{2}i|H_j^{(i)}|$ rotations the head can service all the requests in $H_j^{(i)}$ and return to a point in $T_j^{(i)}$ with angular coordinate 0. The beginning of the first cycle in $V_j^{(i)}$ is in $T_j^{(i)}$ and so the head can move there in $i/2$ rotations. Using Lemma 9.2.6 again, we have that in $\frac{3}{2}i|V_j^{(i)}|$ rotations the head can service all the requests in $V_j^{(i)}$ and return to a point in $T_j^{(i)}$ with angular coordinate 0. The beginning of the first cycle in $H_{j+1}^{(i)}$ is in $T_{j+1}^{(i)}$ and hence the head can move there in i rotations. Summing over all j and i we obtain the result. □

Since $q_i = \lceil 1/f(i\pi) \rceil$ and $f(i\pi) \geq if(\pi)$ by Property 3 of f, we have $iq_i \leq i + q_1$. Hence, $\sum_{i=1}^{2L-1} \frac{3}{2} iq_i \leq 3Lq_1 + 3L^2$. Combined with the analysis for long cycles (see inequality 9.1) we have,

Theorem 9.2.8 *The algorithm* HEADSCHEDULE *has a* $3/2$*-approximation ratio with an additive term of at most* $3Lq_1 + 3L^2$*, where* $L = \lceil t_{\text{fullseek}} \rceil + 1$.

9.3 NP-Hardness of Disk Scheduling

In this section we show that given a reachability function and a set of requests on the disk it is NP-hard to determine the optimal schedule. The reduction is from the following restricted version of the Directed Hamiltonian Cycle problem.

- **Fact 1** The Directed Hamiltonian Cycle problem is NP-complete even if each vertex in the graph is adjacent to at most 3 arcs [56, 24].

Outline of the reduction

Given such a graph G with n nodes, we first place requests on a disk with dimensions $poly(n) \times poly(n)$. Later on we rescale the coordinates to obtain a disk with dimensions

$2\pi \times 1$. We also construct a reachability function such that *all requests can be serviced in n rotations and the disk head can return to its starting point if and only if G contains a Hamiltonian cycle.*

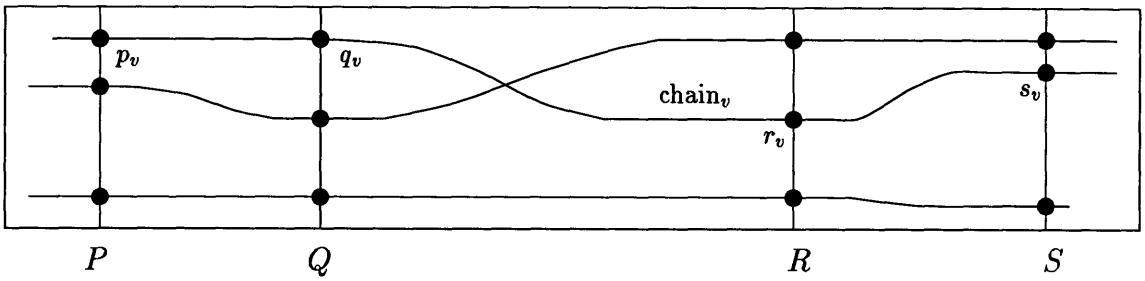

Figure 9-5: The chains of requests.

We shall assume that the disk head must start at a point with angular coordinate $\theta = 0$. There will be four columns of requests that we place on the disk,

$$P = \{p_v : v \in V_G\} \quad Q = \{q_v : v \in V_G\}$$
$$R = \{r_v : v \in V_G\} \quad S = \{s_v : v \in V_G\},$$

where V_G is the vertex set of G. (See Figure 9-5.) These columns are placed so that requests in Q have a higher angular coordinate than requests in P, requests in R have a higher angular coordinate than requests in Q, and requests in S have a higher angular coordinate than requests in R. In addition the line $\theta = 0$ lies between column S and column P. The exact positions of these columns will be determined later. We also place a set of n chains of requests on the disk and denote them by $\{\text{chain}_v : v \in V_G\}$. The construction has the following properties.

1. For all v, chain_v contains the requests p_v, q_v, r_v and s_v.

2. The requests all have integer coordinates.

3. The head can travel from the end of chain_u to the beginning of chain_v crossing $\theta = 0$ exactly once if and only if (u, v) is a directed edge in G.

4. If all the requests are serviced in n rotations then on each rotation chain_v is serviced for some v.

5. The request s_v is above s_u if and only if r_v is above r_u.

6. The request q_v is above q_u if and only if p_v is above p_u.

7. For all v, chain_v can be serviced in one rotation.

Theorem 9.3.1 *Suppose that the above properties are satisfied. Then all of the requests can be satisfied in n rotations and the head can return to its starting point if and only if G has a Hamiltonian cycle.*

Proof: Suppose that G has a Hamiltonian cycle. Let the cycle be $v_0, v_1, \ldots, v_{n-1}, v_0$. Then by Properties 3 and 7 there is a valid solution with n rotations that has the form,

$$\text{chain}_{v_0}, \text{chain}_{v_1}, \ldots, \text{chain}_{v_{n-1}}.$$

Conversely, suppose that we can service the requests in n rotations. By Property 4 the schedule must have the following form.

$$\text{chain}_{u_0}, \text{chain}_{u_1}, \ldots, \text{chain}_{u_{n-1}}.$$

The head must cross $\theta = 0$ only once when traveling from the end of $\text{chain}_{u_{i-1}}$ to the beginning of chain_{u_i}. Note also that since the disk head returns to its starting point, it must be able to travel from the end of $\text{chain}_{u_{n-1}}$ to the beginning of chain_{u_0} crossing $\theta = 0$ only once. Therefore by Property 3, $u_0, u_1, u_2, \ldots u_{n-1}, u_0$ must be a Hamiltonian cycle. □

The Construction

We now define the reachability function that we shall use. It is the simple function,

$$f(\theta) = \theta^2.$$

Enforcing Property 3

We first focus on the region between column S and column P. Suppose that we can arbitrarily specify distances between requests, i.e. suppose that the distances *are not* given by a reachability function. Then the following construction would immediately guarantee Property 3. We define,

$$d(s_u, p_v) = \begin{cases} 1 & \text{if } (u,v) \text{ is an edge in } G \\ 2 & \text{otherwise} \end{cases} \tag{9.2}$$

(Recall that the distance from s_u to p_v is the number of times that the head must cross the line $\theta = 0$ when traveling from s_u to p_v. Recall also that the line $\theta = 0$ lies between s_u and p_v.) However we can only specify distances using a reachability function. Our goal, therefore, is to construct requests with similar distance relationships using the reachability function $f(\theta) = \theta^2$. We make a new graph G', consisting of $2n$ nodes, which lets us define the requests. We transform each vertex $u \in G$ into two vertices $u_{\text{in}}, u_{\text{out}} \in G'$, where u_{in} has only incoming arcs and u_{out} has only outgoing arcs. Let V_{in} be the set of nodes with incoming arcs only and let V_{out} be the set of nodes with outgoing arcs only. The edges in G' are defined by,

$$(u_{\text{out}}, v_{\text{in}}) \in G' \Leftrightarrow (u,v) \in G.$$

We examine the structure of G'. Notice that without loss of generality all nodes in the underlying undirected graph of G' have degree 1 or 2. (If a node has degree 0 or 3 then G has no Hamiltonian cycle.) An undirected graph in which all nodes have degree 1 or 2 is a collection of paths and cycles. Therefore all connected components in G' have one of two structures. (See Figure 9-6.)

1. **A sawtooth.** All nodes have degree 2 except for exactly two nodes that have degree 1. The nodes alternate between being in V_{in} and being in V_{out}.

2. **A circular sawtooth.** All nodes have degree 2. They alternate between V_{in} and V_{out}.

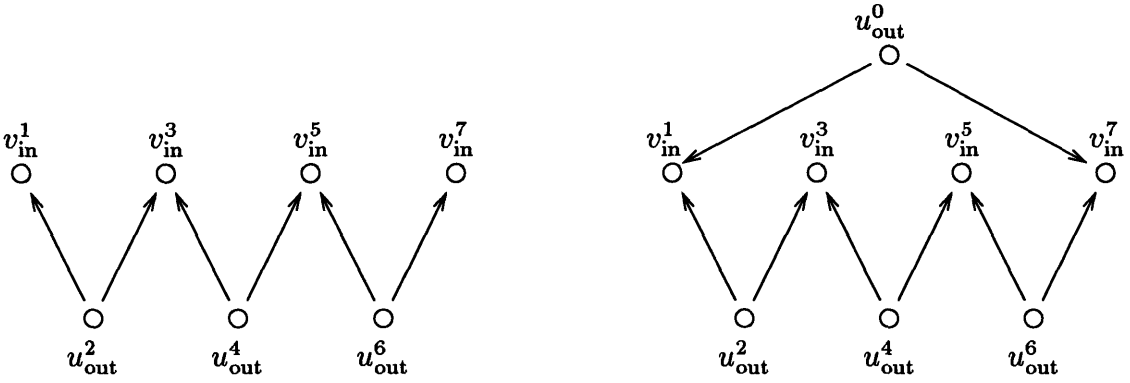

Figure 9-6: (Left) A sawtooth. (Right) A circular sawtooth.

For each node in G' we define a request on the disk. More specifically, for each node $u_{\text{out}} \in V_{\text{out}}$, we define a request s'_u and for each node $v_{\text{in}} \in V_{\text{in}}$ we define a request p'_v. These requests will satisfy,

Requirement 9.3.2 *The head can travel from s'_u to p'_v crossing $\theta = 0$ exactly once if and only if $(u_{\text{out}}, v_{\text{in}})$ is an edge in G'.*

We now show how to place requests that correspond to a sawtooth. Consider a sawtooth in which both endnodes are in V_{in}. (The other three cases are analogous.) Let the nodes in the sawtooth be,

$$v_{\text{in}}^1, u_{\text{out}}^2, v_{\text{in}}^3, \ldots, v_{\text{in}}^k.$$

(See Figure 9-6.) We satisfy requirement 9.3.2 if,

- $s'_{u^{2i}}$ is located at the point $(0, 2i)$.
- $p'_{v^{2i-1}}$ is located at the point $(1, 2i-1)$.

(Recall the definition of the reachability function.)

The case of a circular sawtooth is more difficult. We modify the above construction to deal with this case. Suppose that the nodes in the circular sawtooth are,

$$u_{\text{out}}^0, v_{\text{in}}^1, \ldots, u_{\text{out}}^{k-1}, v_{\text{in}}^k, u_{\text{out}}^0.$$

(See Figure 9-6.) The requests $p'_{v^1}, s'_{u^2}, \ldots, p'_{v^{k-2}}, s'_{u^{k-1}}$ are placed as above. The request p'_{v^k} is moved to $(2, k+3)$. We also place a request s'_{u^0} at $((1-k)/2, -(1-k)^2/4 + 1 - k)$. (Note that k is odd and hence these coordinates are integral.) We would like to have the following distances.

$$\begin{aligned} d(s'_{u^0}, p'_{v^1}) &= 0, \\ d(s'_{u^0}, p'_{v^k}) &= 0, \\ d(s'_{u^0}, p'_{v^i}) &= 1, \text{ if } i \neq 1, k, \\ d(s'_{u^{k-1}}, p'_{v^k}) &= 0. \end{aligned}$$

These equations do indeed hold since,

$$\begin{aligned} \left(2 - \frac{(1-k)}{2}\right)^2 - \frac{(1-k)^2}{4} + 1 - k &= 4 - 2(1-k) + \frac{(1-k)^2}{4} - \frac{(1-k)^2}{4} + 1 - k \\ &= k + 3, \\ \left(1 - \frac{(1-k)}{2}\right)^2 - \frac{(1-k)^2}{4} + 1 - k &= 1 - (1-k) + (1-k) \\ &= 1, \\ (2-0)^2 &= (k+3) - (k-1). \end{aligned}$$

It can now be seen that requirement 9.3.2 is satisfied. (All the other required distances follow immediately from the construction.) See Figure 9-7.

We have shown how to construct requests for each connected component of G' separately. We now place these blocks of requests for each connected component one above the other. We do this in such a way that the difference between the radial coordinates of requests corresponding to different components is at least αn^2 for some sufficiently large constant α. This will ensure that if the head travels between two requests corresponding to two different components then it crosses $\theta = 0$ at least twice. For each request s'_u we place the request s_u so that it has the same radial coordinate as s'_u and has angular coordinate $(1-k)/2$. The request s'_u is connected

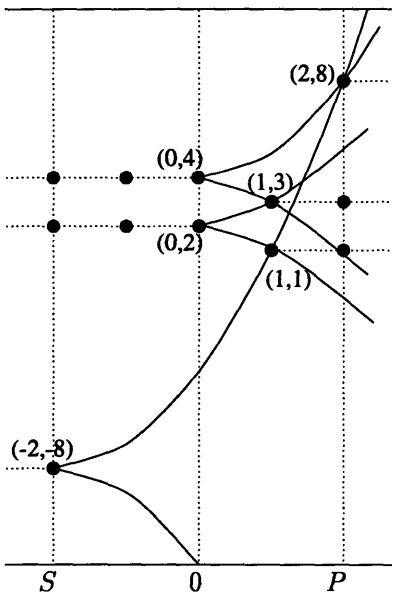

Figure 9-7: The placing of requests for a circular sawtooth with $k = 5$. Here $s'_{u^2} = (0,2)$, $s_{u^2} = (-2,2)$, $p'_{v^1} = (1,1)$, $p_{v^1} = (2,1)$ etc.

to the request s_u by a subchain of requests spaced one unit apart in the angular direction and with the same radial coordinate as s_u and s'_u. Similarly for each request p'_v we place the request p_v so that it has the same radial coordinate as p'_v and has angular coordinate 2. (See Figure 9-7.) Note that we have now defined the exact positions of the columns S and P. Property 3 is now satisfied. The total number of requests used is $O(n^2)$ and the dimensions of the area of disk used are $O(n) \times O(n^3)$. (The angular dimension is $O(n)$ and the radial dimension is $O(n^3)$.)

Enforcing Properties 4, 5 and 6

We next describe the requests in columns Q and R and the additional requests that must be placed between these columns. By Properties 5 and 6 the radial order of the requests in Q and R is determined by the radial order of the requests in P and S. By renumbering the vertices of G we can set the radial coordinate of q_{v_i} to be $5i$. We can then set the radial coordinate of r_{v_i} to be $5h(i)$, where h is a permutation on $0, 1, 2, \ldots, n-1$ defined by the radial order of the requests in S. Our goal in this part of the construction is to place a subchain of requests between q_v and r_v so that in any solution that takes n rotations, this subchain must be serviced in less than one

rotation. This is done for all vertices v in G. Since h is an arbitrary permutation, these chains must cross. However, at the crossing points we shall place the requests so that in an n-rotation schedule, the disk head cannot jump from one subchain to an overlapping subchain.

We first consider the simple case in which h is the transposition $(0, 1, 2, \ldots, n-1) \rightarrow (1, 0, 2, \ldots, n-1)$.

Lemma 9.3.3 *If h is the above transposition, we can place $O(1)$ requests between q_v and r_v for all v so that in an n-rotation schedule each subchain is serviced in less than 1 rotation. The total number of requests used is $O(n)$ and the area of disk used has dimensions $O(1) \times O(n)$.*

Proof: Consider the following sets of requests on the disk.

$$A_0 = \{(0,0),(1,0),(2,0),(3,0),(5,4),(6,5),(7,5)\},$$
$$A_1 = \{(0,5),(1,4),(2,3),(3,2),(4,2),(5,2),(6,1),(7,0)\}.$$

(See Figure 9-8.) Now suppose that $(3,0)$ and $(5,4)$ are serviced in different rotations.

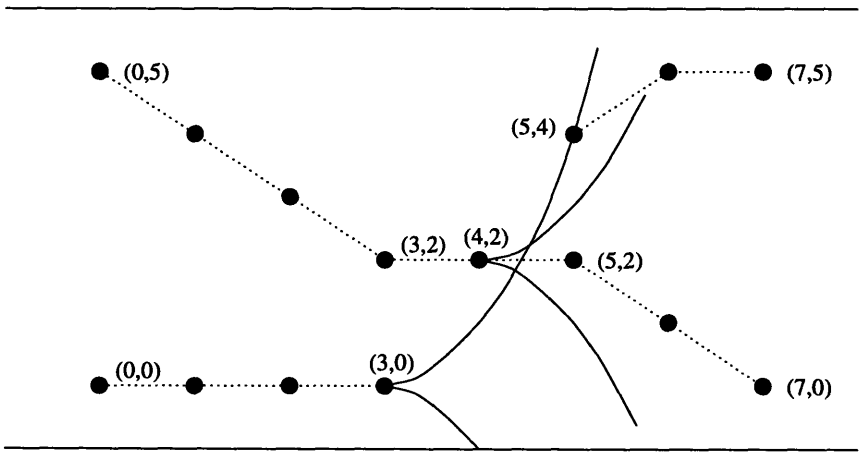

Figure 9-8: Placing requests to enforce a permutation.

Then by the definition of the reachability function, $(4,2)$ must be serviced in a third rotation. Hence if all the requests in A_0 and A_1 are serviced in two rotations then $(3,0)$ and $(5,4)$ must be serviced in one of them and $(4,2)$ must be serviced in the

other. By carrying out similar (but simpler) arguments on other pairs of requests we must have that if all the requests in A_0 and A_1 are serviced in two rotations then all the requests in A_0 must be serviced in one of them and all the requests in A_1 must be serviced in the other.

Now for vertex v_j in G, $j \neq 0, 1$, we place the requests,

$$A_j = \{(0, 5j), (1, 5j), (2, 5j), (3, 5j), (4, 5j), (5, 5j), (6, 5j), (7, 5j)\}$$

on the disk. Suppose that all the requests in $\bigcup_j A_j$ are serviced in n rotations. It is clear that all the requests in A_j must be serviced in a single rotation for $j \neq 0, 1$ and the requests in $A_0 \cup A_1$ must be serviced in 2 rotations. Therefore by the above argument for A_0 and A_1, the requests in A_j must be serviced in a single rotation for all j, $0 \leq j < n$. □

Corollary 9.3.4 *If h is an arbitrary permutation, we can place $O(n^2)$ requests between q_v and r_v for all v so that in an n-rotation schedule each subchain is serviced in less than 1 rotation. The total number of requests used is $O(n^3)$ and the area of disk used has dimensions $O(n^2) \times O(n)$.*

Proof: By an elementary result in algebra an arbitrary permutation is the composition of at most n^2 transpositions of neighboring elements. Hence we can construct a set of requests corresponding to an arbitrary permutation by simply concatenating n^2 structures similar to the one described above. The first column of requests of one structure will be identified with the last column of requests of the previous structure. All of these requests can clearly be placed in a region with dimensions $O(n^2) \times O(n)$ and the number of requests used is $O(n^3)$. The entire region can be shifted in the angular direction so that it lies between columns Q and R, whose exact positions can be determined using the comments below. □

It only remains to add subchains of requests between P and Q and between R and S to complete the enforcement of Properties 4, 5 and 6. It is easy to see that this can be done and so we omit the details since they are a little awkward to describe. Once

these subchains have been constructed it is possible to calculate the exact positions for the columns P, Q, R and S so that all the subchains "match up" to form the complete chains.

We have described the reduction in terms of a disk with dimensions $poly(n) \times poly(n)$. In order to obtain a reduction for a disk with dimensions $2\pi \times 1$ we simply scale all the coordinates of the requests by an appropriate amount. This has the effect of scaling the reachability function. Note that there are $poly(n)$ requests and each request has integral coordinates before the scaling. This, together with the fact that the disk before the scaling has polynomial dimensions, implies that each request can be described using a number of bits that is polynomial in n. Hence the entire input can be described using a number of bits that is polynomial in n. The reduction is complete.

9.4 An Optimal Algorithm for Linear Reachability Functions

Although the disk scheduling problem is NP-hard for general reachability functions, optimal solutions can be obtained for a special case. In this special case, the head either has no radial movement or else has full radial speed s. The reachability function is therefore linear, i.e. $f(\theta) = s\theta$. In addition, we require that the disk head starts at the point $(0,0)$ and ends at $(0,1)$.[4]

One can verify that the linearity of f ensures the *reachability property.* That is, regardless of its current speed, the head can follow any head path passing through its current position. To be more precise, suppose that on one rotation the disk head

[4]In Section 9.3 we showed that the disk scheduling problem is NP-hard when the disk head must start and end at the same place. It is however easy to show that the problem remains NP-hard when the head must start at $(0,0)$ and end at $(0,1)$. This is because the Directed Hamiltonian Path problem is NP-hard when the beginning vertex and the end vertex are specified. We construct an instance of the disk scheduling problem that is almost identical to the one used in Section 9.3. The only difference is that there is a chain of requests from $(0,0)$ to s_u where u is the first request required to be on a Hamiltonian path, and a chain of requests from p_v to $(0,1)$ where v is the last request that is required to be on the path. (Note that these chains might need to cross other chains but this can be done using the techniques of Section 9.3.)

goes from point A to point B to point C, and on another rotation the disk head goes from point D to point B to point E. Then the head can go from A to B to E or from D to B to C. (Note that A, B, etc. are any points on the head path, not necessarily requests.) The reachability property does not hold for general reachability functions.

Suppose that P is a head path that services all the requests. If P requires m rotations let $(\phi, g_P(\phi))$, $0 \leq \phi \leq 2m\pi$, be the location of the head after it rotates through an angle ϕ. Path P satisfies the *monotone property* if $g_P(\phi) \leq g_P(\phi + 2k\pi)$ for any ϕ and positive integer k, where $0 \leq \phi \leq \phi + 2k\pi \leq 2m\pi$.

Lemma 9.4.1 *Suppose that the disk head must start at* $(0,0)$ *and end at* $(0,1)$. *Then there exists an optimal solution to the disk scheduling problem such that the monotone property is satisfied.*

Proof: Consider any optimal solution and its corresponding path P. Suppose that P requires m rotations. We construct a new path Q with m rotations such that Q preserves monotonicity and can be followed by the disk head. In particular, for any angle $\theta \in [0, 2\pi)$ and integer $i \in [0, m-1]$, let $g_Q(\theta + 2i\pi)$ be the ith smallest value from $g_P(\theta)$, $g_P(\theta+2\pi)$, ..., $g_P(\theta+2(m-1)\pi)$. (See Figure 9-9.) By construction, the path Q that corresponds to g_Q takes m rotations and preserves monotonicity. The reachability property implies that Q is realizable by the disk head. □

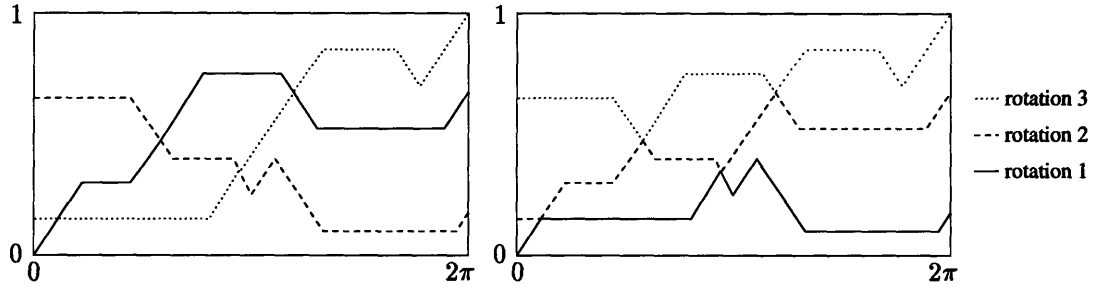

Figure 9-9: (Left) An optimal path P with $m = 3$ rotations. (Right) Path Q with 3 rotations which preserves monotonicity and is realizable by the disk head.

We now describe a situation in which monotonicity is violated. Consider the point

(ϕ_0, r_0). The region *under* (ϕ_0, r_0) consists of points (ϕ, r), where,

$$\begin{cases} 0 \leq r < r_0 - f(\phi - \phi_0) & \text{for} \quad \phi > \phi_0 \\ 0 \leq r < r_0 - f(\phi_0 - \phi) & \text{for} \quad \phi \leq \phi_0. \end{cases}$$

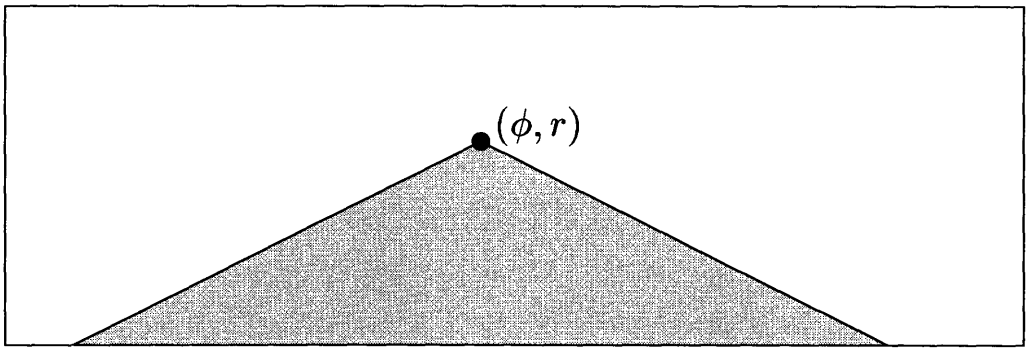

Figure 9-10: The shaded triangle is the region under the point (ϕ, r).

(See Figure 9-10.) If a request $R = (\theta, r)$, $0 \leq \theta < 2\pi$, is serviced on the $(k+1)$st rotation, then we say that R is serviced at angle $2k\pi + \theta$. We have the following.

Lemma 9.4.2 *Suppose request $R = (\theta, r)$ is serviced at angle $2k\pi + \theta$. If, when R is serviced, there are unserviced requests in the region under $(2k\pi + \theta, r)$, then monotonicity is violated.*

Proof: Let g be the function that describes the head path. Suppose that in the region under $(2k\pi + \theta, r)$ there is an unserviced request $U = (\theta + \theta_u, r_u)$, $-\pi < \theta_u \leq \pi$. Then $g(2k\pi + \theta) = r$ and $g(2\ell\pi + \theta + \theta_u) = r_u$ for some $\ell > k$. Since U is unserviced and in the region under $(2k\pi + \theta, r)$, we must have $g(2\ell\pi + \theta) < r$ by definition of the region under a point, i.e. $g(2\ell\pi + \theta) < g(2k\pi + \theta)$ for some $\ell > k$. □

An optimal algorithm. We present an optimal algorithm MONOTONE that services the requests in the following order. The disk head starts at $(0, 0)$. Let the current head position be (ϕ_0, r_0) where ϕ_0 is the actual angle through which the head has rotated. Suppose that $R = (\theta, r)$, where $0 \leq \theta < 2\pi$, is the next request to be serviced by MONOTONE, and suppose that R is serviced at angle $\phi = 2k\pi + \theta$. The following conditions are used to determine R.

1. The reachability cone rooted at (ϕ_0, r_0) contains (ϕ, r);

2. There are no unserviced requests in the region under (ϕ, r);

3. Request R is the first one that is in the cone rooted at (ϕ_0, r_0) and that satisfies condition 2. Stated differently, for any unserviced request $R' = (\theta', r')$, if there exists a k' such that $\phi' = 2k'\pi + \theta'$, $\phi_0 < \phi' < \phi$ and (ϕ', r') is in the cone rooted at (ϕ_0, r_0), then there are unserviced requests under the point (ϕ', r').

By Lemma 9.4.1 there exists an optimal solution OPT such that the monotone property is satisfied. The following theorem shows that if OPT serviced some request before MONOTONE does, then OPT would have to violate monotonicity. Hence, MONOTONE cannot perform worse than OPT.

Theorem 9.4.3 *Let* OPT *be any optimal algorithm that preserves monotonicity. Algorithms* OPT *and* MONOTONE *require the same number of rotations to service all the requests.*

Proof: To obtain a contradiction we assume MONOTONE requires more rotations than OPT. Let $R_1, R_2, \ldots$ be the order in which MONOTONE services all the requests. Let $R_j = (\theta_j, r_j)$ be the first request that OPT services before MONOTONE. Suppose OPT services R_j at angle $\phi_j = 2k\pi + \theta_j$, which implies that MONOTONE services R_j at an angle greater than ϕ_j. Let $R_i = (\theta_i, r_i)$, $i < j$, be the last request MONOTONE services before angle ϕ_j. Suppose MONOTONE services R_i at angle $\phi_i = 2k'\pi + \theta_i$, where $\phi_i < \phi_j$. There are two cases to consider. (See Figure 9-11.)

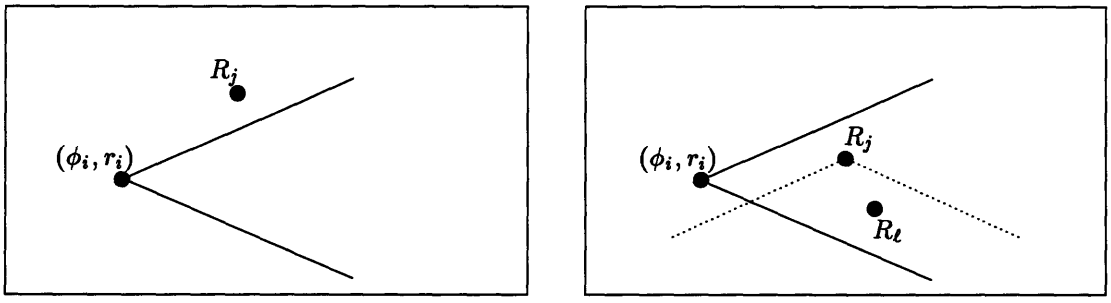

Figure 9-11: (Left) R_j is outside the reachability cone rooted at (ϕ_i, r_i). (Right) R_j is inside the reachability cone rooted at (ϕ_i, r_i).

Case 1. *R_j is outside the reachability cone rooted at (ϕ_i, r_i).* By condition 2, (ϕ_j, r_j) is above the cone. This implies that OPT cannot service R_i at an angle $\phi \in [\phi_i, \phi_j]$, since OPT services R_j at ϕ_j. By the definition of R_j, OPT services R_i no earlier than ϕ_i. Hence, OPT services R_i after R_j. Lemma 9.4.2 implies that OPT violates monotonicity.

Case 2. *R_j is inside the reachability cone rooted at (ϕ_i, r_i).* The reason that MONOTONE does not service R_j after servicing R_i is that there is some request R_ℓ under the point (ϕ_j, r_j). This request R_ℓ is not serviced by MONOTONE by angle ϕ_j, by the definition of R_i. However, MONOTONE services R_ℓ before R_j by the construction of the algorithm (i.e. $i < \ell < j$). Hence, the definition of R_j implies that R_ℓ is not serviced by OPT by angle ϕ_j. Therefore, OPT services R_j before R_ℓ. Lemma 9.4.2 implies that OPT violates monotonicity. □

9.5 A Special Case of the Asymmetric Traveling Salesman Problem

In this section we view the disk scheduling problem as a special case of the asymmetric traveling salesman problem with the triangle inequality (ATSP-Δ). We present an approximation algorithm for this ATSP-Δ problem and then obtain another approximation algorithm for the disk scheduling problem.

In the disk graph, edge lengths are nonnegative integers given by the distance function defined in Section 9.1. If the disk head makes a full seek in t_{fullseek} rotations then all edge lengths are at most $L = \lceil t_{\text{fullseek}} \rceil + 1$. However, an ATSP-$\Delta$ problem that has integer edge lengths from $[0, L]$ is not necessarily a disk scheduling problem and so the algorithms of the previous sections may not apply.

The problem we consider in this section can be formally defined as follows. We are given a graph G with n nodes and a distance function δ on these nodes. The function δ is not necessarily symmetric but it satisfies the triangle inequality, i.e. $\delta(u, v) + \delta(v, w) \geq \delta(u, w)$ for all nodes u, v and w. We first assume that all distances

are 0 or α for some $\alpha > 0$. We present an optimal algorithm for this case. Secondly we assume that all distances are either 0 or else lie between α and β where $0 < \alpha < \beta$. In this case we apply the previous result to obtain a β/α-approximation algorithm. The best known approximation ratio for a general ATSP-Δ problem is $\lceil \log_2 n \rceil$. (See [22].) We have,

Theorem 9.5.1 *Let $\alpha > 0$. If $\delta(u, v) \in \{0, \alpha\}$ for all nodes u and v then the resulting* ATSP-Δ *problem is polynomially solvable.*

Proof: We define a relation on the nodes. Let $u \sim v$ if and only if $\delta(u, v) = \delta(v, u) = 0$. By the triangle inequality this is an equivalence relation. Let $V_1, V_2, \ldots$ be the equivalence classes induced by this relation. Define the distance from equivalence class V_i to class V_j to be $\delta'(V_i, V_j) = \min\{\delta(u, v) : u \in V_i, v \in V_j\}$. One can verify that the triangle inequality holds for δ' and that if $\delta'(V_i, V_j) = 0$ then $\delta'(V_j, V_i) \neq 0$. Consider now a directed graph, H, whose nodes are the equivalence classes. A directed edge (V_i, V_j) exists if and only if $\delta'(V_i, V_j) = 0$. The graph H is acyclic, otherwise there would exist V_i and V_j such that $\delta'(V_i, V_j) = \delta'(V_j, V_i) = 0$. By the triangle inequality on δ' and the acyclicity of H, graph H induces a partial order, $(\mathcal{P}, \prec)$, on the equivalence classes. We say that $V_i \prec V_j$ in $\mathcal{P}$ if and only if there is a path from V_i to V_j in H. Two elements V_i and V_j in $\mathcal{P}$ are *comparable* if either $V_i \prec V_j$ or $V_j \prec V_i$, and they are *incomparable* otherwise. An *antichain* is a set of elements any two of which are incomparable. A *chain* is a set of elements any two of which are comparable.

Lemma 9.5.2 *Let A be the maximum cardinality of an antichain. The length of an optimal tour for our* ATSP-Δ *problem is at least αA.*

Proof: Let $\{V_1, V_2, \ldots, V_A\}$ be an antichain of size A. Since, for all i and j, $V_i \not\prec V_j$ and $V_j \not\prec V_i$, we have $\delta'(V_i, V_j) = \delta'(V_j, V_i) = \alpha$. For all i let v_i be any member of V_i. (Recall that V_i is an equivalence class of nodes in graph G.) The definition of δ' implies that $\delta(v_i, v_j) = \delta(v_j, v_i) = \alpha$ for $1 \leq i, j \leq A$. The optimal traveling salesman tour must visit all these nodes v_i. Hence the tour has length at least αA. □

It remains to show that we can find a tour that achieves this lower bound. Our algorithm is based on the following theorem. See [20, 10].

Dilworth's Theorem *If the largest antichain in a partial order $(\mathcal{P}, \prec)$ has cardinality A, then the partial order can be decomposed into exactly A chains. Moreover this decomposition can be obtained in polynomial time.*

It is clear that no decomposition can have fewer than A chains since every element of the antichain must be in a different chain. What is remarkable is that there always exist A chains that cover the whole partial order. (See [10] for a proof.)

An optimal tour is constructed from the chains in a minimum-size chain decomposition. Under the distance function δ', the total length of a chain is 0 and the distance from the end of any chain to the beginning of any other chain is at most α. Given that the size of the maximum antichain is A, we can therefore link the chains into a cycle of length at most αA. Note that this is a tour of graph H. To obtain a tour of G, we observe that once a tour has visited one node in an equivalence class it can visit all the other nodes in that class in any order without increasing its length. Hence we can construct a tour of G that has length at most αA. By Lemma 9.5.2 this tour is optimal. □

Corollary 9.5.3 *Let $\beta > \alpha > 0$. If either $\delta(u,v) = 0$ or $\alpha \leq \delta(u,v) \leq \beta$ for all nodes u and v then there exists a β/α-approximation algorithm for the resulting* ATSP-Δ *problem.*

If we assume that all nonzero distances are α and apply Theorem 9.5.1, then Corollary 9.5.3 follows. By the comments at the beginning of this section, if the disk head can make a full seek in t_{fullseek} rotations then Corollary 9.5.3 gives a $(\lceil t_{\text{fullseek}} \rceil + 1)$-approximation algorithm for the disk scheduling problem. (Typically $t_{\text{fullseek}} < 2$.)

9.6 The On-line Problem

In this section we consider the on-line disk scheduling problem. Requests arrive over time and are placed into a queue. The disk head can only service requests that are

in the queue. The goal is to maximize the throughput (i.e. service the requests at a high rate). This situation may be viewed as an on-line problem in which we have limited look-ahead. In real systems the requests are known to arrive in a "bursty" fashion [58] and so the preceding analysis of the off-line problem is useful. Suppose a large group of requests arrive together and then there is a period in which no requests arrive. We can use an off-line algorithm to service these requests.

As discussed in Section 9.1.2 many algorithms have been studied in the literature. Of these, shortest-time-first (STF) has been shown to have good throughput. (Recall that under STF the algorithm services the request that it can reach in the smallest amount of time. This is equivalent to the request that it can reach with the smallest amount of rotation.)

We propose an algorithm, CHAIN, for the on-line problem that is similar in spirit to the algorithms of Section 9.5. The key property of CHAIN is that it has better look-ahead than STF and should therefore have better throughput. (By better look-ahead we mean that it considers more than just the next request that it will service.) An interesting open problem is to obtain a meaningful comparison of the two algorithms analytically.

9.6.1 The Algorithm CHAIN

Consider the q requests that are in the queue. We construct a partial order on $q+1$ points, namely the q requests and the current position of the disk head, $P = (\theta_0, r_0)$. For simplicity assume $\theta_0 = 0$. We say that two points $R_i = (\theta_i, r_i)$ and $R_j = (\theta_j, r_j)$ (where $\theta_i, \theta_j \in [0, 2\pi)$) satisfy $R_i \prec R_j$ if and only if $d(R_i, R_j) = 0$. (Recall the distance function defined in Section 9.1.) It can be verified that this defines a partial order. Note that P is a minimal element in this partial order. Algorithm CHAIN proceeds by finding the longest chain whose minimum element is P. It moves the disk head to the request that is directly above P in this chain. (If the longest chain contains P only then the algorithm moves the disk head to an arbitrary request.) The algorithm then repeats, constructing a new partial order.

Bibliography

[1] M. Andrews, B. Awerbuch, A. Fernández, J. Kleinberg, T. Leighton, and Z. Liu. Universal stability results for greedy contention-resolution protocols. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, pages 380 – 389, Burlington, VT, October 1996.

[2] J. Aspnes, Y. Azar, A. Fiat, S. Plotkin, and O. Waarts. On-line load balancing with applications to machine scheduling and virtual circuit routing. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 623–631, 1993.

[3] B. Awerbuch, Y. Azar, and S. Plotkin. Throughput competitive on-line routing. In *Proceedings of the 34th Annual Symposium on Foundations of Computer Science*, pages 32–40, 1993.

[4] B. Awerbuch, Y. Azar, S. Plotkin, and O. Waarts. Competitive routing of virtual circuits with unknown duration. In *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 321–330, 1994.

[5] Y. Azar, A. Broder, and A. Karlin. On-line load balancing. In *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, pages 218–225, 1992.

[6] Y. Azar, B. Kalyanasundaram, S. Plotkin, K. Pruhs, and O. Waarts. Online load balancing of temporary tasks. In *Proceedings of the 1993 Workshop on Algorithms and Data Structures, Lecture Notes in Computer Science 709*, pages 119–130. Springer-Verlag, 1993.

[7] Y. Azar, J. Naor, and R. Rom. The competitiveness of on-line assignments. In *Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 203–210, 1992.

[8] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer. Non-volatile memory for fast, reliable file systems. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[9] Y. Bartal, A. Fiat, H. Karloff, and R. Vohra. New algorithms for an ancient scheduling problem. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 51–58, 1992.

[10] K. Bogart. *Introductory Combinatorics.* Harcourt Brace Jovanovich, Orlando, Florida, 1990.

[11] A. Borodin, J. Kleinberg, P. Raghavan, M. Sudan, and D. Williamson. Adversarial queueing theory. In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing*, pages 376 – 385, Philadelphia, PA, May 1996.

[12] A. Borodin, J. Kleinberg, M. Sudan, and D. Williamson, 1996. Personal communication.

[13] A. Broder, A. Frieze, and E. Upfal. A general approach to dynamic packet routing with bounded buffers. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, pages 390–399, 1996.

[14] A. Broder and E. Upfal. Dynamic deflection routing in arrays. In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing*, pages 348–355, 1996.

[15] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Mathematical Statistics*, 23:493 – 509, 1952.

[16] E. G. Coffman, L. A. Klimko, and B. Ryan. Analysis of scanning policies for reducing disk seek times. *SIAM Journal of Computing*, 1(3), September 1972.

[17] R. L. Cruz. A calculus for network delay, part I: Network elements in isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, January 1991.

[18] R. L. Cruz. A calculus for network delay, part II: Network analysis. *IEEE Transactions on Information Theory*, 37(1):132–141, January 1991.

[19] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. *Internetworking: Research and Experience*, 1(1):3–26, 1990.

[20] R. P. Dilworth. A decomposition theorem for partially ordered sets. *Ann. Math.*, 51(2):161–166, 1950.

[21] A. Elwalid, D. Mitra, and R. H. Wentworth. A new approach for allocating buffers and bandwidth to heterogeneous regulated traffic in an ATM node. *IEEE Journal on Selected Areas in Communications*, 13(6):1115 – 1127, August 1995.

[22] A. M. Frieze, G. Galbiati, and F. Maffioli. On the worst-case performance of some algorithms for the asymmetric traveling salesman problem. *Networks*, 12:23–39, 1982.

[23] G. Gallo, F. Malucelli, and M. Marrè. Hamiltonian paths algorithms for disk scheduling. Technical Report 20/94, Dipartimento di Informatica, Università di Pisa, 1994.

[24] M. Garey and D. Johnson. *Computers and intractability: A guide to the theory of NP-Completeness.* Freeman, New York, 1979.

[25] R. Gawlick, A. Kamath, S. Plotkin, and K. Ramakrishnan. Routing and admission control in general topology networks. Technical report STAN-CS-TR-95-1548, Stanford University, 1995.

[26] R. Geist and S. Daniel. A continuum of disk scheduling algorithms. *ACM Transactions on Computer Systems*, 5(1):77–92, February 1987.

[27] C. C. Gotlieb and H. MacEwen. Performance of movable-head disk storage devices. *Journal of the ACM*, 20(4), October 1973.

[28] R. L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

[29] M. Harchol-Balter and P. Black. Queueing analysis of oblivious packet routing networks. In *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 583–592, 1994.

[30] M. Harchol-Balter and D. Wolfe. Bounding delays in packet-routing networks. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, pages 248–257, 1995.

[31] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *USENIX*, pages 235–245, Winter 1994.

[32] M. Hofri. Disk scheduling: FCFS vs SSTF revisited. *Communications of the ACM*, 23(11), November 1980.

[33] D. M. Jacobson and J. Wilkes. Disk scheduling algorithms based on rotational position. Technical Report HPL–CSP–91–7rev1, Hewlett-Packard Company, 1991.

[34] N. Kahale and T. Leighton. Greedy dynamic routing on arrays. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 558–566, 1995.

[35] A. Kamath, O. Palmon, and S. Plotkin. Routing and admission control in general topology networks with poisson arrivals. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, Atlanta, Georgia, January 1996.

[36] D. R. Karger, S. J. Phillips, and E. Torng. A better algorithm for an ancient scheduling problem. In *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 132–140, 1994.

[37] F. P. Kelly. *Reversibility and Stochastic Networks.* Wiley, New York, 1979.

[38] S. Keshav. *An engineering approach to computer networking.* Addison Wesley, 1997.

[39] L. Kleinrock. *Queueing Systems.* Wiley, New York, 1975.

[40] D. Kotz, S. B. Toh, and S. Radhakrishnan. A detailed simulation model of the HP 97560 disk drive. Technical Report PCS–TR94–220, Dartmouth College, 1994.

[41] H. W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83 – 97, 1955.

[42] E. L. Lawler, J. K. Lenstra, A. H. Rinnooy Kan, and D. B. Shmoys. Sequencing and scheduling: Algorithms and complexity. In S. C. Graves, A. Rinnooy Kan, and P. Zipkin, editors, *Handbook of Operations Research and Management Science, Volume IV: Production Planning and Inventory*, pages 445–522. North-Holland, 1993.

[43] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays • Trees • Hypercubes.* Morgan-Kaufmann, San Mateo, CA, 1992.

[44] F. T. Leighton, B. M. Maggs, and S. B. Rao. Packet routing and job-shop scheduling in O(congestion + dilation) steps. *Combinatorica*, 14(2):167 – 186, 1994.

[45] F. T. Leighton, B. M. Maggs, and A. W. Richa. Fast algorithms for finding O(congestion + dilation) packet routing schedules. Technical report CMU-CS-96-152, Carnegie Mellon University, 1996.

[46] F. T. Leighton and G. Plaxton. Hypercubic sorting networks. *SIAM Journal of Computing (to appear)*, 1997.

[47] T. Leighton. Average case analysis of greedy routing algorithms on arrays. In *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 2–10, 1990.

[48] T. Leighton, 1996. Personal communication.

[49] M. Mitzenmacher. Bounds on the greedy algorithm for array networks. In *Proceedings of the 6th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 346–353, 1994.

[50] W. C. Oney. Queuing analysis of the scan policy for moving-head disks. *Journal of the ACM*, 22(3), July 1975.

[51] R. Ostrovsky and Y. Rabani. Local control packet switching algorithm. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing (to appear)*, May 1997.

[52] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice Hall, New Jersey, 1982.

[53] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: The single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344 – 357, 1993.

[54] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: The multiple-node case. *IEEE/ACM Transactions on Networking*, 2(2):137 – 150, 1994.

[55] S. Phillips and J. Westbrook. Online load balancing and network flow. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 402–411, 1993.

[56] J. Plesnik. The NP-completeness of the Hamiltonian cycle problem in planar digraphs with degree bound two. Unpublished manuscript, 1978.

[57] Y. Rabani and E. Tardos. Distributed packet switching in arbitrary networks. In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing*, pages 366 – 375, Philadelphia, PA, May 1996.

[58] C. Ruemmler and J. Wilkes. UNIX disk access patterns. In *USENIX*, pages 405–420, Winter 1993.

[59] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–29, March 1994.

[60] M. Seltzer, P. Chen, and J. Ousterhout. Disk scheduling revisited. In *USENIX*, pages 313–324, Winter 1990.

[61] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[62] J. Spencer. *Ten Lectures on the Probabilistic Methods.* Capital City Press, Philadelphia, Pennsylvania, 1994.

[63] G. Stamoulis and J. Tsitsiklis. The efficiency of greedy routing in hypercubes and butterflies. *IEEE Transactions on Communications*, 42(11):3051–3061, November 1994.

[64] D. Stiliadis and A. Varma. Frame-based fair queueing: A new traffic scheduling algorithm for packet-switched networks. Technical report UCSD-CRL-95-39, University of California at Santa Cruz, July 1996.

[65] D. Stiliadis and A. Varma. Latency-rate servers: A general model for analysis of traffic scheduling algorithms. In *Proceedings of the Conference on Computer Communications, INFOCOM'96*, 1996.

[66] L. Tassiulas and L. Georgiadis. Any work-conserving policy stabilizes the ring with spatial re-use. *IEEE/ACM Transactions on Networking*, 4(2):205–208, April 1996.

[67] T. J. Teorey and T. B. Pinkerton. A comparative analysis of disk scheduling policies. *Communications of the ACM*, 15(3), March 1972.

[68] J. Turner. New directions in communications, or which way to the information age? *IEEE Communications Magazine*, pages 8 – 15, 1986.

[69] J. Westbrook. Load balancing for response time. In *Proceedings of the 3rd Annual European Symposium on Algorithms*, pages 355–368, 1995.

[70] N. C. Wilhelm. An anomaly in disk scheduling: A comparison of FCFS and SSTF seek scheduling using an empirical model for disk accesses. *Communications of the ACM*, 9(1), January 1976.

[71] B. L. Worthington, G. R. Ganger, and Y. N. Patt. Scheduling algorithms for modern disk drives. In *SIGMETRICS*, pages 241–251, 1994.

[72] B. L. Worthington, G. R. Ganger, Y. N. Patt, and J. Wilkes. On-line extraction of SCSI disk drive parameters. In *SIGMETRICS*, pages 146–156, 1995.

[73] L. Zhang. Virtual Clock: A new traffic control algorithm for packet switching networks. *ACM Transactions on Computer Systems*, 9:101–124, May 1991.

[74] L. Zhang. *An analysis of network routing and communication latency.* PhD thesis, M.I.T., 1997.

[75] L. Zhang, M. Andrews, W. Aiello, S. Bhatt, and K. R. Krishnan. A performance comparison of competitive on-line routing and state-dependent routing. Unpublished manuscript, 1996.