

**Packet Scheduling Strategies for Emerging Service Models
in the Internet**

A Thesis

Submitted to the Faculty

of

Drexel University

by

Hongyuan Shi

in partial fulfillment of the
requirements for the degree

of

Doctor of Philosophy

August 2003

Drexel University
Office of Research and Graduate Studies
Thesis Approval Form
(For Masters and Doctoral Students)

Hagerty Library will bond a copy of this form with each copy of your thesis/dissertation.

This thesis, entitled _____

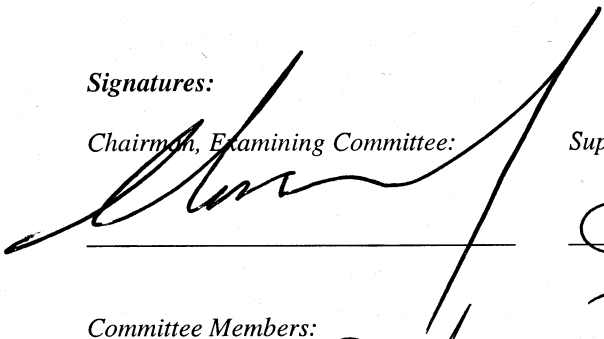
Packet Scheduling Strategies for Emerging

Service Models in the Internet

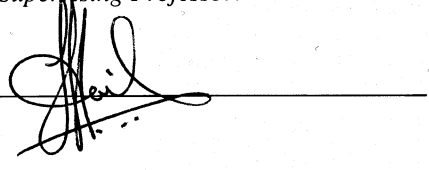
and authored by Hongyuan Shi, is hereby accepted and approved.

Signatures:

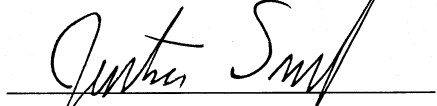
Chairman, Examining Committee:



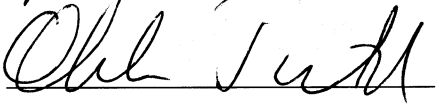
Supervising Professor:

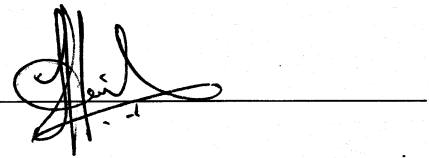


Committee Members:

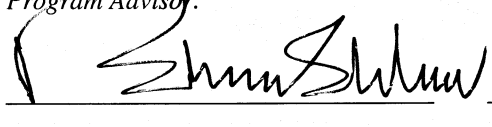




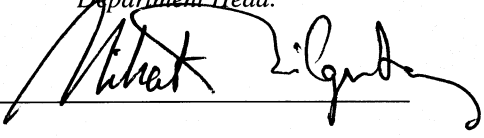




Program Advisor:



Department Head:



Acknowledgments

I would like to express my gratitude to Dr. Harish Sethu. He provided constant encouragement and made many valuable suggestions in both professional and everyday life. He helped me clarify my thoughts and make progress in my research, especially when I needed guidance in overcoming a barrier in my work. I greatly appreciate every effort he made to improve my life in all aspects.

I would like to sincerely thank Dr. Stewart D. Personick for leading me into the field of communication networks and encouraging me to continue working on computer network architecture design.

I would also like to thank Dr. Stewart D. Personick, Dr. Stanislav B. Kesler, Dr. Justin Smith and Dr. Oleh J. Tretiak for taking time to read my dissertation and serve on the committee for my Ph.D. defense.

Many thanks are due to Salil Kanhere, Yunkai Zhou, Alpa Parekh, Madhusudan Hosagrahara, Harpreet Arora, Adam O'Donnell and Kunal Shah. They all helped make the Computer Communication Laboratory a pleasant workplace and an environment conducive to learning. Special thanks to Yunkai Zhou for many inspiring discussions throughout these years. I would like to thank my friends for cheering me up and providing help whenever I need.

Finally, I am grateful to my parents and my sister for constant support and understanding throughout my Ph.D. experience.

Table of Contents

List of Tables	vii
List of Figures	viii
Abstract	xi
Chapter 1. Introduction	1
1.1 Internet Service Models	1
1.1.1 The Demand for New Service Models	1
1.1.2 Service Models Proposed	2
1.1.3 Network Mechanisms to Support Service Models	5
1.2 Requirements for Packet Schedulers	7
1.2.1 Application Requirements	7
1.2.2 Requirement of Fairness	9
1.3 Existing Packet Scheduling Strategies	10
1.3.1 Schedulers for Best Effort Services	10
1.3.2 Schedulers for Guaranteed Services	12
1.3.3 Other Schedulers Supporting QoS	13
1.4 Contributions	14
1.5 Organization	16
Chapter 2. Best-Effort and Guaranteed Services	18
2.1 Introduction	18
2.1.1 Fair Packet Schedulers	18
2.1.2 Motivation and Goals	20

2.2	Greedy Fair Queueing	21
2.2.1	Preliminaries	21
2.2.2	Handling a Newly Backlogged Flow	23
2.2.3	Choosing the Transmission Order	26
2.3	Implementation of GrFQ	27
2.3.1	Fairness Analysis	29
2.3.2	Computational Efficiency	33
2.4	GrFQ-lite	34
2.5	Evaluation of GrFQ and GrFQ-lite	37
2.5.1	A Measure of Instantaneous Fairness	37
2.5.2	Evaluation through Simulations	41
2.5.3	Results with Video Traffic Traces	41
2.5.4	Results with Gateway Traffic Traces	45
	Chapter 3. Controlled Load Service	56
3.1	Network Mechanisms for Controlled Load Service	56
3.1.1	Requirements of the Scheduler for Controlled Load Service	58
3.2	The $CL(\alpha)$ Scheduler	61
3.2.1	Tracking Changes in $ED_P(t)$	63
3.2.2	Computing EDD	67
3.2.3	Limiting ED to α	70
3.3	Analysis	71
3.3.1	Bound on the Extra Delay	72
3.4	Simulation Results	73

Chapter 4. Soft Real-Time Service	78
4.1 Background	78
4.1.1 Scheduling Real-Time Traffic	78
4.1.2 Fairness Issues in Scheduling Real-Time Traffic	80
4.1.3 Fairness of Existing Schedulers with QoS Assurance	83
4.2 SRTS-PQP: Soft Real-Time Scheduler Using Priority Queueing with Promotions	85
4.3 SRTS-CPQ: Soft Real-Time Scheduler Using Conditional Priority Queueing System	87
4.3.1 Conditional Priority Queueing	87
4.3.2 System Structure	88
4.3.3 Maintaining the Timetable	89
4.3.4 Dropping Packets	100
4.4 Analysis of the SRTS-CPQ Scheduler	101
4.5 Discussions and Evaluation	103
4.5.1 Discussions	103
4.5.2 Evaluation Based on Simulations	103
Chapter 5. Concluding Remarks and Future Work	108
5.1 Summary and Concluding Remarks	108
5.2 Future Work	111
Bibliography	113
Appendix A. Gini Index as a Measure of Fairness	118
Appendix B. Data Structure for the SRTS Scheduler	124
Vita	137

List of Tables

2.1	Settings for MPEG-4 traffic sources	42
2.2	Settings for traffic sources from gateway traces	45
3.1	Settings for MPEG-4 traffic sources and token bucket regulators	75
4.1	Variables in a <i>VacancyElement</i>	95
4.2	Settings for traffic sources from VoIP traces	104
4.3	Gini index of average delays among flows	107
B.1	Pointers in a <i>VacancyElement</i>	125
B.2	Different cases when inserting a packet into a timetable.	126

List of Figures

2.1	Pseudo-code of <i>Initialize</i> and <i>Enqueue</i> routines in GrFQ scheduler	30
2.2	Pseudo-code of <i>Dequeue</i> routine in GrFQ scheduler	31
2.3	An illustration of the difference in the disparity in service received while the upper bounds of the relative and absolute fairness measures are identical in two packet scheduling systems	38
2.4	An illustration of the Lorenz curve and Gini index	40
2.5	Gini indices of fair schedulers on backlogged queues with real video traffic traces	48
2.6	Gini indices of fair schedulers with real video traffic traces	49
2.7	Gini indices of fair schedulers with real video traffic at 90% load	50
2.8	Gini indices of fair schedulers with real video traffic at 80% load	51
2.9	Gini indices of fair schedulers on backlogged queues with real gateway traffic traces	52
2.10	Gini indices of fair schedulers with real gateway traffic traces	53
2.11	Gini indices of fair schedulers with real gateway traffic at 90% load	54
2.12	Gini indices of fair schedulers with real gateway traffic at 80% load	55
3.1	Pseudo-code of <i>Initialize</i> and <i>Enqueue</i> routines of the $CL(\alpha)$ scheduler; U , at any given time instant t , stands for $U(t)$	64
3.2	Pseudo-code of <i>Dequeue</i> routine of the $CL(\alpha)$ scheduler	65
3.3	Pseudo-code of <i>TransmitUnmarkedPacket</i> routine used by <i>Dequeue</i> routine of the $CL(\alpha)$ scheduler	66
3.4	Illustration of the sub-cases 1A and 1B in the proof of Lemma 1	69
3.5	Simulation set-up of the $CL(\alpha)$ scheduler	74

3.6	Distribution of the extra delay from $CL(\alpha)$ scheduler: (a) cumulative distribution (b) distribution density	76
3.7	The amount of data from marked packets sent by the $CL(\alpha)$ scheduler and by the ideal but more complex scheduler	77
4.1	Pseudo-code of <i>Dequeue</i> routine in the SRTS-PQP scheduler	86
4.2	Pseudo-code of <i>Enqueue</i> routine in SRTS-CPQ	90
4.3	Pseudo-code of <i>Dequeue</i> routine in SRTS-CPQ	91
4.4	Pseudo-code of <i>TransmitPacket</i> and <i>DropPacket</i> routines in the SRTS-CPQ scheduler	92
4.5	An illustration of assemblies beginning at packet x	94
4.6	An illustration of the structure of a timetable of priority p . The size of each max-assembly is: $ Y_p(1) = Y_p(3) = Y_p(6) = 2$, $ Y_p(5) = Y_p(8) = Y_p(9) = 1$	96
4.7	Incoming traffic throughout the simulation.	104
4.8	The probability of failure with different traffic loads	106
4.9	The average normalized delay of each flow with different traffic loads	107
A.1	An illustration of the Lorenz curve for computing the Gini index	120
A.2	An illustration of two distributions	122
B.1	Create a new <i>VE</i>	125
B.2	Pseudocode of <i>InsertPacket</i> routine	128
B.3	Pseudocode of <i>UpwardSearch</i> routine	129
B.4	Pseudocode of <i>OccupyVacancyLen</i> routine	130
B.5	Pseudocode of <i>DownwardSearch</i> routine	131
B.6	Pseudocode of <i>ComputeAdditionalVacancyNeeded</i> routine	132
B.7	Pseudocode of <i>UpwardUpadteOnVacancyLen</i> routine	133
B.8	Pseudocode of <i>ReleasePacket</i> routine	133

B.9 Pseudocode of <i>SearchPositionInVacancyTree</i> routine	134
B.10 Pseudocode of the insertion routine of <i>VacancyTree</i>	135
B.11 Pseudocode of the deletion routine in <i>VacancyTree</i>	136

Abstract

Packet Scheduling Strategies for Emerging Service Models
in the Internet
Hongyuan Shi
Harish Sethu, Ph.D.

Traditional as well as emerging new Internet applications such as video-conferencing and live multimedia broadcasts from Internet TV stations will rely on scheduling algorithms in switches and routers to meet a diversity of service requirements desired from the network. This dissertation focuses on four categories of service requirements that cover the vast majority of current as well as emerging new applications: best-effort service, guaranteed service (delay and bandwidth), controlled load service, and soft real-time service. For each of these service types, we develop novel packet scheduling strategies that achieve better performance and better fairness than existing strategies.

Best-effort and guaranteed services: A fair packet scheduler designed for best-effort service can also be employed to achieve bandwidth and delay guarantees. This dissertation proposes a novel fair scheduling algorithm, called *Greedy Fair Queueing (GrFQ)*, that explicitly incorporates the goal of achieving better fairness into the actions of the scheduler. A simplified version of the scheduler is also proposed for easier deployment in real networks.

Controlled load service: This dissertation analyzes and defines requirements on packet schedulers serving traffic that request the controlled load service (part of the Integrated Services architecture). We then propose a novel scheduler, called the $CL(\alpha)$ scheduler, which provides service differentiation for aggregated traffic for controlled load service. The proposed scheduler satisfies the defined requirements with a very low processing complexity and without requiring per-flow management.

Soft real-time service: We formally define the service requirements of soft real-time applications which have delay constraints but which can tolerate some packet losses. Two novel schedulers of different levels of complexity are proposed. These schedulers achieve better performance (lower overall loss rates) and better fairness than previously known schedulers.

We adapt a metric used widely in economics, called the *Gini index*, to our purpose of evaluating the fairness achieved by our schedulers under real traffic conditions. The Gini index captures the instantaneous fairness achieved at most instants of time as opposed to previously used measures of fairness in the networking literature. Using real video, audio and gateway traffic traces, we show that the proposed schedulers achieve better performance and fairness characteristics than other known schedulers.

Chapter 1. Introduction

1.1 Internet Service Models

1.1.1 The Demand for New Service Models

The Internet is a network of networks linking computers that share the TCP/IP suite of protocols. Built on the technology of packet switching, as opposed to circuit switching, the Internet permits an efficient means of data exchange for applications that do not require firm performance guarantees. Today's Internet only provides a best-effort service, in the sense that it does its best to deliver good service but provides no guarantees whatsoever on the quality of service. This service model is simple, low-cost and pushes much of the complexity to the end-systems. For example, end-systems recover from packet losses through retransmissions. Most importantly, best-effort service has been a perfectly adequate and efficient model for the vast majority of the early applications such as e-mail and ftp.

The Internet, however, is also a constantly evolving infrastructure, influenced by a number of new applications with diverse requirements. For example, streaming multimedia and real-time interactive applications, which are becoming increasingly popular, require tight bounds on the end-to-end delay and the loss rate. The best-effort service provided by the Internet is not sufficient for these applications. Architectural enhancements and additional mechanisms are required to support the quality of service (QoS) that is demanded by these classes of applications.

An obvious solution, of course, is to provide excess bandwidth in the network so as to ensure that all application requirements will always be met. However, there are several weaknesses to relying entirely upon the availability of sufficient resources. Firstly, today's applications (and not necessarily future applications) already tend to use up more bandwidth than is available at the edge networks. For example, most college campuses

today are forced to restrict the bandwidth allocated to certain kinds of applications (such as music file downloads) in order to ensure an adequate level of service to other applications. Secondly, many critical applications impose the need for the network to provide service differentiation. Thirdly and probably most importantly, QoS assurance also provides a flow protection from the behavior of other misbehaving flows. For example, in the absence of mechanisms to provide QoS assurance, denial of service attacks based on excessive consumption of a resource become possible.

In light of the above need for new service models and mechanisms to support these service models, a number of proposals have been advanced, both in the research literature as well as by the industry and the IETF. This dissertation focuses on some of these service models and proposes novel packet scheduling strategies to support these service models.

1.1.2 Service Models Proposed

There have been several service models proposed by Internet Engineering Task Force (IETF), the industry and the academic research community. Among these, the most popular and well-accepted are Integrated Services (IntServ) [1, 2] and Differentiated Services (DiffServ) [3], both of which have been formally defined by IETF in RFCs.

Integrated Services

The *Integrated Services* model defined by the IETF introduces an architecture to support two kinds of service additional to the best effort service. One is the *guaranteed service* [2] which seeks to provide delay bound guarantees to traffic flows. Such guarantees are ensured by making per-flow reservations using the Resource Reservation Protocol (RSVP), and then expecting schedulers in the routers to abide by the reservations [4]. One of the challenges in providing these guaranteed services is in the management of reservations and scheduling states corresponding to thousands of traffic flows that may all be active at the same time.

Therefore, the Integrated Services framework also specifies a more scalable option called the *controlled load* service [1].

Controlled load service is distinguished by the fact that it seeks to provide users with a quality of service similar to that in a lightly loaded or unloaded network, and without requiring or specifying a target upper bound on the delay or loss probabilities. The idea behind this service model is that many real-time applications do receive adequate performance and quality of service in a lightly loaded network, eliminating the need for very strict performance guarantees. The desired quality of service is intended to be assured through capacity planning and admission control rather than through per-flow management during packet scheduling and forwarding. When a user exceeds traffic specifications approved by the admission control policy, the service obtained by the excess packets degenerates to the best-effort service.

Differentiated Services

In order to address the scalability issues with the Integrated services framework, especially its guaranteed services option, the *Differentiated Services* (DiffServ) framework was proposed as a more scalable choice [3]. The DiffServ model is distinguished from IntServ by the fact that there is no reserved capacity at each router for individual flows or connections. Instead, the quality of service is achieved by regulating traffic and marking packets on the network boundaries. In the DiffServ model, a user first negotiates with the network about the traffic parameters, service quality and price. The result is contained in a service level agreement. The future traffic from this user will be regulated based on the contract before it enters the network. On the boundary of the network, a service classifier is used to map the flow into the suitable *Per-Hop-Behavior* (PHB) class according to its desired service quality. The traffic from the user that is sent in violation of the traffic contract is marked, so that the routers inside the network can decide how to forward the packet and

what level of service to provide to it. Since service differentiation is achieved by marking, per flow management is not necessary. This reduces the processing associated with the scheduling and forwarding of packets at the network nodes, and in addition, allows a more scalable mechanism for achieving quality of service.

To satisfy the vast variety of requirements from diverse applications, appropriate PHB classes and the associated forwarding mechanisms should be defined. *Assured Forwarding* (AF) and *Expedited Forwarding* (EF) are the two PHBs that have been defined by IETF [5, 6]. Assured Forwarding has different PHB classes with different levels of assurance in delivery, while Expedited Forwarding includes only one PHB class with a strong requirement on low delay and steady guarantee of bandwidth. The properties of EF-PHB are particularly suitable for real time communication, such as audio or video teleconferencing.

Besides Expedited Forwarding and Assured Forwarding, more PHB structures have been proposed to fit into the DiffServ framework. In [7], *Dynamic RT/NRT (DRT)-PHB* group is defined with two PHB classes: *RT* class for real-time traffic; *NRT* class for flows without strict delay constraint. Each class has six PHBs, representing different importance levels. DRT-PHB group can also be extended to include more classes and importance levels.

Other Service Models

An alternate service framework is based on differentiating quality of service relatively instead of absolutely [8]. The model is called *Relative Differentiated Services*. With this model, the network does not provide assurance on the absolute quality of service. Instead, it assures that a higher class has a better QoS than a lower class. It is up to the applications to select the right class for its purpose. Therefore, the network only provides relative QoS classes for end systems to choose.

Another architectural framework, called *SCORE* [9], provides guaranteed services while maintaining scalability for core networks. It does not have per flow information management, but adds an additional QoS state to each packet which, however, is incompatible with current IP protocol. Yet another service model, *Asymmetric Best-Effort* [10], provides a “throughput versus delay jitter” differentiated service for IP packets. In this approach, best effort packets are marked as either *Green* or *Blue*. Green packets receive less delay jitter but may experience high losses during congestion intervals. These two classes are still best effort and the traffic sources need to be TCP-friendly.

1.1.3 Network Mechanisms to Support Service Models

A network service model is supported and implemented through several network mechanisms, such as packet scheduling, buffer management, flow control, congestion control, and admission control. A packet scheduling algorithm determines which packet among those awaiting service should be transmitted next. It is the key component affecting bandwidth allocation and service delay since it decides the exact sequence in which packets should be transmitted. Besides making decision on transmission sequence, a scheduling algorithm is also responsible for making decisions on which packets to drop in the presence of congestion.

Buffer management schemes are responsible for making decisions regarding which packets should be stored to await for transmission and how they are stored (for example, in common FIFO queues or per-flow queues). It is closely related to packet scheduling algorithms since a packet should first be stored before transmission. During periods of network congestion when buffer resources become scarce, the buffer management strategy becomes critical to overall performance and QoS achieved by the flows. Frequently, buffer management strategies are very closely tied to the scheduling discipline. However, once a packet is accepted and stored in the buffer, the length of time it takes to begin transmission

depends on the scheduling algorithm.

Packet scheduling strategies, in combination with buffer management schemes, are typically responsible for determining which packets to drop. For example, some multimedia sources may generate packets of multiple priorities using “scalable” coding schemes [11, 12]. In such cases, successful delivery of lower priority packets is desired but not critical for overall quality. The network mechanisms may determine the quality of receiver playback by dropping packets of lower priority whenever necessary. Such mechanisms are able to exploit the elasticity in the application with respect to packet loss rates and still deliver reasonable quality of transmissions.

Flow control mechanisms are used to regulate the traffic injected into the network from applications. Such a mechanism is necessary since traffic arrival characteristics are typically not very smooth and since QoS assurance is easier with smooth traffic. Usually, the most basic parameters used for flow control are average rate, peak rate and burst size. In spite of flow control mechanisms, traffic from different parts of the network may have random correlations and thus, still lead to bursty traffic at certain points in the network causing congestion.

Congestion control mechanisms are used to limit, shape or divert traffic so that the probability of congestion is reduced and thus enhancing QoS assurance. For example, video sources are designed to have rate control mechanism so that it has the ability to reduce the transmission rate for the purpose of congestion control [13]. A more common approach to achieving congestion control is to send feedback information to traffic sources [14]. Upon receiving a message about potential congestion, sources reduce their sending rates appropriately. However, such a mechanism, besides being voluntary on the part of end-users, also involves some inherent delay in the mitigation of congestion after it is first detected. Therefore, it is difficult to completely avoid congestion unless the network is under utilized.

In conjunction with all the above mechanisms, a good admission control strategy is

typically also essential for QoS assurance. The role of admission control is to protect the service quality for existing traffic in the network by determining whether or not to accept a new service request based on current resource utilization. A secondary goal of the admission control strategy is to achieve as high a utilization of the network as possible.

1.2 Requirements for Packet Schedulers

A packet scheduling strategy is a critical component of the mechanisms that are necessary to achieve the QoS desired by flows of traffic. A goal in the design of packet schedulers is to satisfy both user requirements such as delay and bandwidth demands and the network requirements such as efficiency of implementation. In the following, we discuss the specific requirements of both applications and the network as regards packet scheduling.

1.2.1 Application Requirements

An excellent taxonomy of application requirements may be found in [15]. In this dissertation, we focus more specifically on the requirements that are particularly relevant to the work presented in the later chapters. For traditional data applications, such as message exchanges, file transfers and database inquiries, transmission throughput is the most important performance metric. These applications usually have flexible requirements on transmission delay and rate although they do benefit from low delays and high rate. This is the reason that best effort service model is considered sufficient for such applications. Usually, such applications also require that there be no packet losses. However, since the transmission delay requirements are loose, packet losses can be overcome by retransmission of the lost packet by the traffic sources. Some data transfer applications may require higher assurance on packet delivery, but their basic requirement is still high throughput.

The requirements of multimedia applications, however, are significantly more complex. Usually multimedia information is contained in large files; for example, a typical MPEG-4

coded movie of reasonably good quality is about 700 Mbytes. To access recorded multimedia information, one can first download the entire file completely from the source to the local memory and play it back later. In such an operation, the requirements on the network are same as with traditional data traffic. However, since multimedia files tend to be larger than normal data files, sharing multimedia in such a fashion is very inefficient since a multimedia file takes a large amount of local memory resource and can also waste a significant length of time just in the downloading. Since the playback of a multimedia file is sequential, if the playback starts when the first part of the multimedia file arrives, the data can be consumed gradually as time elapses. Once the data is used, it is discarded, saving memory space and eliminating much of the idle wait time associated with downloading. As a result, the receiver only needs a small amount of memory space to store unplayed data. Such a technique, used routinely today, is called *streaming* multimedia transmission. Clearly, streaming operation has a stricter requirement on transmission quality, especially delay and throughput, than normal data transfer applications. Even though streaming applications are real-time applications, they are not two-way interactive applications which require even stringent requirements on delays and bandwidth.

Examples of two-way interactive real-time applications include video conferencing and audio communications (such as Voice-over-IP). In such applications, the quality of service is primarily related to the transmission delay. Since the characteristics of different real-time applications vary significantly, there exist diverse requirements on the transmission delay. Usually transmission delay should not exceed a certain pre-determined bound, such as about 100-200ms for interactive voice and video applications. Otherwise, the transmitted packet is considered to be of no use to the receiver and is equivalent to having been lost. Multimedia packets usually carry information on when the data in a packet is to be played by a receiver in relation to other packets. Some receivers implement a fixed playback delay, i.e., after a certain amount of fixed delay, they play all received packets strictly according to the information carried by the packets. In such cases, the receiver only cares as to whether

or not a packet arrives earlier than a pre-determined time, but does not care as to how early the packet arrives [15]. However, in some applications such as Voice-over-IP, there may be periods of silence during which the receiver may adjust its playback delay based on the current observed end-to-end delays. Thus, for example, a brief silence by a speaker may become just slightly more brief when played by the receiver but overall, such adaptivity to current observed delays allows better quality of transmission. Many audio and video conferencing applications, including *vic* and *vat*, implement such adaptive mechanisms to adjust the playback delay based on the most recent observations of the end-to-end delay. Such applications, while requiring an end-to-end delay bound, benefit significantly if the delay is as low as possible.

From the aspect of data losses, real-time applications can usually be classified as either *hard* or *soft*. Hard real-time applications require that all packets are received within a certain end-to-end delay bound and no packets are lost. Such hard requirements are typical in some video games as well as many real-time control systems. Soft real-time applications can tolerate some packet losses while also maintaining reasonable quality of service. For example, voice applications can tolerate the loss of a few scattered packets without significant loss in perceived quality.

1.2.2 Requirement of Fairness

We now focus on requirements considering the fact that multiple applications or flows of traffic will share the network all at the same time while each application also has certain requirements as described above. Flows of traffic share resources in the network as they traverse the network. Fairness is both an intuitively desirable and practically valuable property in the allocation of network resources amongst the flows. For a network providing best effort service, a fair packet scheduler will help to protect well behaved flows from ill behaved flows which would otherwise consume all the bandwidth resource in the

network [16]. Even in a network that implements a supporting mechanism for guaranteed services, fairness is still desired by users who share network resources. Ideally, the network should not overwhelmingly satisfy one user while just barely satisfying another user, even when both users are actually satisfied. Again, this is not merely because it is intuitively desirable to be fair but also because the features of a network that allow such preferential treatment to a flow can be readily exploited by attackers for a variety of malicious purposes. Fairness is especially critical, and will be demanded by consumers, when there are prices charged to consumers for services rendered.

There exist several notions of fairness. A common notion of fairness, *max-min fairness*, has been proposed for best effort networks. It defines principles to allocate a resource fairly to flows based on the intuition that no flow should receive more service than its demand and that no flow with an unsatisfied demand should receive less service than any other flow [17, 18]. The rationale behind this criterion is that the flows have equal rights to the link resource [16]. For networks with integrated services, this notion of fairness is extended to *utility max-min fairness* [19]. Flows in a network with integrated services may have different utilities even though they are allocated the same amount of resources. Therefore, the allocation should be fair such that the utilities of flows satisfy max-min criterion.

1.3 Existing Packet Scheduling Strategies

Many packet scheduling disciplines have been proposed for best effort services, guaranteed services and differentiated services networks. Here, we will provide a brief summary.

1.3.1 Schedulers for Best Effort Services

In early computer networks, First Come First Serve (FCFS) was a popular scheduling discipline since it results in minimal overall delays for all flows. However, flows are not protected and isolated from the impact of others. A packet-by-packet round robin ap-

proach of scheduling was introduced to overcome this weakness; however, since packets in networks come in a variety of lengths, a flow can easily capture more than its share of bandwidth by transmitting large packets. Generalized Processor Sharing (GPS) is a hypothetical and unimplementable scheduler that exactly achieves the goal of max-min fairness [20]. During each infinitesimal interval of time, the GPS scheduler visits each backlogged flow once and schedules an infinitesimal amount of data proportional to the flow weight for transmission over the output link. Over the last decade, a number of different packet-by-packet scheduling algorithms have been proposed that seek to approximate the GPS scheduler. *Weighted Fair Queueing (WFQ)* [16, 20] and its variants [21–23] try to emulate the GPS scheduler by time-stamping each arriving packet with a *finish number*, the expected completion time of the packet if it were scheduled by the ideally fair GPS scheduler, and serve the packets in increasing order of their finish numbers. Thus, these schedulers seek to emulate GPS through preserving the same order in the packet transmissions as in GPS. Such schedulers are known as *timestamp-based schedulers*.

Another class of scheduling algorithms are those based on round-robin or *frame-based* approaches which do not achieve as good a fairness as most of the schedulers discussed above but which are significantly simpler to implement in both hardware and software. A well known example is *Deficit Round Robin (DRR)* [24]. While serving flows in a round-robin order, DRR eliminates the unfairness due to different packet lengths by keeping a *deficit counter* for each queue to measure the past unfairness. The amount of service allocated to each flow is determined by the *quantum* of the queue and the deficit counter from the last service. A similar method was proposed in [25, 26], later known as *Surplus Round Robin (SRR)*. Several variants of DRR have been proposed to further reduce the unfairness of the scheduling decision, such as *Nested-DRR* [27] and *Pre-order DRR* [28]. All the above schedulers require the knowledge of the maximum packet length to ensure a low computation complexity. A novel frame-based scheduler, *Elastic Round Robin (ERR)* [29], uses a method which does not need the maximum packet length to achieve fairness with low

complexity. It adjusts the amount of service to each flow based on the latest service history. Therefore, ERR achieves the best performance of fairness while maintaining per packet computation complexity of $O(1)$. Recently, ERR is further improved as proposed in [30]. The new scheduling discipline, called *Prioritized Elastic Round Robin (PERR)*, borrowing the principle used in Pre-order DRR, re-orders the transmission sequence within each frame of ERR. It achieves fairness close to that of timestamp-based schedulers with a lower processing complexity than timestamp-based schedulers.

1.3.2 Schedulers for Guaranteed Services

Schedulers for guaranteed services can be classified into two groups, work-conserving and non-work-conserving. A work-conserving server is not idle whenever there exists a packet awaiting transmission. In a non-work-conserving system, no packet may be transmitted even though there are packets available to transmit. A non-work-conserving scheduler is typically used where packet arrival characteristics can be predicted or when it is desirable to shape the traffic in a particular fashion for the next scheduler in the path of the traffic.

Many fair schedulers for best effort service have an important characteristic that the upper bound on the delay experienced by a flow is directly related to the bandwidth allocated to that flow. This property makes fair schedulers capable of providing guarantees. Such schedulers include both frame-based and timestamp-based schedulers.

A different approach to achieving guaranteed delays, based on deadlines, is used in the Delay Earliest-Due-Date (Delay-EDD) scheduler [31]. The Delay-EDD scheduler transmits packets in the order of packet deadlines. The deadline of each packet is computed based on the service contract and previous service history of the associated flow. The scheduler cannot provide a service guarantee by itself; the guarantee depends on the assumptions made about traffic characteristics. However, because the system is work-conserving, the

traffic pattern can be easily distorted by the random behavior within the network. Therefore, it is desirable to maintain a traffic pattern on the output link that is similar to traffic arrival characteristics. Jitter-EDD [32] extends Delay-EDD by adding a regulator before the scheduler. The regulator holds a packet until it is eligible for transmission. The eligibility time is computed based on the difference between transmission time and the deadline of the packet at the upstream scheduler. Stop-and-Go [33] uses a framing method. The scheduler holds a packet until the new frame starts.

1.3.3 Other Schedulers Supporting QoS

Many other schedulers have been proposed to serve QoS requirements besides the simple services discussed above. The Waiting Time Priority (WTP) scheduler was proposed three decades ago [34] and is recently employed in the context of relative differentiated services [8]. It is adopted to provide proportional delay to different service classes. Each class is associated with a weight. The system monitors the head waiting time of each class normalized by the class weight. The scheduler will service the class with the maximum normalized head waiting time so that waiting times of all classes are close to equalized. Another approach, the Mean-Delay Proportional (MDP) scheduler [35], attempts to equalize the normalized mean delay among different service classes. The mean delay is defined as minimum possible average delay of packets previously transmitted and currently awaiting for transmission in the queue. Therefore, the MDP scheduler makes decisions based on both service history and current system situation. A similar approach, named as Hybrid Proportional Delays (HPD) [36], uses a delay metric which combines the average delay of transmitted packets and the waiting time of current head packet in the queue. The importance ratio between the two parameters is selected through an empirical study to approach the theoretical model of Proportional Delay Differentiation (PDD). Again the service discipline of the HPD scheduler attempts to equalize the normalized delay metric among service

classes.

Instead of providing proportional average delays to the flows, the Weighted Earliest Due Date (WEDD) scheduler [37] attempts to provide proportional deadline violation probabilities to service classes. The rationale behind this design is that some voice communication applications do not care about the delay of each packet, but rather the probability of missing the deadline. However, it cannot provide enough assurance to applications requiring packet losses be scattered and not concentrated in a burst.

1.4 Contributions

This dissertation focuses on the design of novel packet scheduling strategies for four important service models: best-effort, guaranteed, controlled load and soft real-time. Our solution for best-effort schedulers also serves as a solution for guaranteed services for real-time applications. For each of these service models, we propose a novel packet scheduler that achieves better performance and/or fairness than previously known schedulers.

We present *Greedy Fair Queueing (GrFQ)*, a novel scheduler that explicitly incorporates the goal of achieving a better fairness into the actions of the scheduler. The relative fairness bound (RFB), first used in [21], is a popular measure of fairness that is most frequently used to judge the fairness achieved by a scheduler. We prove that the fairness achieved by GrFQ, based on the RFB, is extremely close to that of the best among known fair schedulers. Further, we shown that it achieves a better bound on the *normalized* lag than other known schedulers [23]. The per-packet dequeuing complexity of GrFQ is $O(\log N)$ with respect to the number of flows.

We further argue that existing measures of fairness do not accurately capture the actual fairness achieved at most instants of time, and therefore, do not represent a true measure of the ability of a scheduler to successfully deliver end-to-end quality for applications, especially real-time applications. A new measure of fairness based on *Gini index* is proposed.

This measure captures the instantaneous fairness of a scheduler and, unlike other measures based on bounds, also captures the fairness of the scheduler in its handling of flows during idle periods. Both of these characteristics of the new measure are important for the design of scheduling disciplines for multimedia traffic, since such traffic tends to have heavy-tailed distribution of packet lengths and highly irregular patterns of backlogged and idle periods. With the Gini index as the measure of instantaneous fairness and using real video traffic traces and real gateway traffic traces, we show that the GrFQ scheduler achieves better fairness than any other known scheduler at virtually all instants of time. We further propose a simplified version of the scheduler, called GrFQ-lite, which avoids the emulation of a fluid flow system and has a per-packet work complexity of $O(1)$ in the computation of the timestamps. Using real traffic traces again, we demonstrate that GrFQ-lite is also able to achieve close to or better fairness than most other schedulers including those that are significantly more computationally intensive in their emulation of the ideally fair fluid-flow GPS system.

As discussed earlier, fair schedulers are able to also serve as schedulers for guaranteed services such as bandwidth. The GrFQ scheduler, because it achieves the better fairness than other schedulers, also achieves better delay guarantees than other known schedulers. Fair queueing schedulers, however, cannot be used to efficiently serve applications with quality requirements other than bandwidth. Other service models, such as the controlled load service and soft real-time services, impose unique new requirements on the schedulers and cannot be served by fair queueing schedulers.

In designing a scheduler for controlled load service, we begin with analyzing the requirements of applications using the controlled load service defined within IETF's Integrated Services architecture. The controlled load service requires source points to regulate the traffic and mark packets that are sent in violation of the traffic contract. One of the requirements we define is that the additional delay of unmarked packets caused due to the transmission of marked packets should be bounded. An $O(1)$ scheduler to achieve this

bound is non-trivial. In this dissertation, we propose the $CL(\alpha)$ scheduler, which bounds this extra delay to α or less. Using real traffic traces, we show that the $CL(\alpha)$ scheduler meets its design requirements. The principle used in this algorithm may also be used to schedule flows with multi-level priorities, such as in some scalable real-time video streams as well as in other emerging service models of the Internet that mark packets to identify drop precedences [3, 38, 39]. In such cases with multiple levels of drop precedences, the principle of the $CL(\alpha)$ scheduler would have to be applied in a hierarchical manner to bound the impact of each lower priority layer on the delays experienced by higher priority layers.

Since the $CL(\alpha)$ scheduler only provides loose quality differentiation on delay, it cannot satisfy applications with both loss and delay constraints. Many soft real-time applications have requirements on transmission deadlines and a maximum packet loss rate. Further, current schedulers for soft real-time applications cannot achieve fairness when approaching the service goal, the real performance achieved is not satisfactory. We study the characteristics of soft real-time applications and define the requirements on the scheduler in this context. We propose two new schedulers for soft real-time services, which achieve significant improvement in the packet loss rates achieved. Further, one of our schedulers, while achieving better performance than other previously proposed schedulers, also achieves better fairness amongst the flows. We analytically prove the work complexity of our schedulers. Using real Voice-over-IP traffic traces, we also demonstrate the better results achieved by our schedulers.

1.5 Organization

The rest of this dissertation is organized as follows. Chapter 2 presents Greedy Fair Queueing (GrFQ) discipline and the new measure for instantaneous fairness, Gini index. Chapter 3 illustrates the design of $CL(\alpha)$ scheduler. In Chapter 4, two schedulers for soft

real-time applications are proposed which reduced the probability of packet losses and also achieve better fairness in the distribution of the packet delay. Finally, Chapter 5 concludes the dissertation.

Chapter 2. Best-Effort and Guaranteed Services

2.1 Introduction

2.1.1 Fair Packet Schedulers

Fair packet schedulers serve to achieve fairness in the case of best-effort traffic, and can also serve to achieve delay and bandwidth guarantees required by real-time traffic [18]. In this chapter, we design a novel scheduler, *Greedy Fair Queueing (GrFQ)*, that may be used for both best-effort service and guaranteed services.

As illustrated in Section 1.3.1, the Generalized Processor Sharing (GPS) model provides an ideally fair but unimplementable traffic scheduling discipline. A number of different packet-by-packet schedulers have been proposed over the last decade that seek to approximate the GPS scheduler. The earliest such algorithm was *Weighted Fair Queueing (WFQ)* [16, 20] which tries to emulate the GPS scheduler by time-stamping each arriving packet with a *finish number*, the expected completion time of the packet if it were scheduled by the ideally fair GPS scheduler. The WFQ scheduler then serves the packets in increasing order of their finish numbers. Thus, the WFQ scheduler seeks to emulate GPS through preserving the same order in the packet transmissions as in GPS. A number of different variants of WFQ have been proposed which seek to either improve the accuracy or reduce the complexity in the computation of the finish number. *Self-Clocked Fair Queueing (SCFQ)* [21], uses the finish number of the packet currently being transmitted in the computation of finish numbers, and thus achieves an easier implementation and also, very good fairness. *Start-time Fair Queueing (SFQ)* [22] is a variant of SCFQ which uses the starting time of the packet currently in service to compute the timestamp of the arriving packet. Certain other variants of WFQ use additional eligibility criteria in the determination of the next packet to transmit. For example, *Worst-case Fair Weighted Fair Queueing*

(WF^2Q) [23] transmits the packet with the lowest finish number among those that would have already begun transmission under the GPS scheduler.

A different approach to the design of fair scheduling algorithms was proposed by Stiliadis and Varma based not on an explicit emulation of GPS through the use of finish numbers, but instead based on periodically re-calibrating a global variable indicating the progress of the scheduler. This reduces the complexity of timestamp computations and can be shown to permit the design of provably fair and computationally efficient schedulers such as *Starting Potential Fair Queueing (SPFQ)* [40]. A somewhat similar approach of time-stamping flows instead of packets, leading to a similar level of computational efficiency, is used in *Time-Shift Scheduling (TSS)* [41]. In this scheme, each flow is assigned an increasing time-stamp and the packet chosen for transmission is from the flow with the least time-stamp.

In general, time-stamp based schedulers usually consist of two components that define the scheduler: the method of computation of the timestamp and the method used to determine the transmission order based on the timestamps. The method of computing the timestamp is either based on the GPS reference system (as in WFQ and WF^2Q) or is independently computed based on tracking of the system progress (as in SPFQ and TSS). The determination of the transmission order is typically based either only on transmitting the packet with the smallest time-stamp first (as in WFQ, SCFQ, SFQ, SPFQ and TSS) or on using additional eligibility criteria based on the GPS reference system (as in WF^2Q). Note that eligibility criteria can be used only in reference to the GPS scheduler, since only the GPS system can provide meaningful criteria for eligibility.

Another class of scheduling algorithms are those based on round-robin or frame-based approaches [24,29] which do not achieve as good short-term fairness as most of the schedulers discussed above but which are significantly simpler to implement in both hardware and software.

2.1.2 Motivation and Goals

The fairness of scheduling algorithms is most commonly judged by the *relative fairness bound (RFB)* [18, 42], a popular measure of fairness first used in [21] and later used in several other works [22, 24, 29, 40, 41]. The RFB captures the maximum possible difference between the normalized service received by any two backlogged flows, and therefore, serves as a measure of fairness. The RFB of the ideally fair GPS scheduler, of course, is 0.

The design of a real packet-by-packet scheduler based on achieving the same or similar order of packet transmissions as GPS does not necessarily lead to a close approximation of the GPS scheduler with a low RFB. For example, the WFQ scheduler can be shown to allow a flow to lead other flows by an arbitrarily large amount [23]. While not all of the schedulers discussed above have used an emulation of GPS finish time to achieve fairness, none have explicitly incorporated the desired result such as a low RFB directly into the design methodology of the scheduler and into the packet-by-packet actions of the scheduler. Such goal-oriented action taken by the scheduler at each packet boundary leads to another significant benefit beyond achieving a low RFB. This has to do with the fact that the RFB is only a bound and does not quite capture the overall fairness of the scheduler. For example, a scheduler that rarely reaches the upper bound of the fairness measure will not, even though it should, achieve a better measure of fairness than another scheduler that frequently or almost always operates at the same upper bound. A goal-oriented greedy scheduler can not only achieve a low RFB by seeking to minimize it with its decision at each packet boundary, but it is also more likely to achieve a low difference in the normalized service among any two flows at all instants of time.

Fairness at most instants of time as opposed to merely a low RFB offers practical value to many applications. Real-time applications called playback applications solve the problem of variations in the delay by using a playback buffer at the receiver. Incoming data is buffered and then replayed at a certain playback point. Adaptive playback applications

such as `vic` and `vat` dynamically change the playback point based on the current observed delay [43] and therefore, benefit from the low and stable latency behavior that comes with better fairness at most instants of time. This forms an important aspect of our motivation behind this work.

2.2 Greedy Fair Queueing

2.2.1 Preliminaries

Throughout the rest of this dissertation, we use the words “session” and “flow” interchangeably. Consider a set of flows, $1 \leq i \leq N$, which share an output link of peak bandwidth rate R . Flow i is associated with a weight w_i . Let $\mathbf{B}(t)$ be the set of backlogged flows at time t and let $W(t)$ be the sum of the weights of these backlogged flows. The smallest of the weights is denoted by w_{\min} , and the largest by w_{\max} . The total service received by flow i over time interval (t_1, t_2) under a given scheduling discipline P is denoted by $S_i^P(t_1, t_2)$. If the scheduling discipline under consideration is GPS, this total service is denoted by $S_i^G(t_1, t_2)$.

An early measure of fairness achieved by a scheduler, used in the works by Bennett and Zhang [23] and also by Parekh and Gallager [20], captures the upper bound on the difference in the service received with the real scheduler and that with GPS since the beginning of a backlogged period for any given flow. More formally, the measure is defined as follows:

$$\max_{\forall \tau \geq 0} \left(S_i^G(0, \tau) - S_i^P(0, \tau) \right).$$

This measure, however, does not use normalized service, i.e., service received by a flow normalized by its weight, and therefore, is not as adequate a measure of fairness as the RFB. We now provide a formal definition of the RFB.

Definition 1 *Let (t_1, t_2) be an interval of time during which all flows under consideration are continuously backlogged while being served by the scheduling policy, P . The relative*

fairness measured with respect to a pair of flows (i, j) over time interval (t_1, t_2) , denoted by $RF_{(i,j)}(t_1, t_2)$, is defined as,

$$RF_{(i,j)}(t_1, t_2) = \left| \frac{S_i^P(t_1, t_2)}{w_i} - \frac{S_j^P(t_1, t_2)}{w_j} \right|. \quad (2.1)$$

The relative fairness with respect to a flow i over time interval (t_1, t_2) , denoted by $RF_i(t_1, t_2)$, is defined as,

$$RF_i(t_1, t_2) = \max_{\forall j} RF_{(i,j)}(t_1, t_2) \quad (2.2)$$

The relative fairness over time interval (t_1, t_2) , $RF(t_1, t_2)$, and the relative fairness bound (RFB) can be defined as,

$$RF(t_1, t_2) = \max_{\forall i} RF_i(t_1, t_2) \quad (2.3)$$

$$RFB = \max_{\forall (t_1, t_2)} RF(t_1, t_2). \quad (2.4)$$

The relative fairness bound has been frequently used in the evaluation of the fairness of several scheduling disciplines [21, 22, 24, 29, 40, 41]. A related measure of fairness is called the *absolute fairness bound (AFB)* that captures the upper bound on the difference between the normalized service received by a flow under the scheduler P and that it would receive under the ideally fair GPS scheduler [18]. It can be shown that the absolute and relative fairness measures are related to each other by a simple relationship [42]. Therefore, and for historical reasons, we discuss only the relative fairness bound in this chapter.

For convenience of notation in our subsequent discussions on relative fairness, we introduce the following definition.

Definition 2 Let \mathbf{M} be a finite set of size greater than one and consisting of positive real numbers as its elements. We define the spread of the set, denoted by $D(\mathbf{M})$, as the absolute difference between the largest and the smallest elements of the set. More formally, the spread is defined as:

$$D(\mathbf{M}) = \max_{x, y \in \mathbf{M}} |x - y| = \max\{\mathbf{M}\} - \min\{\mathbf{M}\}$$

Note that $\text{RF}(t_1, t_2)$ is nothing but the spread of the set of real numbers corresponding to the normalized service received by the flows in the interval (t_1, t_2) .

2.2.2 Handling a Newly Backlogged Flow

Recall that the RFB is the maximum difference between the normalized service received by any two flows over any given interval of time during which the flows are both continuously backlogged. If τ is the instant of time that all flows become backlogged, a greedy scheduling strategy with the goal of achieving a low RFB is simply one that minimizes the value of $\text{RF}(\tau, t)$ at each instant t that a decision is made regarding the choice of the next packet to transmit. In this case where all flows have been backlogged for identical lengths of time, each flow deserves the same amount of total normalized service. During the execution of a scheduler, however, a flow may change from an idle state to that of being backlogged or vice-versa quite frequently. Since a flow at an idle state should not receive any service, a fair scheduler should not allocate the same amount of total normalized service to a flow that is just backlogged as it would to another flow which has been backlogged for a long time. When the current set of backlogged flows have been continuously backlogged for different lengths of time, as in almost any real scenario, one cannot use a common interval over which to compare the normalized service received by flows and obtain a meaningful value for the relative fairness.

Consider a case in which N flows have been continuously backlogged during the interval $(0, t)$. With the ideally fair GPS scheduler, a flow i with weight w_i receives a normalized service equal to another backlogged flow j with weight w_j , i.e.

$$\frac{S_i^G(0, t)}{w_i} = \frac{S_j^G(0, t)}{w_j} \quad (2.5)$$

Thus, at time t , the above indicates the ideal amount of service each backlogged flow should have received.

Now, consider a case in which several flows have been backlogged since time t_1 . One

of these flows, flow i , changes its status from being backlogged to idle at time $t_2 > t_1$ and later becomes backlogged again at time $t_3 > t_2$. In order to accurately and meaningfully compare the service received by all the flows at instants after t_3 , we would have to assign an appropriate value of the normalized service received by flow i until time t_3 so that the comparison is over the entire interval (t_1, t_3) . Here we claim that to fairly treat a newly backlogged flow is to neither favor it nor punish it for its idle period in the interval (t_2, t_3) . Therefore, based on (2.5), for purposes of comparisons between the service received by flows, we should assume that flow i at time t_3 has received service equal to¹,

$$w_i S_j^G(t_1, t_3^-) / w_j \quad (2.6)$$

However, if flow i has already received more than the above amount of service before time t_2 while it was backlogged, then total assumed service should be $S_i^P(t_1, t_2)$. This is because a flow that receives excess service should not be able to become idle and then immediately become backlogged again without being disadvantaged later for the excess service it received earlier. These concepts and similar arguments have also been made in [21, 40, 44].

In order to correctly assign each flow a value that measures the service it has received thus far based on the above method, we define a state variable called the *session utility*, denoted by $u_i(t)$, for each flow i as a function of time.

Assume that the system starts at time 0. During the period (t_1, t_2) that a flow i is continuously being serviced, its session utility is updated as follows:

$$u_i(t_2) = u_i(t_1) + S_i^P(t_1, t_2) / w_i \quad (2.7)$$

We now discuss how to update the session utility of a flow that just becomes backlogged. Let flow i become backlogged for the first time or backlogged again at time t . Our

¹The notation “ τ^- ” denotes the time instant just before time τ .

goal in assigning a session utility value to flow i at time t is to ensure that the comparison between session utilities of all the flows is being made as though the flows have all been backlogged for the same length of time. Accordingly, flow i is assigned the following session utility value:

$$u_i(t) = \max\{u_i(t-), v(t)\} \quad (2.8)$$

where $v(t)$ is the GPS virtual time, which is actually the normalized service received by a continuously backlogged flow in the GPS reference scheduler².

Under certain circumstances, a flow can be unbacklogged in the GPS system while still being backlogged in the real system if the transmission of its packets is delayed in the real system. At the time when this flow becomes backlogged again in the GPS system, its session utility should incorporate the amount of service it would receive during the unbacklogged interval in the GPS system. However, if the flow is still backlogged in the real system, such changes of the flow state in the GPS system are not explicitly shown by the cumulative service. To capture this situation, each packet is also timestamped with the *beginning utility*. The beginning utility of packet k in flow i , or $BU_{i,k}$, is defined as the utility of flow i at the time when packet k is ready for transmission. In order to set the beginning utility correctly, each flow needs to maintain the finishing utility of the last packet in the queue. Suppose the last packet in flow i at time t is packet z . The finishing utility of flow i at time t , or $FU_i(t)$, is defined as the utility of flow i at the time instant when packet z finishes transmission. Thus, at time t , if packet k arrives at flow i , the beginning utility of this packet is set as:

$$BU_{i,k} = \max\{FU_i(t-), v(t)\}$$

The finishing utility of flow i would then be updated as:

$$FU_i(t) = BU_{i,k} + L_{i,k}/w_i$$

²The reader is referred to [45] for detailed discussion on the computation of the GPS virtual time.

where $L_{i,k}$ is the length of packet k of flow i . When a transmission is finished, session utility should be updated based on its previous value and the beginning utility of the new head packet in the queue. Suppose a packet transmission from flow i finishes at time t , and the new head packet is packet h . The flow utility should be updated as follows:

$$u_i(t) = \max\{u_i(t-), BU_{i,h}\} \quad (2.9)$$

With the above definition of the session utility, as given by Equations (2.7), (2.8) and (2.9), a newly backlogged flow can be treated as if it had been backlogged for the same length of time as all other flows. Therefore, the goal of the scheduler is simply to minimize the difference between the maximum and the minimum session utilities among all the sessions, i.e., minimize the spread of the set consisting of the session utilities of flows.

2.2.3 Choosing the Transmission Order

Let $\mathbf{U}(t)$ denote the set consisting of the session utility values of all the backlogged flows at time t in the GrFQ scheduler. That is,

$$\mathbf{U}(t) = \{u_i(t) \mid i \in \mathbf{B}(t)\},$$

where $\mathbf{B}(t)$ is the set of backlogged flows at time t .

The basic principle behind the GrFQ scheduler is to transmit the packet that, upon completion of its transmission, will lead to the smallest spread in the set of session utilities. At the boundaries of packet transmission, we will therefore need to compute the session utility of each flow if the packet at the head of its queue is chosen for transmission. Here we define the concept of the potential session utility of each session as follows:

Definition 3 *Let t be a time instant when the server completes a transmission. The potential session utility of a backlogged flow i at time t , $\hat{u}_i(t)$, is defined as the session utility of the flow upon completion of the transmission of the packet at the head of its queue at time*

t. Suppose the packet *h* is at the head of the queue of flow *i* at time *t*. Let $L_{i,h}$ be the length of packet *h*. Then,

$$\hat{u}_i(t) = u_i(t) + L_{i,h}/w_i$$

Denote by $\hat{\mathbf{U}}_i(t)$ the set of session utilities of all the flows after completion of the transmission that begins at time *t* of a packet from flow *i*. By our definition of session utilities, $\hat{\mathbf{U}}_i(t)$ is formed by merely replacing $u_i(t)$ in the set $\mathbf{U}(t)$ by $\hat{u}_i(t)$.

$$\hat{\mathbf{U}}_i(t) = \{\hat{u}_i(t)\} \cup \{u_j(t) \mid j \neq i \text{ and } j \in \mathbf{B}(t)\}$$

Upon completion of each packet transmission, the GrFQ scheduler will choose to transmit the next packet from the flow with the minimum value of $D(\hat{\mathbf{U}}_i(t))$ among all the backlogged flows.

2.3 Implementation of GrFQ

From the previous description of the GrFQ scheduler, a naive implementation would entail the computation of the spread corresponding to all different sets, $\hat{\mathbf{U}}_i(t)$, for each flow $i \in \mathbf{B}(t)$. This implies an $O(N)$ per-packet complexity in the worst case which is somewhat prohibitive. In this section, we discuss how we manage to accomplish the computations required to determine the next packet to transmit with a significantly lower complexity of $O(\log N)$.

Suppose the size of $\mathbf{B}(t)$ is *k*, i.e., we have *k* backlogged flows at time *t*. Let $u_{c_1}, u_{c_2}, \dots, u_{c_k}$ be the elements of the set $\mathbf{U}(t)$, such that $u_{c_1} \leq u_{c_2} \leq \dots \leq u_{c_k}$. The goal of the GrFQ scheduler is to choose the packet from the head of the queue of flow *f* such that $D(\hat{\mathbf{U}}_i(t))$ is the smallest for $i = f$. In other words, the goal of the GrFQ scheduler is to transmit from the flow that leads to the smallest spread in the set of session utilities at the next transmission boundary.

In our implementation, we maintain one binary search tree and one heap data structure:

- *Tree of Session Utilities (TSU)*, a binary search tree of the elements of the set $\mathbf{U}(t)$. This tree is updated at each packet transmission boundary.
- *Heap of Potential Session Utilities (HPSU)*, a minimum heap of all potential session utilities, $\hat{u}_i(t)$, $i \in \mathbf{B}(t)$. Let \hat{u}_m be the minimum among them.

In *TSU* and in *HPSU*, insertion and deletion operations are of complexity $O(\log N)$. Finding the maximum and the minimum in *TSU* is an $O(\log N)$ operation, while finding the minimum in *HPSU* is only an $O(1)$ operation. At each packet transmission boundary, since only one flow changes its session utility, the complexity of maintaining these trees is $O(\log N)$.

Now, the following theorem shows that not all the potential session utilities require to be examined in order to determine the flow from which the next packet needs to be transmitted. In fact, the theorem shows that the choice of a flow for the GrFQ scheduler is narrowed down to only one of two flows.

Theorem 1 *Let t be a time instant at a packet transmission boundary. Suppose that k flows are backlogged at time t and their session utilities form the set $\mathbf{U}(t)$. Let $u_{c_i}(t)$, $1 \leq i \leq k$, be the elements of $\mathbf{U}(t)$, such that $u_{c_1}(t) \leq u_{c_2}(t) \leq \dots \leq u_{c_k}(t)$. Let flow m has the minimum potential session utility at time t , i.e.*

$$\hat{u}_m(t) = \min_{1 \leq i \leq k} \hat{u}_{c_i}(t)$$

Consider a scheduler that follows the procedure below to determine the next packet to transmit:

- *If $\hat{u}_{c_1}(t) \leq u_{c_k}(t)$ or if $D(\hat{\mathbf{U}}_{c_1}(t)) \leq D(\hat{\mathbf{U}}_m(t))$, transmit packet from flow c_1 ; otherwise, transmit packet from flow m .*

This scheduler ensures that the next packet chosen for transmission is from the flow f such that $D(\hat{\mathbf{U}}_i(t))$ is the smallest for $i = f$.

Proof: We prove the theorem by considering two cases based on the relationship between $\hat{u}_{c_1}(t)$ and $u_{c_k}(t)$.

Case 1: $\hat{u}_{c_1}(t) \leq u_{c_k}(t)$. For all $i \neq c_1$, since $u_i(t) \geq u_{c_1}(t)$, we get,

$$D(\hat{\mathbf{U}}_i(t)) = \max\{u_{c_k}(t), \hat{u}_i(t)\} - u_{c_1}(t) \geq u_{c_k}(t) - u_{c_1}(t) \geq u_{c_k}(t) - u_{c_2}(t) = D(\hat{\mathbf{U}}_{c_1}(t)).$$

Therefore, choosing a packet from flow c_1 results in the minimal spread in the session utilities after completion of the transmission of the chosen packet.

Case 2: $\hat{u}_{c_1}(t) > u_{c_k}(t)$. For all $i \neq c_1$, we have $u_i(t) \geq u_{c_1}(t)$; and for all $i \neq m$, we have $\hat{u}_i(t) \geq \hat{u}_m$. Therefore, for $i \neq c_1$, we get,

$$D(\hat{\mathbf{U}}_i(t)) = \max\{u_{c_k}(t), \hat{u}_i(t)\} - u_{c_1}(t) \geq \max\{u_{c_k}(t), \hat{u}_m(t)\} - u_{c_1}(t) = D(\hat{\mathbf{U}}_m(t)).$$

Therefore, transmitting a packet from any flow other than c_1 leads to at least as large a spread at the next packet boundary as one would get with transmitting a packet from flow m . Therefore, the packet chosen to transmit should be either from c_1 or from m if the spread at the next packet boundary is to be minimized. Which of these two flows yields the smaller spread is simply determined by comparing $D(\hat{\mathbf{U}}_{c_1}(t))$ and $D(\hat{\mathbf{U}}_m(t))$ as in the statement of the theorem. ■

Figure 2.1 and 2.2 present the pseudo-code of the GrFQ scheduler. Note that the *Dequeue* routine uses the simplification due to Theorem 1 above.

2.3.1 Fairness Analysis

In the following, we use weights instead of reserved rates. For purposes of fairness analysis, the two are equivalent. If the sum of the reserved rates on a link is less than its capacity R , then one may assign weights directly proportional to the reserved rates.

Theorem 2 *Given a GrFQ scheduler, during any given interval (t_1, t_2) in which flows are continuously backlogged,*

$$\left| \frac{S_i(t_1, t_2)}{w_i} - \frac{S_j(t_1, t_2)}{w_j} \right| \leq 2 \max \left(\frac{L_{i,\max}}{w_i}, \frac{L_{j,\max}}{w_j} \right).$$

```

Initialize: (Invoked when the scheduler is initialized)
1  TSU ← NULL;
2  HPSU ← NULL;

Enqueue: (Invoked when a packet P arrives at flow i)
3  if (Queue i is empty) then
4       $u_i \leftarrow \max\{u_i, v(t)\}$ ; /* See Eq. (2.8)*/
5       $\hat{u}_i \leftarrow u_i + P.Length / w_i$ ;
6      AddFlowToTSU(i);
7      AddFlowToHPSU(i);
8  end if;
9   $P.BU \leftarrow \max\{FU_i(t), v(t)\}$ ;
10  $FU_i(t) \leftarrow P.BU + P.Length / w_i$ ;
11 AddPacketToQueue(i, P);

```

Figure 2.1: Pseudo-code of *Initialize* and *Enqueue* routines in GrFQ scheduler

for any two flows i and j .

Proof: Without loss of generality, we assume that both flows are backlogged since time 0. We prove the theorem by showing that, for any given instant of time t ,

$$\left| \frac{S_i(0, t)}{w_i} - \frac{S_j(0, t)}{w_j} \right| \leq \max \left(\frac{L_{i, \max}}{w_i}, \frac{L_{j, \max}}{w_j} \right) \quad (2.10)$$

Since the maximum difference in service occurs immediately after a transmission from flow i or flow j , we will consider only these instants of time. Consider one of these instants of time, τ_e . Without loss of generality, assume that it is flow i that completes its transmission at time τ_e . Let τ_b be the instant of time that flow i begins this transmission.

Case 1: At time τ_b , assume flow j has received less normalized service than flow i . Now, from Theorem 1, the GrFQ scheduler only chooses either the flow with the smallest session utility or the flow with the smallest potential session utility. Given that flow i is chosen for transmission at time τ_b and flow i is not the flow with the smallest session utility, it must be the flow with the smallest potential session utility. Thus, since flow j 's

```

Dequeue:
12 while (At least one queue is not empty) do
13    $c_1 \leftarrow \text{TreeMinimumOfTSU};$ 
14    $c_k \leftarrow \text{TreeMaximumOfTSU};$ 
15   if ( $\hat{u}_{c_1} \leq u_{c_k}$ ) then  $P \leftarrow \text{HeadPacketOfQueue}(c_1);$ 
16   else
17      $c_2 \leftarrow \text{TreeSuccessorInTSU}(c_1);$  3
18      $m \leftarrow \text{MinimumOfHPSU};$ 
19      $D_{c_1} \leftarrow \hat{u}_{c_1} - u_{c_2};$ 
20      $D_m \leftarrow \max\{u_{c_k}, \hat{u}_m\} - u_{c_1};$ 
21     if ( $D_{c_1} \leq D_m$ ) then  $P \leftarrow \text{HeadPacketOfQueue}(c_1);$ 
22     else  $P \leftarrow \text{HeadPacketOfQueue}(m);$  end if;
23   end if;
24    $\text{TransmitPacket}(P);$ 
25   Update TSU and HPSU; /* See Eq. (2.9)*/
26 end while;

```

Figure 2.2: Pseudo-code of *Dequeue* routine in GrFQ scheduler

potential utility is larger, after the transmission from flow i , $L_{j,max}/w_j$ must be larger than the difference between $S_i(0, \tau_e)/w_i$ and $S_j(0, \tau_e)/w_j$.

Case 2: At time τ_b , assume flow j has the same or more normalized service than flow i . We will prove this case by induction on time instants under consideration. Assume that (2.10) holds true for all time instants $t \leq \tau_b$. Now, after completion of transmission from flow i at time τ_e , flow i will not be ahead of flow j in the normalized service it receives by more than $L_{i,max}/w_i$ since one transmission from flow i cannot cause more than this difference beginning with a lower normalized service than flow j . Therefore, (2.10) holds at time τ_e .

The above proves (2.10) for $t = \tau_e$ under the assumption that it holds for $t \leq \tau_b$. Since the difference in normalized services is 0 at time 0, (2.10) also holds for any time $t > 0$.

Since $S_i(t_1, t_2) = S_i(0, t_2) - S_i(0, t_1)$, the theorem is proved by substituting t by t_1 and t_2 in (2.10) and combining the resulting equations. ■

Corollary 1 *The Relative Fairness Bound of the GrFQ scheduler is given by,*

$$\text{RFB} = \frac{2L_{\max}}{w_{\min}} \quad (2.11)$$

where L_{\max} is the size of the largest packet in the system and w_{\min} is the minimum of the flow weights.

Proof: The proof follows readily from Theorem 2. ■

It may also be noted that the relative fairness measure of GrFQ as expressed in Theorem 2 is further from the optimal than that achieved by some schedulers such as SCFQ. This is not entirely surprising since the GrFQ scheduler is a greedy algorithm and is not necessarily guaranteed to achieve the most optimal final result. However, as shown by the analysis in this section, the GrFQ scheduler achieves a value of the RFB that is extremely close to that of SCFQ. Also, as we will demonstrate in our section on simulation results, the GrFQ scheduler achieves better overall fairness characteristics than any other scheduler, including those with significantly higher computational complexity.

Corollary 2 *If all flows become backlogged at time 0 and remain continuously backlogged until time t ,*

$$\left| \frac{S_i(0, t)}{w_i} - \frac{G_i(0, t)}{w_i} \right| \leq \left(1 - \frac{w_{\min}}{W}\right) \frac{L_{\max}}{w_{\min}}, \quad (2.12)$$

where $S_i(0, t)$ is the amount of service received by flow i from a GrFQ server and $G_i(0, t)$ is the amount of service the flow would have received from the ideally fair fluid flow GPS server.

³The successor of an element c_1 in a binary search tree is the element with the smallest key value greater than that of c_1 .

Proof: Using the relationship proved in [42], we have,

$$\left| \frac{S_i(0, t)}{w_i} - \frac{G_i(0, t)}{w_i} \right| \leq \left(1 - \frac{w_{\min}}{W}\right) \text{RF}(0, t) \quad (2.13)$$

According to Theorem 2, $\text{RF}(0, t) \leq L_{\max}/w_{\min}$. Therefore, the statement of the corollary follows readily from Theorem 2. \blacksquare

According to [23], the service difference between WF^2Q and GPS satisfies

$$|S_i^{\text{WF}^2\text{Q}}(0, t) - G_i(0, t)| \leq L_{\max}$$

Therefore for the WF^2Q scheduler, we have

$$\left| \frac{S_i^{\text{WF}^2\text{Q}}(0, t)}{w_i} - \frac{G_i(0, t)}{w_i} \right| \leq \frac{L_{\max}}{w_{\min}} \quad (2.14)$$

Comparing 2.12 and 2.14, we can see that the absolute fairness measure of GrFQ is better than that of WF^2Q . The WF^2Q scheduler is commonly believed to be the most fair scheduler [18] because it achieves the best possible lag in comparison to the ideally fair GPS scheduler. However, as seen from the above corollary, the GrFQ scheduler achieves a better *normalized* lag than the WF^2Q scheduler.

2.3.2 Computational Efficiency

Theorem 3 *The GrFQ scheduler has a per-packet work complexity of $O(\log N)$ with respect to the number of flows.*

Proof: The per-packet work complexity of GrFQ is determined by the per-packet work complexity of its *Enqueue* and *Dequeue* routines shown in Figure 2.1 and 2.2.

Enqueuing a packet entails adding the packet to the tail of its flow's queue if the queue is not empty, requiring only $O(1)$ time. However, if the corresponding queue is empty when the packet arrives, the *Enqueue* routine needs to initialize the corresponding session utility, and then insert the flow into the data structures *TSU* and *HPSU*. The insertion procedures require $O(\log N)$ time with respect to the number of flows.

To dequeue a packet, the scheduler needs to compare the spread of potential session utilities of two flows. Computing the spread requires the utilities of at most four flows, as stated by Theorem 1. The determination of these flows in the *TSU* and the *HPSU* data structure takes $O(\log N)$ and $O(1)$ time, respectively. Also, upon transmission of the packet, the scheduler updates the data structures *TSU* and *HPSU*. Since both insertion and deletion operations in these data structures are of $O(\log N)$ complexity, the overall time taken by the *Dequeue* routine is also $O(\log N)$. ■

2.4 GrFQ-lite

As mentioned earlier, there are two components to a timestamp-based scheduler: the method by which the timestamps are computed and the method by which the transmission order is determined. The computation of the timestamp in the GrFQ scheduler involves the computation of the virtual time, $v(t)$, based on an emulation of the ideally fair GPS scheduler. Certain other schedulers such as WFQ and WF²Q also involve the computation of the virtual time based on an emulation of the GPS scheduler. This is a computationally intensive task and is of work complexity $O(N)$ with respect to the number of flows. Therefore, various computationally efficient methods of system tracking and timestamp computations have been proposed for practical implementations, including SCFQ, SPFQ and TSS.

In SCFQ, the system status is tracked through *self-clocking*; when the system is busy, the timestamp is always equal to the finishing time of current packet in transmission. Since the service order is determined by the policy of smallest finishing-time first, the assigned timestamps are monotonically increasing functions of time. In such a self-clocked system, a flow which becomes backlogged at the beginning of a transmission will bear a starting credit that is the same as the one which comes later but before the completion of the transmission. This punishes the packet of a flow that arrives early and this error in the time-stamps of the earlier flow is always maintained leading to biased scheduling decisions

thereafter.

SPFQ and TSS solve this problem by including the time factor between two arrivals. The system state is based on the starting-time of the packet in transmission plus the time elapsed since the beginning of the transmission. Upon finishing each transmission, the tracking mechanism will perform a re-calibration and set to the starting-time of the next packet for transmission. This method is closer to the GPS system than the self-clocked system, since a flow that arrives earlier will have an earlier starting point. However, deviations still exist from the ideal GPS system. During each transmission, the system clock increases as if all flows are backlogged. However, in a real system, flows are not always backlogged. In these schedulers, the system clock is always tracking the lower bound of the real system value. If a flow uses the tracked system state as the reference to set its starting point, an error is introduced. The largest error occurs at the instant just before a transmission finishes.

In spite of the deviations from the GPS in the tracking mechanisms of SCFQ, SPFQ and TSS, their practical value cannot be overlooked since the processing time is only of $O(1)$. These tracking methods can all be used along with GrFQ's method of determining the transmission order given the timestamps. However, the result with such a combination of the system tracking method and GrFQ method of determining the transmission order is sub-optimal since the above-mentioned errors in the timestamp computation will mislead the scheduler away from the right decision. In this section, we introduce a novel technique of tracking system state that also incurs a complexity of only $O(1)$. In combination with the GrFQ method of determining the transmission order, this computationally efficient scheduler, called *GrFQ-lite* achieves better fairness characteristics than the other schedulers of equivalent complexity in system tracking.

Consider a case in which N flows have been continuously backlogged during the interval $(0, t)$. The total service received by all the flows using scheduling policy P is the summation of $S_i^P(0, t)$ for $1 \leq i \leq N$. Assuming a work-conserving scheduler, this is also

the total service received by all the flows if the server is the ideally fair GPS scheduler. With the ideally fair GPS scheduler, a flow i with weight w_i receives service equal to

$$\frac{w_i}{W(t)} \sum_i S_i^P(0, t) \quad (2.15)$$

Thus, at time t , the above indicates the ideal amount of service flow i should have received.

The above leads us to the definitions of the nominal system service and the nominal system time, approximations of the accurate values in the ideally fair GPS system. Suppose that, starting from time t_0 until time t_1 , $\mathbf{B}(t)$ consists of the same set of flows. At time t ($t_0 \leq t < t_1$), the nominal system service, or $S^*(t)$, is defined as the total service provided by the system from t_0 to t , i.e.,

$$S^*(t) = \sum_{i \in \mathbf{B}(t)} S_i^P(t_0, t)$$

The nominal system time, $v^*(t)$, is computed from $S^*(t)$ as

$$v^*(t) = \frac{S^*(t)}{W(t)} \quad (2.16)$$

At time t_1 , a flow i becomes empty in the real system. The nominal system service will adjust its value by subtracting an amount of service which flow i would receive in the GPS system up to t_1 , i.e.,

$$S^*(t_1) = S^*(t_1-) - w_i v^*(t_1-)$$

Since w_i is excluded from $W(t)$, the nominal system time at t_1 is the same as $v^*(t_1-)$. However, the ticking rate of nominal system clock will accordingly change to $1/W(t_1)$. If a flow j becomes backlogged at time t_2 , the nominal system service would adjust its value as

$$S^*(t_2) = S^*(t_2-) + w_j v^*(t_2-)$$

The nominal system time also adjusts its rate by including w_j into $W(t_2)$.

Therefore, the nominal system time is a monotonically increasing function of time and is also a continuous function of time. The computation complexity of $v^*(t)$ is of $O(1)$. To

utilize the nominal system time, one only needs to replace GPS virtual time $v(t)$ by the nominal system time $v^*(t)$. GrFQ-lite is the same scheduler as GrFQ, except that $v(t)$ is replaced by $v^*(t)$ in all computations.

2.5 Evaluation of GrFQ and GrFQ-lite

2.5.1 A Measure of Instantaneous Fairness

To evaluate the performances of GrFQ and GrFQ-lite, we use a new measure of instantaneous fairness [46, 47] based on an adaptation of the Gini index used widely in the field of economics. This metric is capable of measuring the fairness at any instant of time during the execution of a scheduler. In earlier sections, we have shown that the relative fairness bound (RFB) of the GrFQ scheduler is better than or as good as the relative fairness bound of any other known packet-by-packet scheduler. However, as mentioned in Section 2.1, the RFB is just a bound and does not quite capture the overall quality of a fair scheduler. This is because a scheduler that rarely reaches the upper bound of relative fairness will achieve the same measure of fairness as another scheduler that frequently or almost always operates at the same upper bound. Therefore, we also need an *instantaneous* measure of fairness that captures the fairness achieved by the scheduler at any given instant of time.

In addition, fairness measures based on upper bounds on the spread in the session utilities also do not inform us of how a scheduler treats packets of one flow in comparison to those of another. Fairness, after all, is expected to be based on a comparison among the levels of service received by all the flows and not merely on the maximum difference in the normalized service received by flows. Figures 2.3(a) and (b) illustrate an example where the bars represent the service received by flows under two different schedulers A and B, during a certain interval of time in which all flows are backlogged. Assuming the weights associated with the flows are identical, the dashed line represents the level of service each of the flows would receive under the ideally fair GPS scheduler. One may observe from the

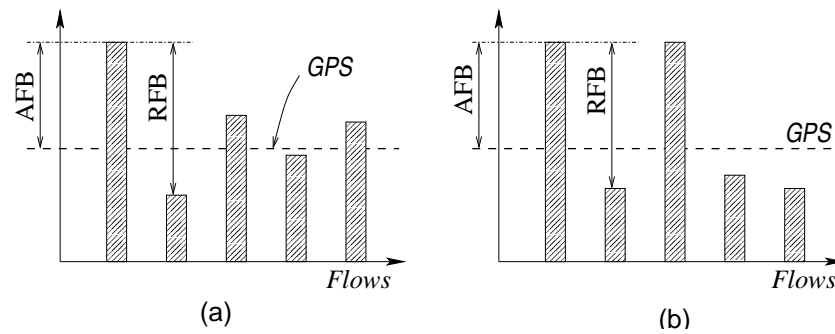


Figure 2.3: An illustration of the difference in the disparity in service received while the upper bounds of the relative and absolute fairness measures are identical in two packet scheduling systems

figures that scheduler B leads to a greater disparity in the levels of service received by the flows since scheduler A allows more flows to achieve service close to the ideally fair level. If the absolute and relative fairness bounds are exactly reached in this interval of time, note that both schedulers would lead to the same values for the RFB and the AFB, even though, the levels of service received by flows under scheduler A are closer to each other than with scheduler B.

Thus, measures of fairness based on an upper bound serve the excellent purpose of capturing the fairness characteristics of a scheduler in a single number. However, they do not capture the overall behavior of the scheduler at all instants of time and also do not quite capture the characteristics of the distribution of the service among all the flows (the AFB only reports the maximum deviation from GPS for any flow while the RFB reports the maximum difference in service received by two different flows, but neither capture the overall fairness of the allocation among all the flows.) This is addressed by the measure of *instantaneous* fairness described below based on measures of inequality used in the field of economics.

Various measures of inequality have been used in the field of economics for several

decades in the study of social wealth distribution and many other economic issues of interest [47]. Some of these methods are related to the theory of majorization used in mathematics as a measure of inequality [48]. This theory has occasionally found use in research in computer networks in the fairness analysis of protocols [49]. A popular, mature and one of the least ambiguous measures of inequality developed in the field of economics is that based on the concept of the *Lorenz curve* and *Gini index* [47]. Since fairness is often believed to be a concept based on equality of treatment to all competitors with equal rights to a resource (though not necessarily equal allocation to all competitors), one can borrow from the field of economics and use these concepts to judge the fairness of our schedulers at each instant of time.

Consider the problem of measuring the inequality among k quantities, $g_1 \leq g_2 \leq \dots \leq g_k$. Define $d_0 = 0$, and $d_i = d_{i-1} + g_i$, for $1 \leq i \leq k$. Now, a plot of d_i against i is a concave curve, known as the *Lorenz curve* [50], as shown in Figure 2.4(a). Note that if there is perfect equality in these k quantities, the Lorenz curve will be a straight line starting from the origin. The Gini index measures the area between the Lorenz curve and this straight line, and thus measures the inequality amongst the k quantities [47]. Appendix A provides a detailed discussion on different computational methods of the Gini index and a comparison of the Gini index with other common measures of inequality.

In our case, we wish to measure the inequality in the session utility of the backlogged flows at any given instant of time. The Gini index in our case is the area between the Lorenz curve of the actual session utilities and the Lorenz curve corresponding to the ideally fair GPS scheduler. A discussion of the difference between the Gini index for packet schedulers and for economic studies is provided in Appendix A.

When the sum of the k quantities is the same as the sum in the case of perfect equality, the Lorenz curve always lies below the straight line corresponding to the Lorenz curve of the ideal equal case, as shown in Figure 2.4(a). However, the sum of the session utilities is almost never exactly identical to the sum of the session utilities in the ideally fair GPS

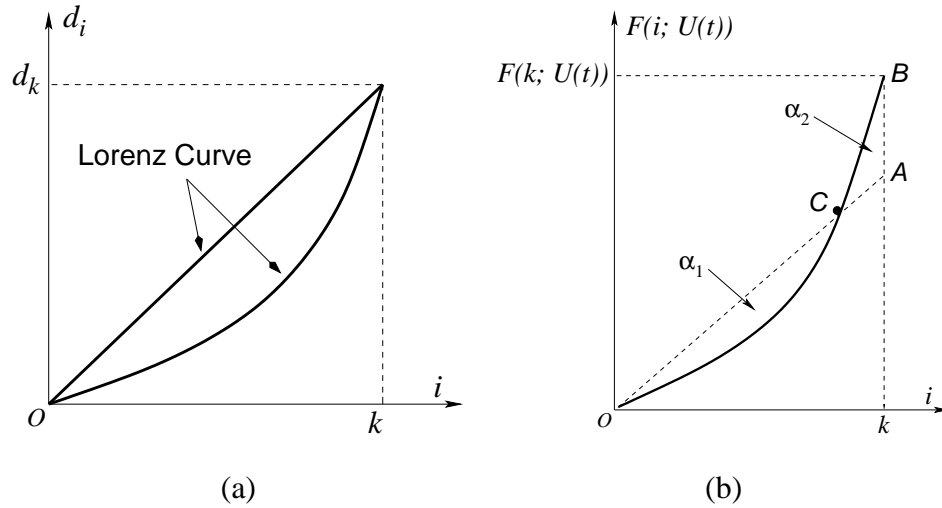


Figure 2.4: An illustration of the Lorenz curve and Gini index

case. Note that, in a work-conserving scheduler, only the sum of the total service delivered is identical to that in the ideally fair GPS scheduler. The sum of the session utilities which represents the sum of the *normalized* service delivered is not identical to that in the GPS system. In a work-conserving scheduler, when the sum of the k quantities is not the same as the sum in the case of perfect equality as with the GPS scheduler, a portion of the Lorenz curve for the actual scheduler will lie below and another portion will lie above the straight line Lorenz curve for the GPS scheduler. This is illustrated in Figure 2.4(b). The sum of the areas marked by α_1 and α_2 in the figure is the Gini index.

We describe the computation of the Gini index formally as follows:

Definition 4 Let $\mathbf{U}(t)$ denote the set of session utilities obtained with a real scheduler at time t , and let $\mathbf{G}(t)$ denote the same with the GPS scheduler. Let $u_{c_1}, u_{c_2}, \dots, u_{c_k}$ be the elements of the set $\mathbf{U}(t)$, such that $u_{c_1} \leq u_{c_2} \leq \dots \leq u_{c_k}$. The Lorenz Curve of the set of session utilities $\mathbf{U}(t)$ is the function $F(i; \mathbf{U}(t))$, given by,

$$F(i; \mathbf{U}(t)) = \sum_{j=1}^i u_{c_j}, 0 \leq i \leq k$$

The Gini index over the k elements in $\mathbf{U}(t)$ is computed as:

$$\sum_{i=1}^k |F(i; \mathbf{U}(t)) - F(i; \mathbf{G}(t))| \quad (2.17)$$

2.5.2 Evaluation through Simulations

We construct experiments with real traffic traces of various characteristics. In this study, we have simulated the following schedulers:

- WFQ [16, 20], for obvious historical reasons.
- WF²Q [23], because it achieves the best possible lag in comparison to the ideally fair fluid flow GPS scheduler, and has therefore been sometimes believed to be the most fair packet scheduler. (As discussed earlier, the GrFQ scheduler achieves a better lag if normalized by the weight of the flow. However, we do find that WF²Q is the scheduler that comes closest in fairness to our GrFQ scheduler.)
- Time-Shift Scheduling (TSS) [41], for its apparent closeness to the packet-by-packet actions of GrFQ.
- Self-Clocked Fair Queueing (SCFQ) [21], as a representative scheduler among those with the best relative fairness bound.
- Starting Potential Fair Queueing (SPFQ) [40], a representative scheduler based on the concept of rate-proportional servers introduced in [44].

2.5.3 Results with Video Traffic Traces

In this set of experiments, each flow is from an MPEG-4 video trace. We use video traces collected from the Telecommunication Networks Group at Technical University of Berlin, Germany [51]. The traces used in our study are coded using MPEG-4 of different qualities. For each input, one distinct video stream is used, and the starting point within the

Table 2.1: Settings for MPEG-4 traffic sources

Source	1	2	3	4
Movie Name ⁴	J	S	W	B
Video Quality	High	High	Medium	Low
L_{\min} (bytes)	72	158	26	26
L_{\max} (bytes)	16,745	22,239	4,690	7,565
r_{avg} (Kbps)	770	580	78	110
r_{peak} (Mbps)	3.3	4.4	0.94	1.5
Weight (w_i)	9.88	7.44	1	1.41
Link Capacity	1.54 Mbps			
Total Time	112 seconds			

video stream is randomly selected. Table 2.1 shows the settings for this set of input traffic. The flow weights are set based on average rate of each flow. Here we set the weight of the slowest flow as 1, and weights of other flows are equal to the ratios of their average rate to the smallest rate.

The GrFQ scheduler

In our comparisons with the GrFQ scheduler, we assume continuously backlogged queues since, with accurate tracking of the GPS scheduler, it does not matter whether or not the queues are continuously backlogged. The traffic used to feed the backlogged queues is generated by extracting information on packet lengths from MPEG-4 traces. Figure 2.5(a) plots the average length of packets in transmission during each periodic interval of 560 ms. It provides a rough idea of the changes in packet lengths with time. Throughout the simulation period, the average packet length ranges from 1200 bytes to 3200 bytes.

Figures 2.5(b-c) plot the performances of GrFQ, WF²Q and WFQ. Since queues are backlogged all the time, variations in the packet lengths are the only reason for unfair

⁴The alphabet letters, J, S, W and B, stand for movies “Jurassic Park I”, “Silence Of the Lambs”, “Star Wars IV”, and “Mr. Bean”, respectively.

transmission order. Therefore, the performances of TSS, SCFQ and SPFQ are very close to that of WFQ since, even though their method of timestamp computations is different, the timestamps of arriving packets are very close in the case of each of these schedulers. Timestamp values differ significantly only when flows go back and forth between the backlogged state and the unbacklogged state. Therefore, for reasons of brevity, Figure 2.5 plots only the comparisons between GrFQ, WFQ and WF²Q.

As defined in Equation (2.17), the lower the Gini index, the more fair the algorithm. Figure 2.5 shows that the GrFQ scheduler displays better fairness than WFQ and WF²Q. The fairness achieved by WF²Q is very close to that of GrFQ while WFQ clearly has a worse fairness performance.

The GrFQ-lite scheduler

In our comparisons with the GrFQ-lite scheduler, we assume that flows will go back and forth between a backlogged state and a non-backlogged state. Since the effectiveness of a tracking scheme is more clearly exhibited when the link is close to fully utilized, we set the link capacity such that the sum of average rates of all the flows is more than 98% of the link capacity.

Figure 2.6 shows the values of the Gini index during the simulations of six different schedulers including the GrFQ scheduler. For clarity in comparison, we plot each scheduler's Gini index distribution on a separate graph, with the GrFQ-lite scheduler plotted on each of the graphs.

Figure 2.6(a) shows the average length of arriving packets among all sessions during the simulation interval. Since MPEG-4 streams generate periodic traffic, the traffic load is directly proportional to packet lengths. Therefore, from this figure, we observe 3 large bursts starting around time instants 32, 60, and 92 seconds. Figures 2.6(b–f) show the GrFQ scheduler's fairness in comparison to WF²Q, WFQ, SPFQ, TSS and SCFQ, respectively.

Even though WF²Q assumes accurate GPS tracking and GrFQ-lite achieves an approximation, as is readily observed, GrFQ-lite comes very close in fairness to the WF²Q scheduler.

Amongst SCFQ, TSS, and SPFQ, SPFQ and TSS achieve fairness closer to the GrFQ scheduler. SCFQ is more unfair than SPFQ and TSS during the bursts. As discussed before, this is because SCFQ uses the finishing time stamp of the packet currently being served as the start time for newly arrived packets in all sessions. This introduces a discrepancy in the computation of the timestamp in the flows other than the one currently being served. This discrepancy does not get corrected at a later time, leading to a less fair allocation.

SPFQ and TSS solve the above problem by using the starting value corresponding to packet currently being served. One can also see from Figure 2.6(d) and (e) that the fairness achieved by these two schedulers is similar.

In general, the fact that the GrFQ-lite scheduler achieves better fairness most of the time illustrates that the tracking scheme of system time proposed here, as given by (2.16), is a good approximation to the GPS scheduler.

Results with Different Traffic Loads

In these experiments, we increase the capacity of the transmission link to compare the performance of schedulers under different traffic load situations. We simulate two cases: one with 90% traffic load; the other with 80% traffic load. The incoming traffic is still from real video traces as listed in Table 2.1.

Figure 2.7 and 2.8 show the Gini index of GrFQ, GrFQ-lite, WF²Q, WFQ, SPFQ, TSS and SCFQ with 90% and 80% traffic load, respectively. Since we are more interested in the performance of the practical design, we plot the Gini index of GrFQ-lite in each graph to compare with other schedulers including GrFQ. From these graphs, one can observe that GrFQ-lite maintains a better performance than all other schedulers most of time. For some

Table 2.2: Settings for traffic sources from gateway traces

Source	1	2	3	4	5	6
Router Name ⁵	ADV	ANL	APN	BUF	MEM	TXS
Interface	OC3	OC3	OC3	OC3	OC3	OC3
L_{\min} (bytes)	38	28	29	32	32	28
L_{\max} (bytes)	4470	9180	1500	1560	4470	9180
r_{avg} (10^6 Bps)	0.56	0.63	1.4	1.45	0.37	2.1
Weight (w_i)	1.5	1.6	3.6	3.8	1	5.8
Link Capacity	6.4×10^6 Bps					
Total Time	50 seconds					

intervals, GrFQ-lite performs even better than GrFQ with this set of incoming traffic.

2.5.4 Results with Gateway Traffic Traces

Here we repeat the experiments in Section 2.5.3 with real gateway traffic traces. As opposed to video streams, packet lengths in router traffic are more uniformly distributed while the time interval between two packet arrivals may be random. These properties result in different performance from schedulers.

The traces used in this set of simulations are provided online by National Laboratory for Applied Network Research [52]. Now each input is fed by a gateway traffic trace with a random starting time. The settings of this set of experiments is listed in Table 2.2. Similar to the previous set of experiments, the link is close to fully utilized. The sum of average rates of all flows is about 99% of the link capacity. Flow weights are selected based on the observed average rates of the traces used in experiments. The computation method for Gini index is same as in previous experiments.

⁵The long names of routers are: Argonne National Laboratory(ANL), APAN(APN), University of Buffalo(BUF), University of Memphis(MEM), and Rice University(TXS).

The GrFQ scheduler

As before, we assume continuously backlogged queues. Figure 2.9(a) plots the average length of packets in transmission during each period of 250 ms. Figures 2.9(b-c) show the value of Gini index of GrFQ, WF²Q and WFQ. The relative performances of WF²Q and WFQ versus GrFQ are similar to the results under video traffic, as shown in Figure 2.6. Again, the GrFQ scheduler achieves the best fairness among these schedulers.

The GrFQ-lite scheduler

In this case, as before, we assume that flows will go back and forth between a backlogged state and a non-backlogged state. Figure 2.10(a) shows the traffic load throughout the duration of the simulation. There are 3 overloaded periods around time intervals 5–10, 20–30 and 35–45 seconds. Among the six schedulers under simulation, GrFQ-lite and WF²Q both have excellent fairness performance (shown in Figure 2.10(b)), even though GrFQ-lite is using $O(1)$ service tracking system while WF²Q uses the $O(N)$ service tracking system.

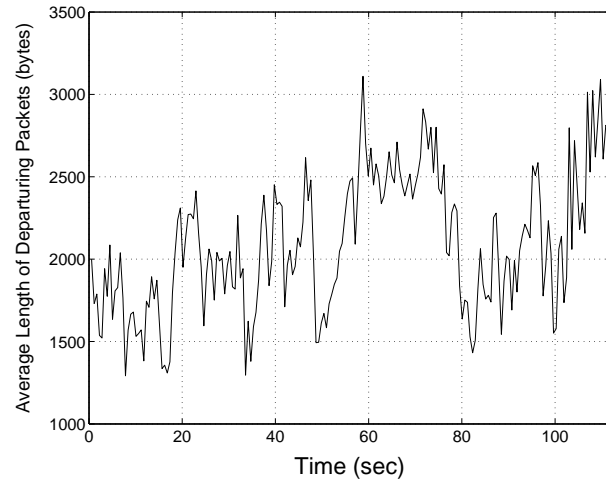
Figures 2.10(c-f) show the fairness performance of WFQ, SPFQ, TSS, and SCFQ as compared to GrFQ-lite. SPFQ, TSS and SCFQ have similar fairness performance, as shown in Figures 2.10(d-f). However, except for a few short unfair intervals, SPFQ and TSS still exhibit slightly better fairness than SCFQ.

Results with Different Traffic Loads

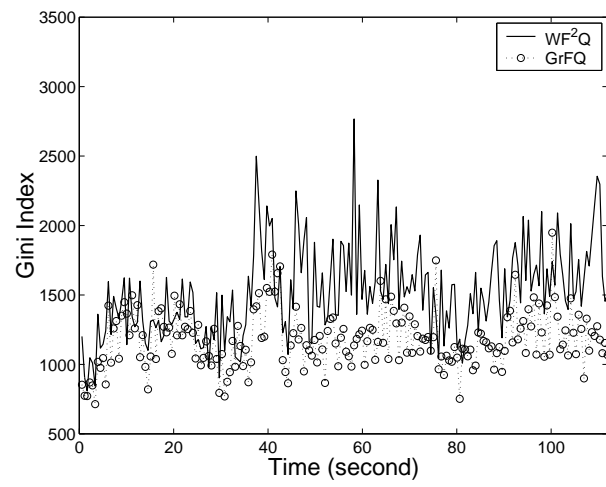
Here we compare the performance of schedulers under 90% traffic load and 80% traffic load. The incoming traffic is from real gateway traces as listed in Table 2.2.

Figure 2.11 and 2.12 show the Gini index of GrFQ, GrFQ-lite, WF²Q, WFQ, SPFQ, TSS and SCFQ with 90% and 80% traffic load, respectively. From Figure 2.12 one can observe GrFQ, GrFQ-lite and WF²Q maintain better fairness than the rest of the schedulers

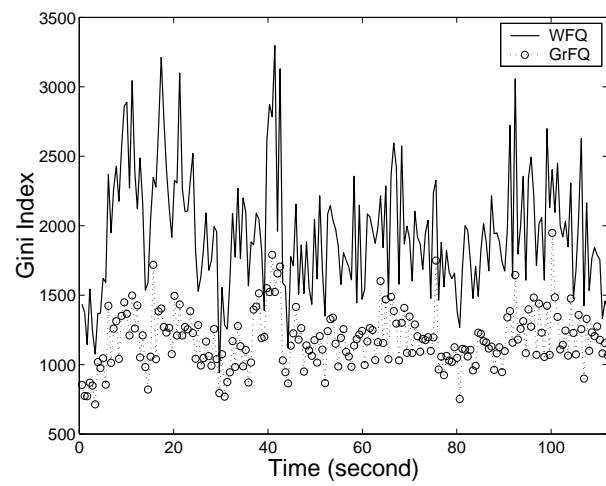
and they also achieve similar performance. Thus GrFQ-lite bears significant advantages of achieving good performance with very low processing complexity.



(a)



(b)



(c)

Figure 2.5: Gini indices of fair schedulers on backlogged queues with real video traffic traces

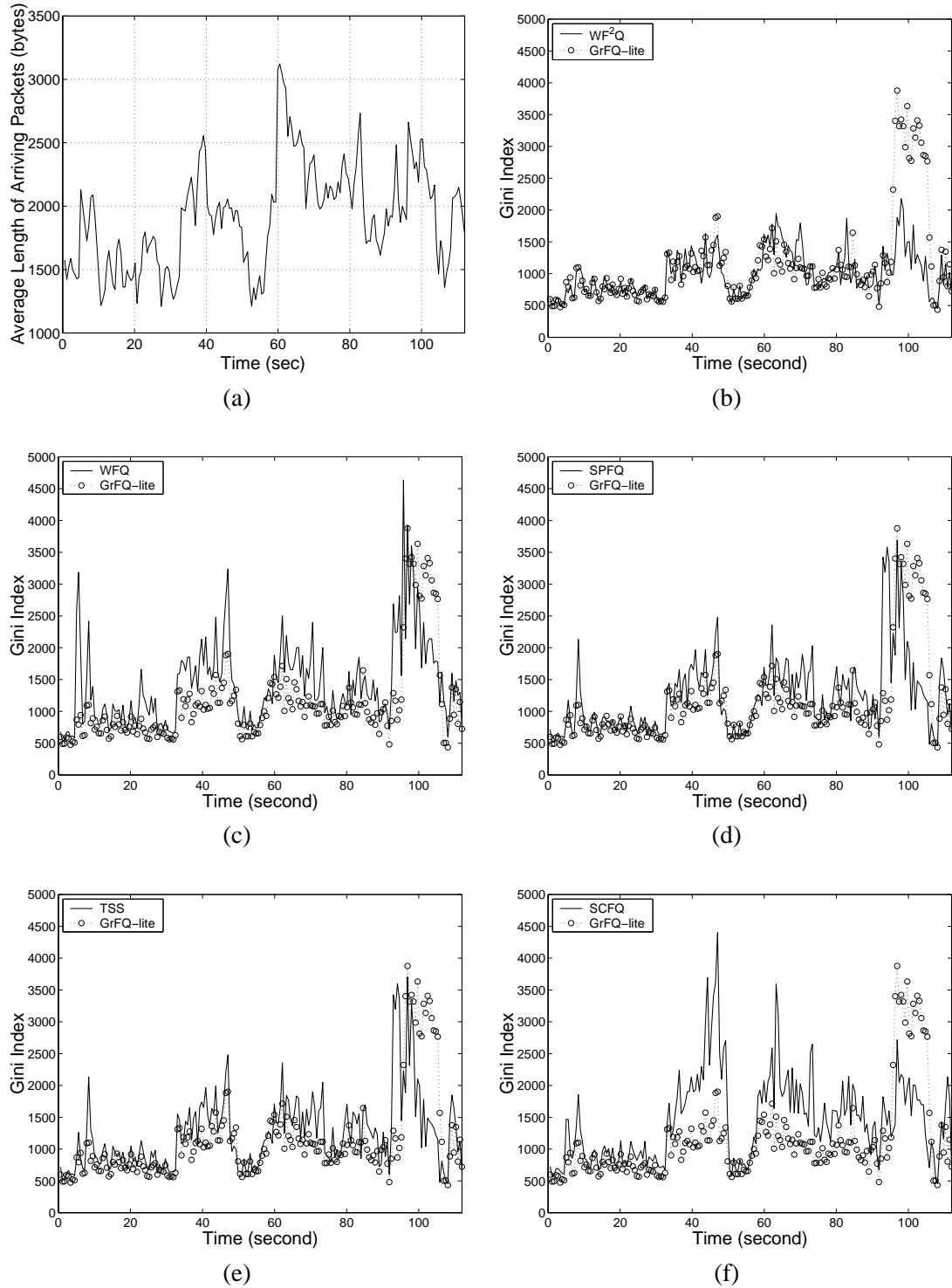


Figure 2.6: Gini indices of fair schedulers with real video traffic traces

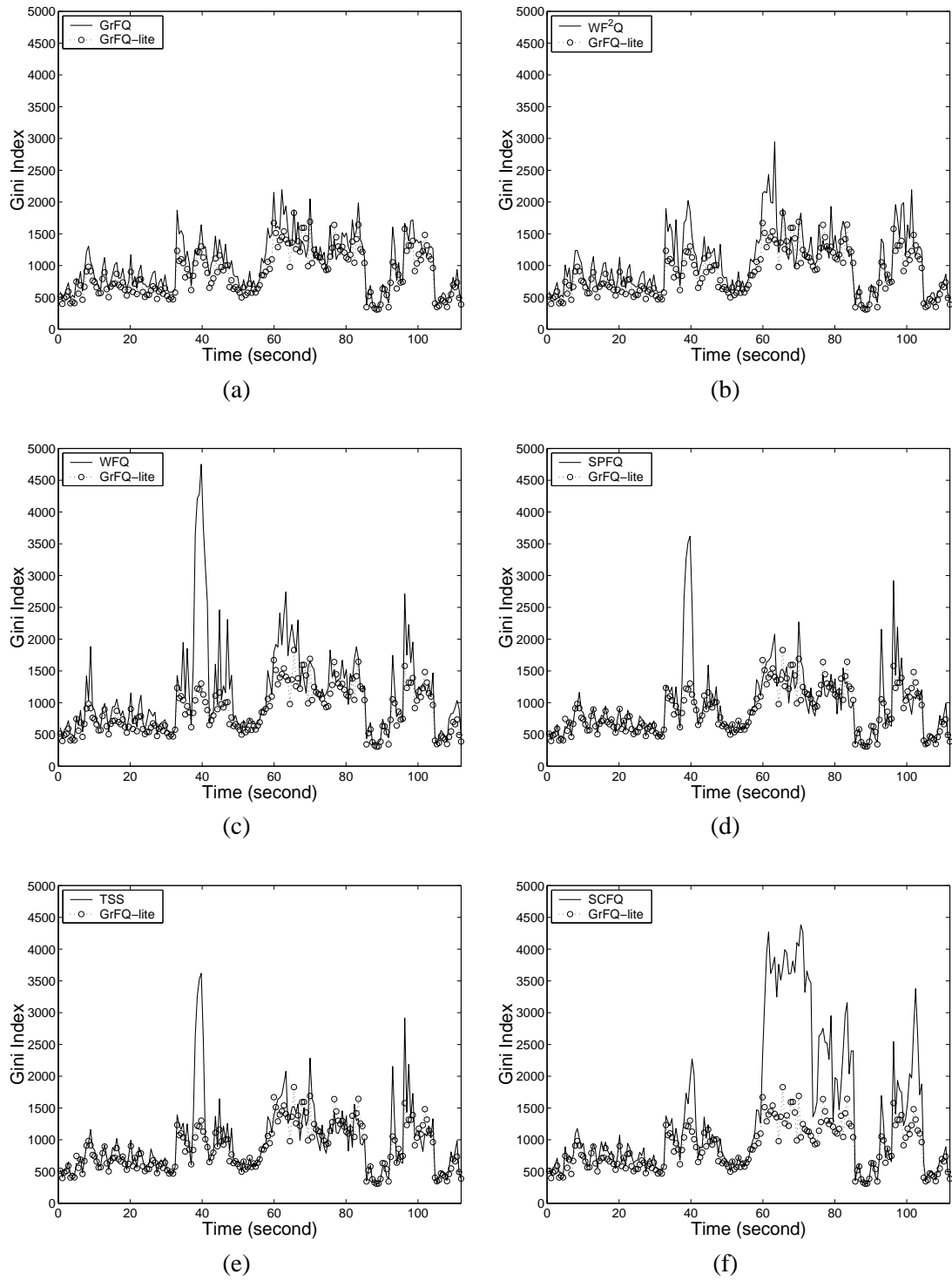


Figure 2.7: Gini indices of fair schedulers with real video traffic at 90% load

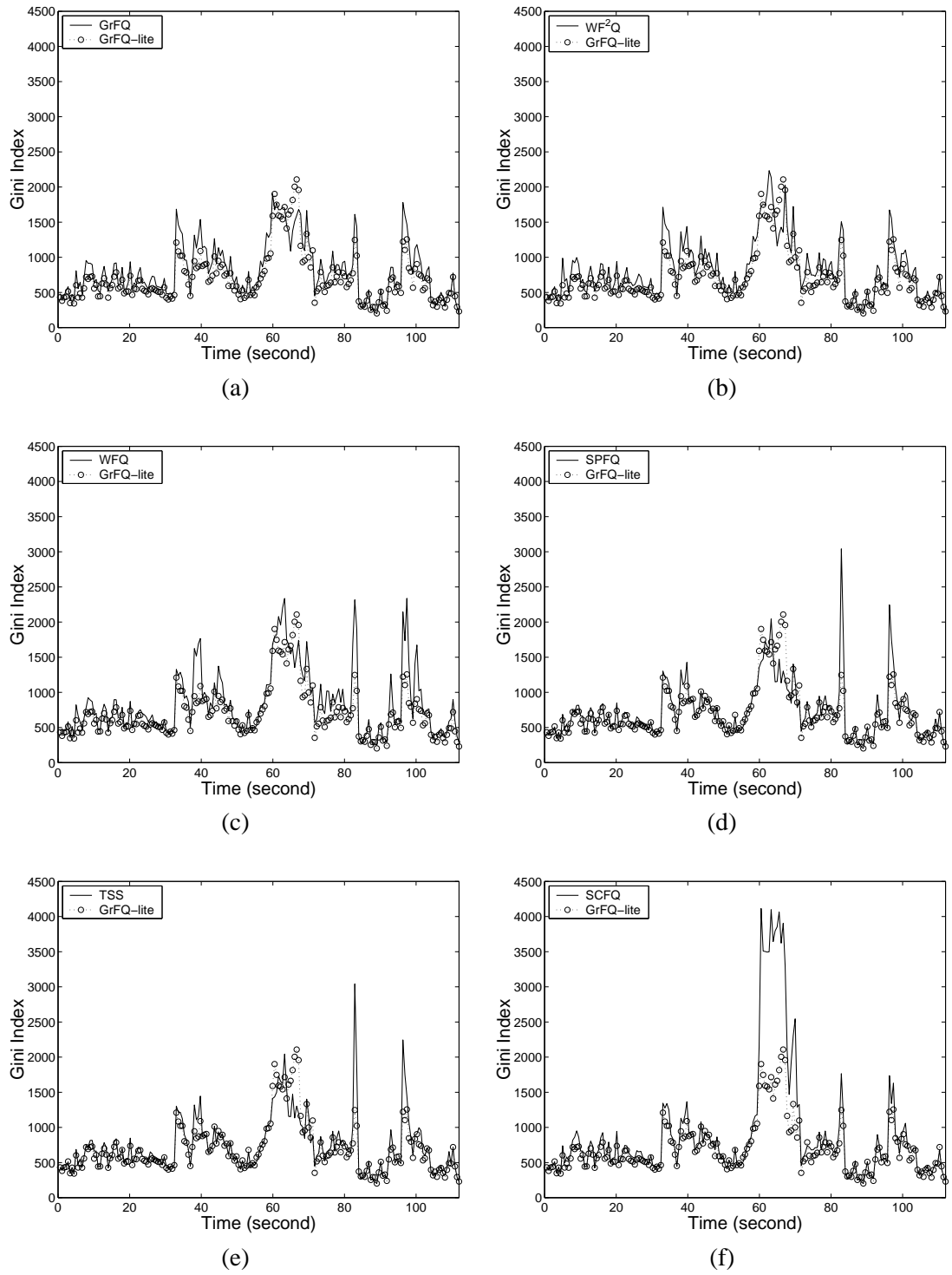
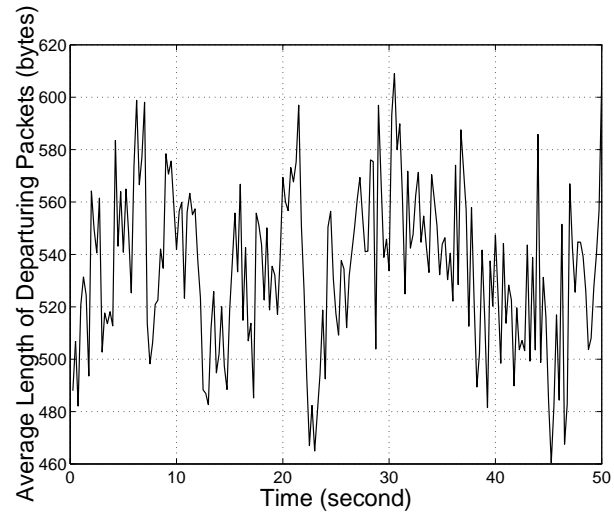
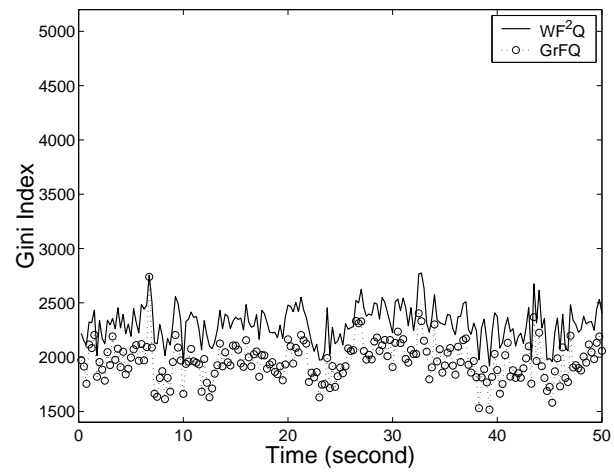


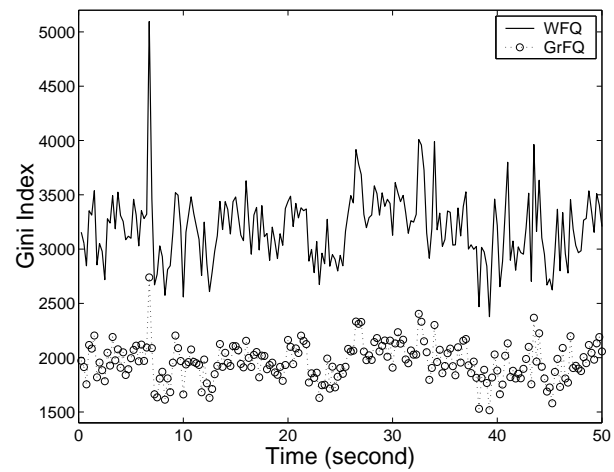
Figure 2.8: Gini indices of fair schedulers with real video traffic at 80% load



(a)



(b)



(c)

Figure 2.9: Gini indices of fair schedulers on backlogged queues with real gateway traffic traces

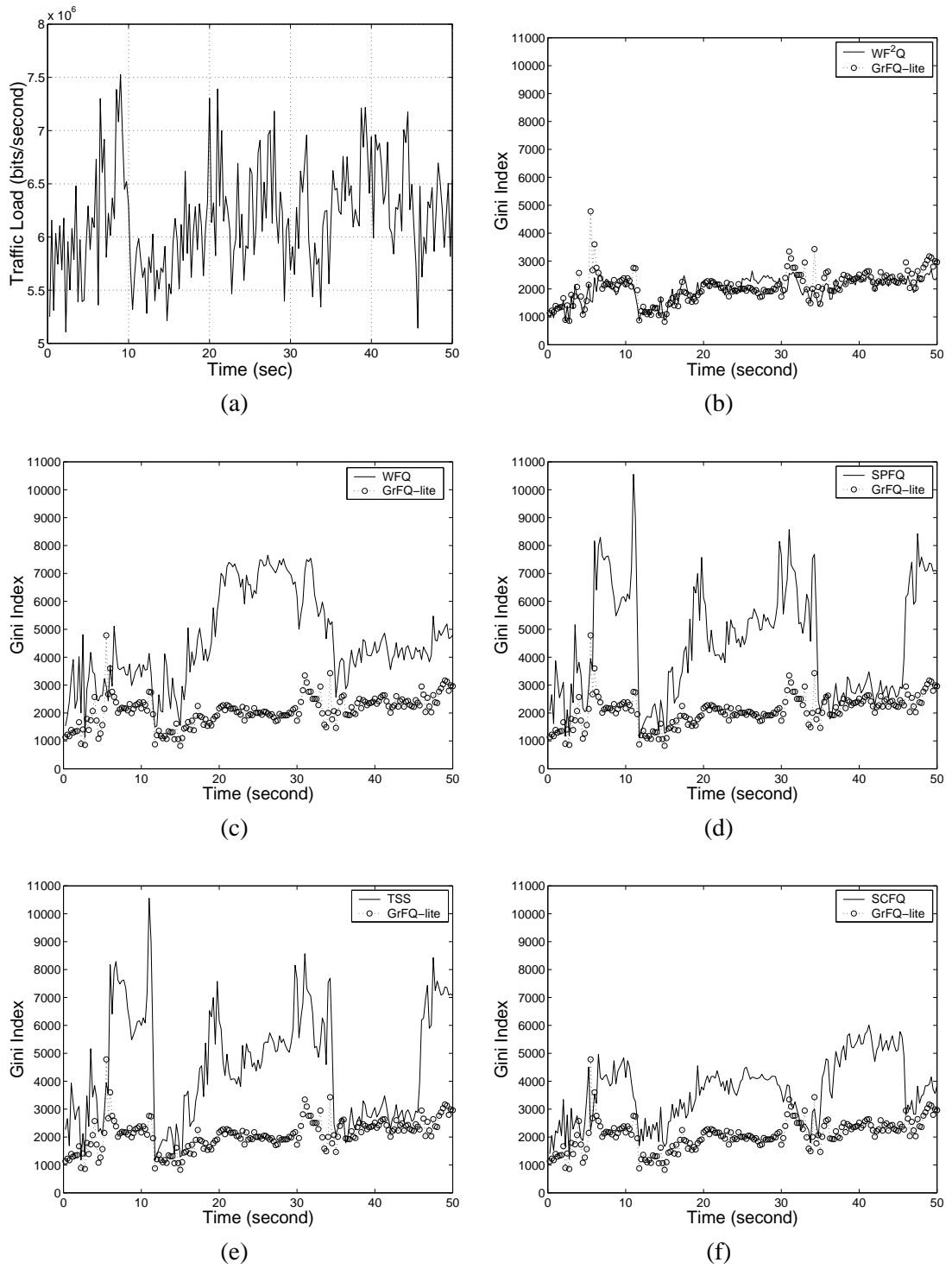


Figure 2.10: Gini indices of fair schedulers with real gateway traffic traces

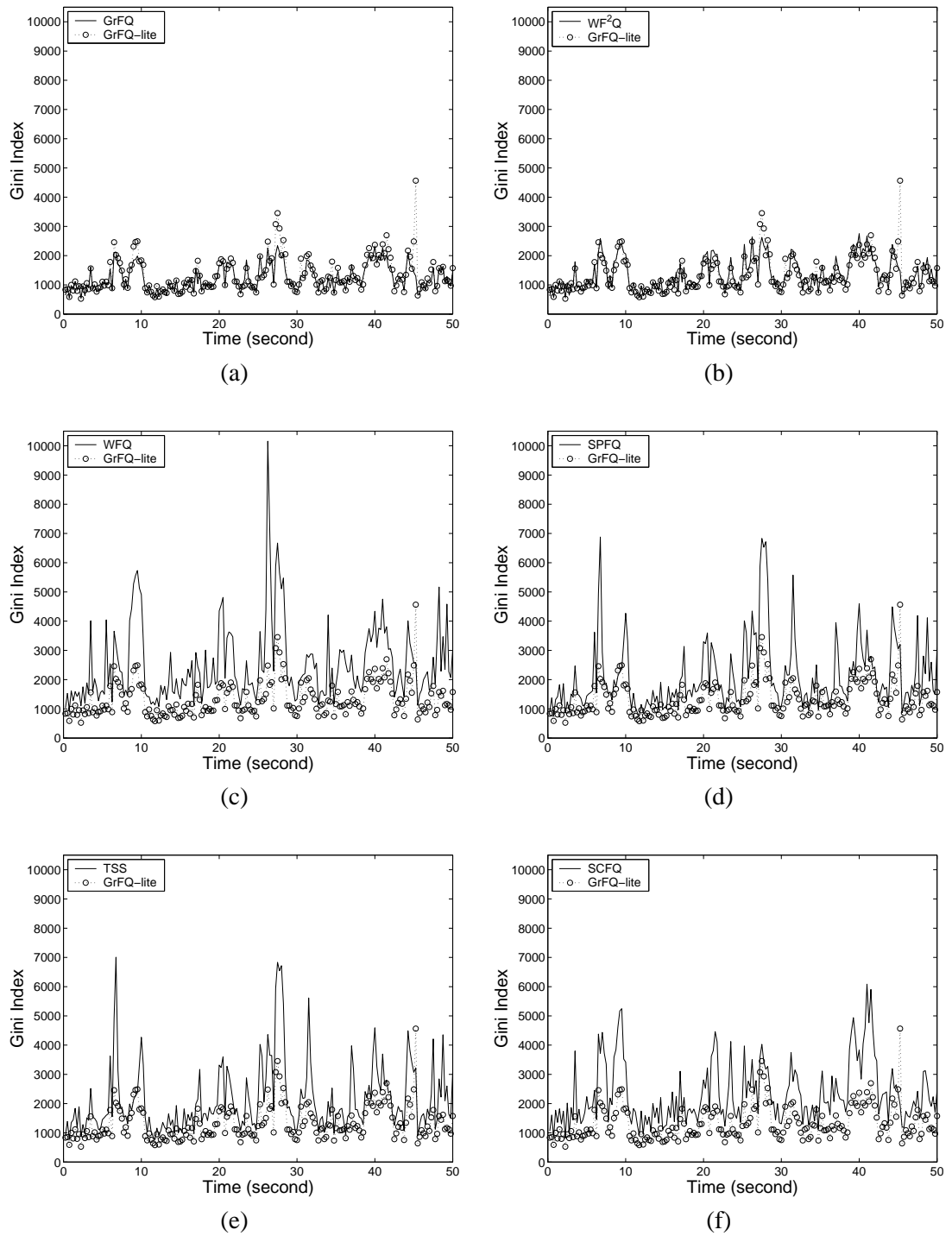


Figure 2.11: Gini indices of fair schedulers with real gateway traffic at 90% load

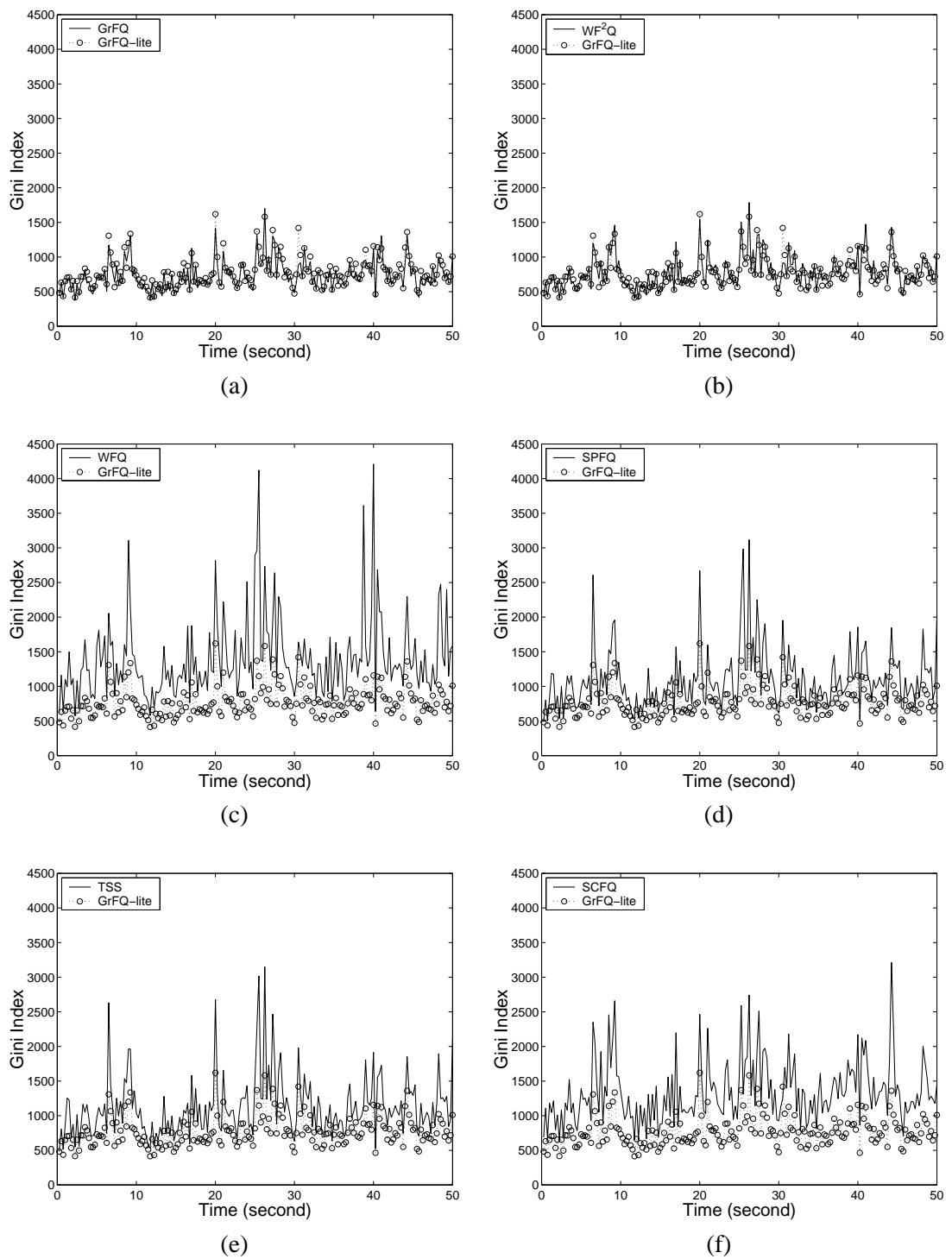


Figure 2.12: Gini indices of fair schedulers with real gateway traffic at 80% load

Chapter 3. Controlled Load Service

3.1 Network Mechanisms for Controlled Load Service

Controlled load service is defined under the Integrated Services model to provide users with a quality of service similar to that in a lightly loaded or unloaded network, and without requiring or specifying a target upper bound on the delay or loss probabilities. The idea behind this service model is that many real-time applications do receive adequate performance and quality of service in a lightly loaded network, eliminating the need for very strict performance guarantees. The desired quality of service is intended to be assured through capacity planning and admission control rather than through per-flow management during packet scheduling and forwarding. When a user exceeds traffic specifications approved by the admission control policy, the service obtained by the excess packets degenerates to the best-effort service.

Controlled load service allows a scalable means to achieve the required quality of service since it does not require the network to distinguish between flows beyond the admission control stage at the edges of the network. Each user/application provides an estimate of its traffic specifications, T_{spec} , and the service provider admits the traffic based on one of several possible admission control strategies that determine whether or not supporting the new user will still keep the network “lightly loaded” [53]. Packets sent by an application in excess of the T_{spec} agreed upon by the user and the service provider are marked by a traffic policer at the entry point into the network. As required by the definition of the controlled load service, the unmarked packets receive service similar to that in a lightly loaded network but the marked packets receive only a best-effort service. To preserve the generality of our solutions, we refer to these excess packets as simply *marked* packets and the rest as *unmarked* packets. Such marking of packets to indicate their level of importance

for dropping policies within the network is also used in the Differentiated Services model defined by the IETF [3] and in related buffer management and congestion control strategies as in the RED with In/Out (RIO) [38]. In addition, certain applications such as the multicasting of video streams employ schemes where the signal is encoded in layers so that progressively higher quality in the received stream can be achieved by receiving packets from more layers [11, 12]. Packets from layers corresponding to higher quality may be selectively dropped by the routers at congested points in the network.

Several admission control [53], packet forwarding and scheduling strategies [54, 55] have been suggested for use in routers to support the controlled load service. The authors in [55] propose a strategy that dynamically alters the priority of the packets (for e.g., marked or unmarked) to appropriately achieve the expected arrival time of each packet. The complexity of the algorithm is of $O(n \log n)$ with respect to n , the number of flows. However, the controlled load service was designed as a scalable alternative to providing guaranteed service for applications that do have certain quality of service requirements but which can, to some extent, adapt to changes in network conditions. One of the goals in the design of our scheduler for controlled load service is that it should be efficient with no per-flow management of flows, and with an $O(1)$ dequeuing complexity with respect to the number of flows and also the number of packets awaiting service. We consider it important to preserve this original intent in the implementation of mechanisms that support the controlled load service. To this end, as intended by the designers of this class of service and as also suggested in [54], we believe that a First-Come-First-Serve scheduling strategy is adequate for controlled load service, in combination with an effective admission control policy and a simple threshold-based buffer management strategy.

In addition to the simplicity and the scalability desired in the mechanism that supports the controlled load service, it is also desirable that we provide some guidelines on how to treat marked packets in relation to unmarked packets. For example, if capacity planning and admission control are reliably and correctly executed, the scheduler will have enough

bandwidth for the unmarked packets. However, it is desirable that as many marked packets be transmitted as possible while the delay caused to unmarked packets because of marked packet transmissions is bounded. Even though scheduling algorithms have been studied extensively for best effort traffic as well as for guaranteed services, scheduling strategies for merged packet streams with marked and unmarked packets requiring different levels of service have not been studied within a theoretical framework. Scheduling of packets seeking the controlled load service provides such a context with packets of the same flow belonging to different priority levels. In the design of the scheduler for controlled load service, our goal is to provide a framework and a mechanism that recognizes the trade-offs in the conflicting requirements of sending as many marked packets as possible, while ensuring that the unmarked packets of the same or other traffic streams continue to experience delays consistent with that in a lightly loaded network.

This chapter presents a scheduler of $O(1)$ per-packet complexity with respect to the number of flows and the number of packets awaiting service, and which ensures that the impact of marked packets on the delay experienced by an unmarked packet is bounded. In addition, it transmits very close to the maximum possible number of marked packets that may be transmitted while meeting the above goals.

3.1.1 Requirements of the Scheduler for Controlled Load Service

Our primary goal in the design of a scheduler for controlled load service is to preserve the original intent in the design and specification of the controlled load service. Certainly, it would be inappropriate to add implementation complexity to the service by adding per-flow management in the routers. Therefore, it is desirable that the scheduler use some simple discipline such as first-come-first-served (FCFS), while aggregating packets from all flows awaiting service by the scheduler into the same queue. Note that, in a lightly loaded network with regulated traffic, the FCFS scheduling discipline is expected to be

more than adequate. This facilitates the design of an efficient scheduler with a per-packet dequeuing complexity that is independent of the number of flows and also the number of marked or unmarked packets awaiting service in the queue. Also, by such a strategy which places all the packets in the same queue, the packets within the same flow, marked or unmarked, are delivered in order.

Secondly, the controlled load service packets do not have a delay or bandwidth specification. Therefore, a scheduler cannot make decisions based on delay requirements as in traditional guaranteed-service schedulers such as virtual clock or weighted fair queuing [56]. Instead, it is the capacity planning phase and admission control mechanism, based on the *Tspec* provided by the applications, that are responsible for ensuring that the packets can receive a delay approximating that in a lightly loaded network. Therefore, given effective capacity planning and admission control, it is sensible for the scheduler to assume that the unmarked packets of one flow will not affect the unmarked packets of another flow to the point that the network appears congested to any flow. However, the marked packet arrival characteristics are not part of the *Tspec* and therefore, unregulated. The scheduler does have to ensure that the impact of too many marked packets on the quality-of-service received by the traffic flows is kept under control within a certain acceptable bound. Since delay is the primary QoS parameter for real-time traffic, we can define the scheduler requirement as follows: the scheduler should guarantee that, *for any unmarked packet, the additional delay caused by marked packet transmissions is no more than a certain constant, α* . In other words, if an unmarked packet, in the absence of marked packets, could be forwarded with a delay of Δ , then the delay of the same packet in the presence of marked packets should be no more than $\Delta + \alpha$. The quantity α may be defined by the router or may be a negotiated quantity between service providers.

Finally, we do wish to send as many marked packets (unregulated, best-effort packets) as possible without violating the above requirement on the impact of marked packets on the delays experienced by unmarked packets, and also without violating the requirements

on the efficiency and complexity of the scheduler. This set of requirements is non-trivial to meet, especially in the absence of per-flow management. Note that in the absence of per-flow tracking and management of packet arrivals, the scheduler cannot predict with sufficient precision the new packet arrival characteristics, and therefore, cannot know whether sending a marked packet at a certain instant of time can be a cause for additional delay for unmarked packets at some later time.

In summary, the following are the goals in the design of our scheduler for controlled load service:

1. The scheduler should be efficient with no per-flow management of flows, and with an $O(1)$ dequeuing complexity with respect to the number of flows and with respect to the number of packets awaiting service.
2. The scheduler should be able to ensure that the impact of marked packets on the delay experienced by an unmarked packet is bounded.
3. Given the above two goals, the scheduler should be able to transmit as many marked packets as possible. For example, the scheduler should not trivially achieve the above goals by dropping all marked packets.

We assume that, within any given router, the quantity α is the same for all flows in the network. This critical parameter cannot be used in a cumulative fashion across multiple routers in the path of a flow since this would imply that schedulers would have to manage per-flow states. We assume that the capacity planning and the admission control phases will determine and set this parameter for each of the routers in the network prior to the beginning of transmissions that require such service.

3.2 The $CL(\alpha)$ Scheduler

The $CL(\alpha)$ scheduler for Controlled Load service presented in this section meets the requirements specified in the previous section. For any given α , the $CL(\alpha)$ scheduler ensures that the increase in the delay experienced by an unmarked packet due to the presence of marked packets is bounded by α .

The $CL(\alpha)$ scheduler maintains a single FCFS queue for all arriving packets. Marked as well as unmarked packets are all added to the tail of the same queue in the order of their arrival times. The $CL(\alpha)$ scheduler removes packets from the head of the queue for transmission, dropping marked packets if necessary. In our presentation of the scheduler, we assume that a marked packet transmission will not be pre-empted for the transmission of an unmarked packet. Consequently, we also assume that α is no smaller than the maximum possible length of time it may take to transmit a marked packet. Without this assumption, however, pre-emption will be necessary to ensure that marked packet transmissions do not increase the delay of an unmarked packet by more than α . While our presentation assumes no pre-emption for purposes of improved clarity, the $CL(\alpha)$ scheduler can be trivially changed to allow pre-emption if so desired.

In this paper, we use the following definitions of the delay of a packet and the extra delay of an unmarked packet at time instant t .

Definition 5 *The arrival time of a packet is the instant of time that the last bit of the packet arrives into the queue of packets awaiting transmission by a scheduler. The departure time of a packet is the instant of time that the last bit of the packet is transmitted by the scheduler. The delay of a packet at a scheduler is the length of the time interval between the arrival time and the departure time of the packet.*

Definition 6 *The extra delay, denoted by $ED_P(t)$, experienced at a scheduler by an unmarked packet, P , at time instant t is the cumulative additional delay that is experienced by P caused by transmissions of marked packets before time t .*

In a scheduling policy in which all marked packets are always dropped, the extra delay of an unmarked packet is always zero. The extra delay of a packet at a scheduler represents the difference between the delay experienced by the packet at the scheduler and the delay it would experience with a reference scheduler that drops all marked packets.

The extra delay, as defined above, is a function of time and changes whenever the scheduler chooses to send marked packets. If a scheduler chooses to drop all marked packets after time t , the extra delay of unmarked packets in the queue will not increase after time t . Obviously, when a marked packet is scheduled for transmission, all the unmarked packets in the queue will suffer an extra delay. In addition, some of the extra delay is “passed on” further to the unmarked packets which arrive after the transmission of the marked packet. This is because, in a first-come-first-served queue, a packet’s delay depends on the time at which its predecessor is served. Thus, the extra delay caused to one unmarked packet can cause an extra delay to unmarked packets that arrive later as well.

The extra delay of a newly arrived packet, however, is not merely equal to the extra delay suffered by its predecessor until this time, and can actually be less than that of its predecessor in the queue. This is best illustrated by considering an unmarked packet that arrives during a period of low congestion with only a small number of packets in the queue. However, the unmarked packet ahead of it in the queue, i.e., the predecessor packet, may have arrived in the queue during a period of heavy congestion when the queue length was large and thus may have a large extra delay associated with it. It is possible that the queue length later reduces and the newly arrived packet will only inherit a portion of the delay suffered previously by the predecessor packet. We will analyze these aspects of the extra delay in greater detail in Section 3.2.1.

The goal of the $CL(\alpha)$ scheduler is to ensure that $ED_P(t) \leq \alpha$ for all unmarked packets at all time instants t . In achieving this goal, the system has to (i) keep track of the changes in the extra delay of each packet in the queue and also (ii) determine the extra delay inherited by each new arriving packet from its predecessor packet. In the following, we describe an

efficient algorithm that manages these two important functions in the $CL(\alpha)$ scheduler.

3.2.1 Tracking Changes in $ED_P(t)$

To track the extra delay of packets in the queue as it changes with time, a naive method is to maintain an extra delay counter for each unmarked packet in the queue. The scheduler in such a case would have to check each of the extra delay counters before sending a marked packet. Upon sending a marked packet, it would have to update each counter by the transmission time of the marked packet. Obviously, this scheme has a processing delay proportional to the number of unmarked packets in the queue, with the potential of severely limiting scheduling efficiency when the queue length is large. The $CL(\alpha)$ scheduler, however, achieves significantly better scalability by inferring the extra delay of each unmarked packet from that of its predecessor packets in the queue. In fact, the $CL(\alpha)$ scheduler succeeds in achieving an $O(1)$ per-packet work complexity. Figures 3.1, 3.2 and 3.3 present a pseudo-code description of the $CL(\alpha)$ scheduler.

Recall that the FCFS queue consists of both marked and unmarked packets. We denote the first unmarked packet in the queue that has not yet completed transmission as the *unmarked head*. Note that an unmarked packet that is being transmitted at a certain instant of time is the unmarked head at that instant. Similarly, we define the *unmarked tail* as the last unmarked packet in the queue that has not yet completed transmission. The $CL(\alpha)$ scheduler maintains a record of the extra delays for the unmarked head and tail packets, denoted by *HeadED* and *TailED*, respectively. Both of these values have to be updated whenever a marked packet is transmitted while there are unmarked packets in the queue.

Suppose the arriving *unmarked* packets are labeled as 1, 2, ..., in the order of their arrival times. Let a_i be the arrival time of packet i . Let d_i be the departure time of packet i . If packet i arrives before packet $i - 1$ completes its transmission (i.e., $a_i < d_{i-1}$), note that neither of the packets would have completed transmission during the interval between

```

Initialize: (Invoked when the scheduler is initialized)
1  HeadED ← 0;
2  TailedED ← 0;

Enqueue: (Invoked when a packet P arrives)
3  if (P is unmarked) then
4      if (EDDQueueIsNotEmpty AND 5U < TailedED) then
6          AddToEDDQueue(TailedED - U);
7          TailedED ← U;
8      else
9          AddToEDDQueue(0);
10     end if;
11 end if;
12 AddPacketToQueue(P);

```

Figure 3.1: Pseudo-code of *Initialize* and *Enqueue* routines of the $CL(\alpha)$ scheduler; U , at any given time instant t , stands for $U(t)$

a_i and d_{i-1} . The additional accumulated extra delay due to marked packet transmissions during this time interval is the same for both packets. That is, for $a_i \leq t \leq d_{i-1}$, we have,

$$ED_i(t) - ED_i(a_i) = ED_{i-1}(t) - ED_{i-1}(a_i)$$

Thus,

$$ED_i(t) = ED_{i-1}(t) - [ED_{i-1}(a_i) - ED_i(a_i)] \quad (3.1)$$

Note that the quantity $ED_{i-1}(a_i) - ED_i(a_i)$ does not change with time, and is a constant for a given i . To further simplify our presentation and analysis, we define this quantity below.

Definition 7 Consider an unmarked packet i , and its predecessor unmarked packet $i - 1$. At the instant a_i when packet i arrives, if the predecessor packet has not yet completed its transmission, define *ExtraDelayDifference_i* or EDD_i of an unmarked packet i as the difference between the extra delay of packets $i - 1$ and i at time instant a_i . If the predecessor packet has already completed its transmission, define EDD_i as 0.

```

Dequeue:
13 while (QueueIsNotEmpty) do
14    $P \leftarrow \text{PacketAtHeadOfQueue}$ ;
15   if ( $P$  is unmarked) then
16      $\text{TransmitUnmarkedPacket}(P)$ ;
17   else /*  $P$  is marked. */
18     if (EDDQueueIsNotEmpty) then
19       if ( $\text{HeadED} + \text{TxTime}(P) \leq \alpha$ ) then
20          $\text{HeadED} \leftarrow \text{HeadED} + \text{TxTime}(P)$ ;
21          $\text{TailedED} \leftarrow \text{TailedED} + \text{TxTime}(P)$ ;
22          $\text{TransmitPacket}(P)$ ;
23       else
24          $V \leftarrow \text{UnmarkedHeadOfQueue}$ ;
25         Drop all packets ahead of  $V$ ;
26          $\text{TransmitUnmarkedPacket}(V)$ ;
27       end if;
28     else
29        $\text{TransmitPacket}(P)$ ;
30     end if;
31   end if;
32 end while;

```

Figure 3.2: Pseudo-code of *Dequeue* routine of the $\text{CL}(\alpha)$ scheduler

In other words, $\text{EDD}_i = \text{ED}_{i-1}(a_i) - \text{ED}_i(a_i)$ if unmarked packet i arrives while unmarked packet $i - 1$ is still in the system, and $\text{EDD}_i = 0$ otherwise.

Now, from (3.1),

$$\text{ED}_i(t) = \text{ED}_{i-1}(t) - \text{EDD}_i, \quad (3.2)$$

Therefore, $\text{ED}_i(t)$ can be obtained from EDD_i , and the extra delay of the predecessor packet in the queue, $\text{ED}_{i-1}(t)$.

For each unmarked packet i in the queue awaiting transmission, one may associate a constant value, EDD_i . The scheduler maintains these EDD values in a separate queue, which we shall denote by *EDDQueue*. The head of this queue, denoted by *EDDHead*, con-

```

TransmitUnmarkedPacket(P)
33 if ( $EDDQueue.length \geq 2$ ) then
34     Remove  $EDDHead$  from  $EDDQueue$ ;
35      $E \leftarrow CurrentHeadOfEDDQueue$ ;
36     if ( $E.EDD < HeadED$ ) then
37          $HeadED \leftarrow HeadED - E.EDD$ ;
38     else
39          $HeadED \leftarrow 0; TailedED \leftarrow 0$ ;
40     end if;
41     TransmitPacket(P);
42 else
43     TransmitPacket(P);
44     Remove  $EDDHead$  from  $EDDQueue$ ;
45     /* New unmarked packets may arrive during the transmission. */
46      $E \leftarrow CurrentHeadOfEDDQueue$ ;
47     if ( $E \neq NULL$  AND  $E.EDD < HeadED$ ) then
48          $HeadED \leftarrow HeadED - E.EDD$ ;
49     else
50          $HeadED \leftarrow 0; TailedED \leftarrow 0$ ;
51     end if;
52 end if;

```

Figure 3.3: Pseudo-code of *TransmitUnmarkedPacket* routine used by *Dequeue* routine of the $CL(\alpha)$ scheduler

tains the EDD value corresponding to the unmarked head. Similarly, the tail of this queue, denoted by $EDDTail$, contains the EDD value corresponding to the unmarked tail. Let packet h be the unmarked head at time t . If there is another unmarked packet in the queue, then the next unmarked packet in the queue, packet $h + 1$, should have a corresponding entry in the EDD queue equal to EDD_{h+1} . Note from Equation (3.2) that $ED_{h+1}(t)$ can be computed from EDD_{h+1} and the $HeadED$ (extra delay of packet h) at time t . After packet h is transmitted, the value computed for ED_{h+1} becomes the new $HeadED$, since packet $h + 1$ is now the new unmarked head. Thus, $HeadED$, always contains the extra delay corresponding to the current unmarked head which has not yet completed transmission.

3.2.2 Computing *EDD*

The above mechanism of tracking the extra delay of each unmarked packet relies upon knowledge of the correct *EDD* value corresponding to the packet. The $CL(\alpha)$ scheduler sets this value for each unmarked packet at the instant that the packet arrives into the queue. We will need the following definition and lemma to explain the algorithm used to determine the *EDD* value of an unmarked packet.

Definition 8 *Consider the system at time instant t . Define $U(t)$ as the minimum possible additional time it will take for the unmarked tail in the system at time t to complete its transmission.*

$U(t)$ may also be thought of as the additional time it will take a packet that arrives at time t to begin its transmission if all marked packets in the system that have not yet begun transmission at time t are dropped. In this paper, we assume that unmarked packets will not pre-empt the transmission of a marked packet. Therefore, $U(t)$ includes the residual transmission time of the packet being transmitted at time t even if it is a marked packet. Thus, $U(t)$ is the sum of this residual transmission time and the transmission times of all the unmarked packets in the queue at time t awaiting the beginning of transmission.

We now proceed to obtain an expression for $ED_i(a_i)$ that facilitates the computation of the *EDD* values.

Lemma 1 *During an execution of the $CL(\alpha)$ scheduler, when unmarked packet i arrives into the queue at time instant a_i ,*

$$ED_i(a_i) = \min\{ED_{i-1}(a_i), U(a_i)\},$$

if its predecessor, packet $i - 1$, has not yet completed transmission ($a_i \leq d_{i-1}$), and,

$$ED_i(a_i) = U(a_i),$$

otherwise.

Proof: Since $ED_i(a_i)$ records the cumulative additional delay of packet i caused by marked packet transmissions before time a_i , we prove the statement of the theorem by comparing the departure time of the packet in the $CL(\alpha)$ scheduler assuming no marked packet is transmitted after a_i and its departure time in a reference system which drops all marked packets.

In this proof, we now separately consider each of the two cases in the statement of the lemma.

Case 1 ($a_i \leq d_{i-1}$): In this case, the predecessor packet has not yet completed transmission at time a_i . If an unmarked packet i arrives at time a_i , the earliest possible time at which it can begin transmission is $a_i + U(a_i)$, which can occur only if no additional marked packets are transmitted after time a_i . Its earliest departure time is $a_i + U(a_i) + L_i/R$, where L_i is the length of the packet and R is the peak rate of the link. Let \hat{d}_i be the departure time of the packet using the reference scheduler that drops all marked packets. The extra delay of packet i at time a_i , $ED_i(a_i)$, is the component of the delay caused to the packet due to transmissions of marked packets before time instant a_i . Therefore, this is nothing but the difference between $a_i + U(a_i) + L_i/R$ and \hat{d}_i . In other words,

$$ED_i(a_i) = a_i + U(a_i) + L_i/R - \hat{d}_i \quad (3.3)$$

We will now consider two sub-cases and use the Equation (3.3) above to prove the lemma.

Sub-Case A ($ED_{i-1}(a_i) \leq U(a_i)$): This sub-case is illustrated in Figure 3.4(a). In this case, packet $i - 1$ departs at time $\hat{d}_{i-1} = a_i + [U(a_i) - ED_{i-1}(a_i)]$ in the reference system. In the reference system, packet i can begin its transmission at time \hat{d}_{i-1} and completes the transmission at time $\hat{d}_i = \hat{d}_{i-1} + L_i/R$. Using Equation (3.3),

$$\begin{aligned} ED_i(a_i) &= \left(a_i + U(a_i) + \frac{L_i}{R} \right) \\ &\quad - \left(a_i + [U(a_i) - ED_{i-1}(a_i)] + \frac{L_i}{R} \right) \\ &= ED_{i-1}(a_i). \end{aligned}$$

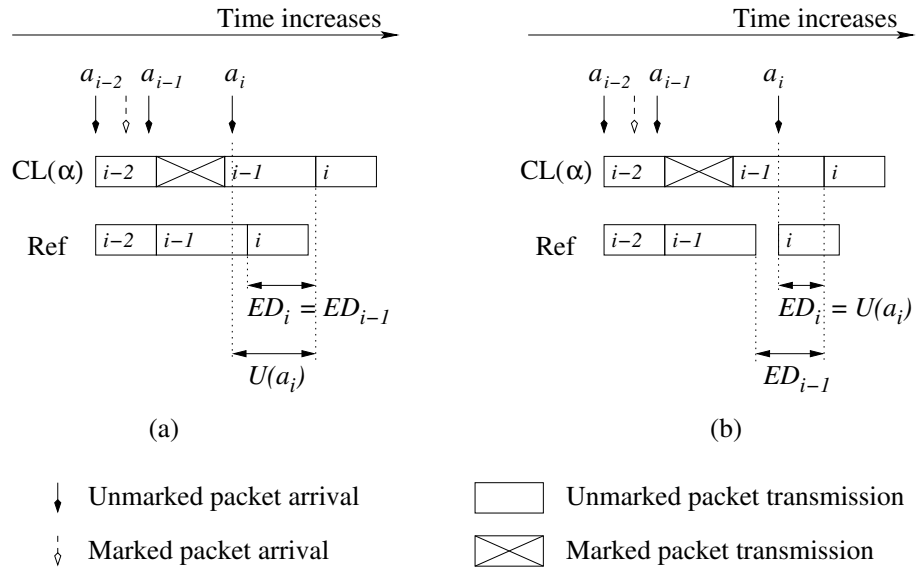


Figure 3.4: Illustration of the sub-cases 1A and 1B in the proof of Lemma 1

Sub-Case B ($ED_{i-1}(a_i) > U(a_i)$): This sub-case is illustrated in Figure 3.4(b). In this sub-case, packet $i - 1$ has already completed its transmission in the reference system at time instant a_i . Therefore, in the reference system, packet i will find the queue empty upon arrival at time a_i and will begin transmission immediately. Thus, the departure time for packet i in the reference system is $\hat{d}_i = a_i + L_i/R$. Using Equation (3.3) again, we have,

$$ED_i(a_i) = \left(a_i + U(a_i) + \frac{L_i}{R} \right) - \left(a_i + \frac{L_i}{R} \right) = U(a_i).$$

Case 2 ($a_i > d_{i-1}$): For the case that packet $i - 1$ has completed its transmission at time a_i in the $CL(\alpha)$ scheduler, the extra delay of packet i is not simply zero even though it is the only unmarked packet in the queue. When packet i arrives, the scheduler may be transmitting a marked packet. Since the scheduler does not pre-empt this transmission, the unmarked packet will acquire an extra delay equal to $U(a_i)$, the residual transmission time of the marked packet currently being transmitted. \square

Lemma 1 relates the extra delay of a newly arrived packet at time a_i to that of its predecessor packet and to $U(a_i)$. The extra delay of the predecessor packet is nothing but the *TailedED* maintained by the scheduler. $U(t)$ is also easily maintained by the scheduler with updates upon the arrival and departure of packets. This allows an easy computation of the *EDD* value corresponding to each packet, representing the difference between the extra delay values at time a_i between that of packet i and its predecessor.

3.2.3 Limiting *ED* to α

In Lemma 1, it is proved that $ED_k(t)$ is no more than $ED_{k-1}(t)$. Thus, the unmarked head has the largest $ED(t)$ for any given t among all the unmarked packets in the queue. When the scheduler tries to send a marked packet, it only needs to make sure that the *HeadED* will not exceed α . Thus, checking the *EDs* of all the packets in the queue is rendered unnecessary, and thus reduces the per-packet work complexity to $O(1)$.

If *HeadED* will exceed α upon transmission of a marked packet, the scheduler will drop that marked packet and all marked packets ahead of the unmarked head before beginning the transmission of the unmarked head. Searching the queue for the next unmarked packet is obviously not a scalable option, and will not preserve the $O(1)$ complexity of this algorithm. Therefore, the $CL(\alpha)$ scheduler associates with each element of *EDDQueue* a pointer that indicates the position or address of the corresponding unmarked packet in the queue. Recall that each element of *EDDQueue* corresponds to a unique unmarked packet in the queue. When the scheduler determines that the marked packet at the head of the queue cannot be transmitted, it can simply look up the pointer associated with the *EDDHead* and send the unmarked packet corresponding to it.

3.3 Analysis

In this section, we present an analysis of the performance and the efficiency of the $CL(\alpha)$ scheduler and prove that the scheduler satisfies the requirements listed in Section 3.1.1.

Theorem 4 *The $CL(\alpha)$ scheduler has a per-packet work complexity of $O(1)$.*

Proof: All of the operations in the enqueueing and dequeueing routines are shown in Figures 3.1, 3.2 and 3.3. The theorem is proved by showing that the number of these operations is $O(1)$.

To enqueue an unmarked packet, one needs to find out the states of the scheduling system, set EDD and add it to $EDDQueue$, set $Tailed$, and finally append the packet to the end of the queue. To enqueue a marked packet, the server needs only one operation of appending the packet to the end of the queue. In either case, the number of operations of constant time complexity is bounded by a small finite constant. Thus, the *Enqueue* routine in Figure 3.1 has a per-packet work complexity of $O(1)$.

In dequeueing an unmarked packet, one needs to update $HeadED$ and $Tailed$, operations that are readily verified to be of $O(1)$ time complexity. In dequeueing a marked packet, the scheduler needs to first determine if it should be transmitted at all, which is based on a simple comparison operation. If the packet is to be transmitted, the scheduler only needs to update $HeadED$ and $Tailed$, which again involves only an $O(1)$ complexity in time. If the marked packet is not to be transmitted, the scheduler dequeues the unmarked head packet in $O(1)$ time using the pointer stored in the elements of the $EDDQueue$ and without going through a search operation among the packets. Thus, the total time taken to dequeue a marked packet is also of complexity $O(1)$. \square

3.3.1 Bound on the Extra Delay

In the following, we prove that the $CL(\alpha)$ scheduler correctly computes the extra delay of each unmarked packet, and that the $CL(\alpha)$ scheduler successfully bounds the extra delay of each unmarked packet to α .

Theorem 5 *During any execution of the $CL(\alpha)$ scheduling discipline, the additional delay of an unmarked packet caused by the transmission of marked packets is never greater than α .*

Proof: The $CL(\alpha)$ scheduler computes the values in the *EDDQueue* based on Lemma 1. When the unmarked head is transmitted, as per Equation (3.2), a new value of *HeadED* is computed as the difference between the previous value and the EDD value corresponding to the new unmarked head. Thus, *HeadED* represents the extra delay of the new unmarked head. For each marked packet transmission thereafter, the $CL(\alpha)$ scheduler increments the *HeadED* value by the transmission time of the marked packet. Thus, the *HeadED* value always contains the extra delay of the unmarked head packet at all time instants. Recall that the $CL(\alpha)$ scheduler does not transmit a marked packet if its transmission time plus *HeadED* is larger than α . Therefore, if an unmarked packet becomes the unmarked head with an extra delay of less than or equal to α , it will be transmitted early enough to ensure that its extra delay never goes beyond α .

Using induction on the sequence on unmarked packets, we now proceed to show that every unmarked packet has an extra delay less than or equal to α when it becomes the unmarked head.

As the basis step of the induction, consider the very first unmarked packet that arrives at the scheduler. This becomes the unmarked head with a *HeadED* value equal to the residual time of the current marked packet transmission, which is guaranteed to be less than or equal to α .

As part of the inductive step of the proof, assume that each of the unmarked packets until packet $i - 1$, i.e., each unmarked packet that arrive at or before a_{i-1} , experiences an extra delay of no more than α when it becomes the unmarked head. We have to now prove that packet i will also experience an extra delay of no more than α when it becomes the unmarked head. We consider two cases:

(a) If unmarked packet i arrives after packet $i - 1$ completes transmission, it immediately becomes the current unmarked head similar to the case of the first unmarked packet that arrives in the system. Therefore, it has an extra delay no more than α when it becomes the unmarked head.

(b) If unmarked packet i arrives before packet $i - 1$ completes transmission, from Lemma 1, the extra delay of packet i is always less than or equal to the extra delay of packet $i - 1$ until packet $i - 1$ completes transmission. Since packet $i - 1$ is transmitted with an extra delay of less than or equal to α , packet i will have an extra delay of no more than α when it becomes the unmarked head.

From the above inductive proof, every unmarked packet has an extra delay less than or equal to α when it becomes the unmarked head. Now, as long as the packet has an extra delay of less than or equal to α when it becomes the unmarked head, it will be transmitted before its extra delay goes higher than α as is ensured by the $CL(\alpha)$ scheduler. This proves the theorem. \square

3.4 Simulation Results

The effectiveness of the $CL(\alpha)$ scheduler may be demonstrated using simulation. In our simulation, we use seven sources, each of which generates an MPEG-4 video stream. These video streams of certain popular movies and sports programs are from the traces made available by the Telecommunication Networks Group at the Technical University of Berlin, Germany [51]. In order to remove any effects due to possible correlation between

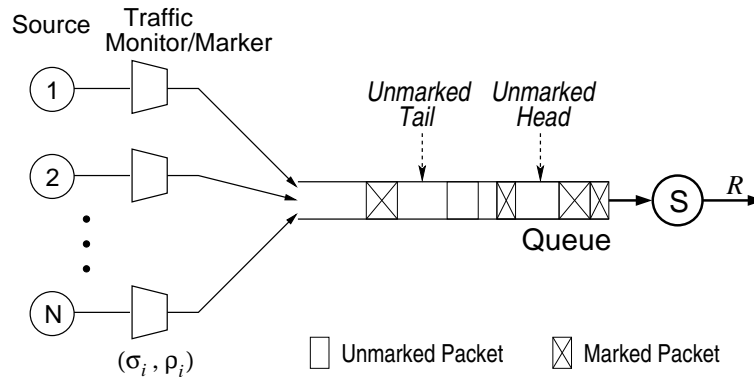


Figure 3.5: Simulation set-up of the $CL(\alpha)$ scheduler

early portions of the video streams, in our simulation, each source begins transmitting at a random point within the movie trace. The generated traffic is policed by token bucket regulators and associated packet markers before it arrives at our $CL(\alpha)$ scheduler. The traffic policer and marker for flow i is configured to allow a long-term average rate of ρ_i and a maximum burst size of σ_i . Packets in the source traffic that do not conform to these token bucket parameters are marked. Figure 3.5 illustrates the simulation setup. The details of the video stream sources and the token bucket parameters are listed in Table 3.1.

We assume that, through the capacity planning phase, traffic policers are configured so that $\sum_i \rho_i \leq R$, where R is the peak link rate at the output of the scheduler. In our experiment, we use values of the token generation rates, ρ_i , such that $\sum_i \rho_i \approx 0.98R$. We also ensure that each source traffic has a higher long-term average rate than allowed by the policer, so that a sufficient number of marked packets are generated to verify the algorithm. In our simulation, the peak link rate R is selected to be smaller than the sum of the average transmission rates (not the same as the corresponding token generation rate) of video streams so that the input queue is backlogged most of time with either marked or unmarked packets, and so that some marked packets would have to be dropped.

Our simulation implements an instance of the $CL(\alpha)$ scheduler where $\alpha = 50$ ms. The

Table 3.1: Settings for MPEG-4 traffic sources and token bucket regulators

Source	1	2	3	4	5	6	7
Movie Name ⁶	J	S	W	B	D	K	F
Video Quality	High	Medium	High	Medium	High	High	High
L_{\min} (bytes)	72	28	26	27	71	307	130
L_{\max} (bytes)	16,745	11,915	9,370	7,565	16,960	15,813	14,431
r_{avg} (Kbps)	770	180	280	180	700	830	840
r_{peak} (Mbps)	3.3	2.4	1.9	1.5	3.4	3.2	2.9
ρ_i (Kbps)	567	135	203	135	473	567	709
σ_i (bytes)	16,755	11,935	9,384	7,570	16,982	15,848	14,458
Link Capacity	2.83 Mbps						
α	50 ms						
Total Time	160 seconds						

duration of the simulation is 160 seconds. Figure 3.6(a) shows a cumulative distribution of the extra delay of unmarked packets that go through the $\text{CL}(\alpha)$ scheduler over the length of the experiment. Figure 3.6(b) shows the distribution density represented as a histogram of the extra delay experienced by these unmarked packets. Figure 3.6 verifies that no unmarked packets suffer an extra delay greater than $\alpha = 50$ ms in the $\text{CL}(\alpha)$ scheduler.

Once the $\text{CL}(\alpha)$ scheduler determines that transmitting the marked packet at the head of the queue will increase the extra delay of an unmarked packet beyond α , it decides to send the unmarked packet and drop all marked packets ahead of it. This ensures the $O(1)$ per-packet complexity of the scheduler. At a greater complexity, one might design a scheduler that tries to send a smaller marked packet that will also not increase the extra delay of the unmarked head beyond α , and thus increase the number of marked packets that are transmitted. In fact, an ideal scheduler that maximizes the data in marked packet transmissions will need to examine each marked packet in the queue ahead of the unmarked

⁶The alphabet letters stand for the following movies and sports programs:

J: "Jurassic Park I" S: "Silence Of the Lambs" W: "Star Wars IV"
 B: "Mr. Bean" D: "Die Hard III" K: "Alpine Ski" F: "Formula 1"

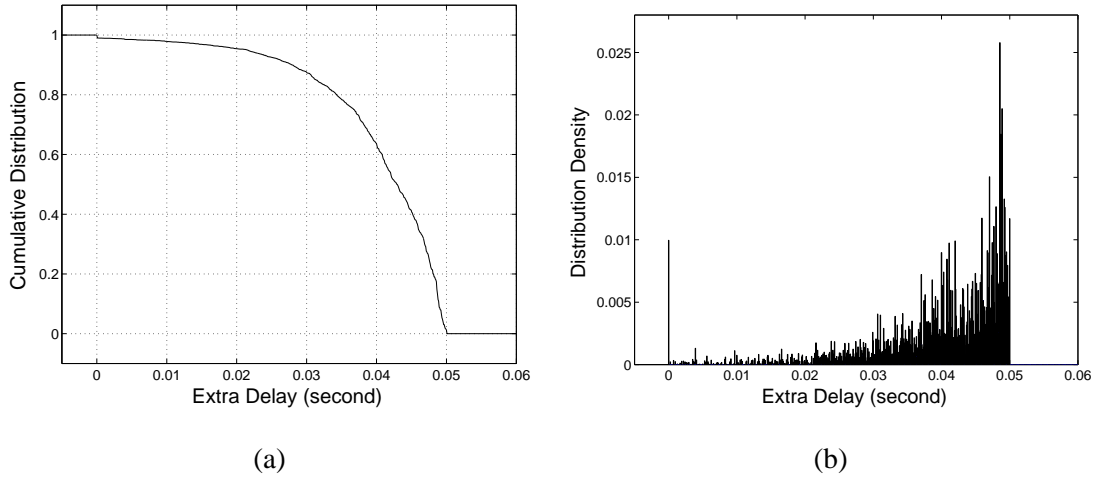


Figure 3.6: Distribution of the extra delay from $CL(\alpha)$ scheduler: (a) cumulative distribution (b) distribution density

head, and send exactly the set of marked packets that together make up the largest amount of data that can be transmitted without increasing the extra delay of the unmarked head beyond α . However, this will require the scheduler to examine each of the marked packets in the queue ahead of the unmarked head, and the number of packets one may have to examine is unbounded except by the size of the queue. Our $CL(\alpha)$ scheduler makes a compromise in favor of achieving simplicity of implementation and a lower per-packet work complexity. Figure 3.7 shows the amount of data from marked packets transmitted by the $CL(\alpha)$ scheduler and that transmitted by the ideal scheduler under the simulation setup described earlier. The figure illustrates that the amount of data in marked packets transmitted using the $CL(\alpha)$ scheduler is almost identical to that transmitted by an ideal but more complex scheduler.

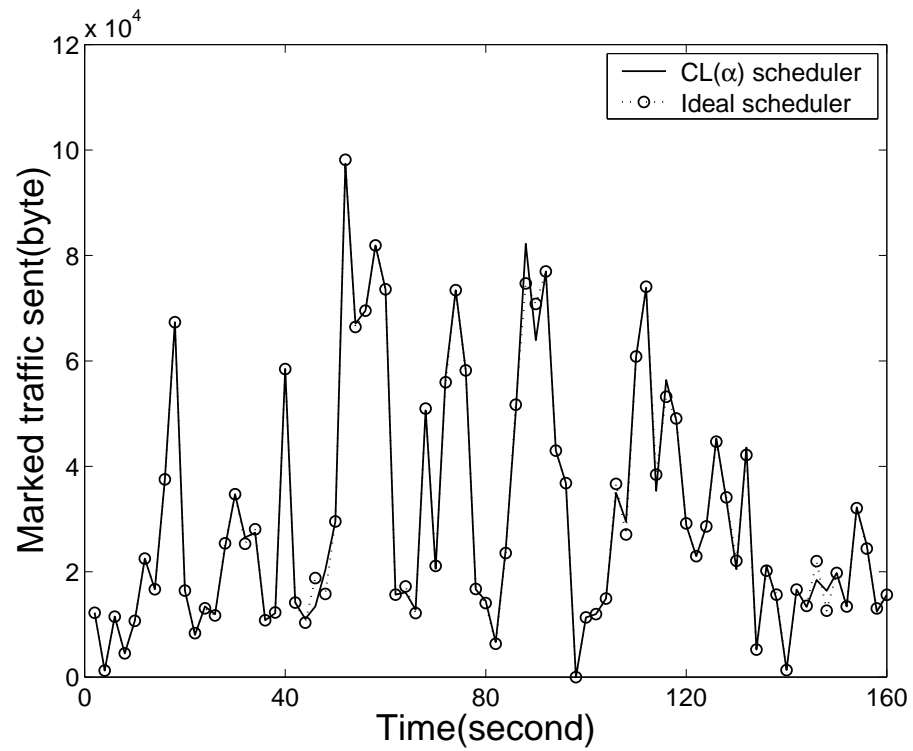


Figure 3.7: The amount of data from marked packets sent by the CL(α) scheduler and by the ideal but more complex scheduler

Chapter 4. Soft Real-Time Service

4.1 Background

4.1.1 Scheduling Real-Time Traffic

As more people use the Internet as a medium to share and distribute multimedia information, researchers and developers are motivated to design a versatile network to support such transmissions with a satisfactory level of quality-of-service. Since multimedia data tends to consume large amounts of local resources, multimedia traffic is frequently streamed and then played back in real time. Such real-time traffic, therefore, typically has much tighter requirements on the timeliness of transmission than traditional data traffic. A variety of techniques have been proposed and employed to provide guaranteed delays to real-time traffic flows. We discuss some these below.

The First Come First Served (FCFS) scheduling strategy, used in most early routers, achieves the minimum average delay among all flows. However, it cannot provide any differentiation between real-time traffic and elastic traffic, and therefore it cannot provide a delay guarantee. The static priority scheduler has been proposed to provide some level of discrimination between real-time traffic and elastic data traffic [54]. In this method, the lower priority queue is serviced only when the higher priority queue is empty. Thus, real-time traffic in a higher-priority queue has better chance of transmission with a lower delay. However, it still cannot provide any level of quality assurances to real-time traffic.

Fair queueing schedulers are known to allocate bandwidth resource fairly to flows, and can be used to also achieve guaranteed delays. One group of such schedulers, including Weighted Fair Queueing (WFQ) [16, 20] and its variants [21–23], provide a closed-form relationship between transmission delay bound and bandwidth allocation. Their service principles are all based on an ideal fair scheduler model, Generalized Processor Sharing

(GPS). A slightly different approach, used in Starting Potential Fair Queueing (SPFQ) [40], is based on the concept of Rate-Proportional Servers [57], where GPS is not the basis for scheduling. Instead, scheduling decisions are made based on a global variable that is constantly updated upon the transmission of each packet. The SPFQ scheduler achieves the same relationship between the delay bound and the minimum bandwidth allocation as that achieved by WFQ. Such a relationship provides a feasible way to guarantee transmission delay bounds to real-time flows.

The delay bound guaranteed to a flow by fair queueing schedulers such as the above is inversely proportional to the minimum bandwidth allocated to the flow. Thus, a flow desiring a low delay has to reserve a large share of the bandwidth in order to ensure a delay guarantee. This can frequently result in a situation where the router has to allocate a larger amount of bandwidth resource than what is actually consumed by a flow. This is because a flow may require low delays but may not necessarily be a high-volume flow. Thus, in using fair queueing to guarantee delay bounds, the network resources cannot be fully utilized because the provisioning and admission control have to make sure that the network has enough resource to support the delay requirements of existing flows.

A different approach, Earliest Deadline First (EDF) [58], is able to decouple the proportional relationship between delay bound and bandwidth. Packets are stamped with desired deadlines on them and are scheduled in order of their deadlines. Therefore, a flow can be served with low delay through having an early deadline rather than through reserving a large bandwidth. Together with an admission control policy, EDF can guarantee delay bounds to real-time traffic.

Some real-time applications with stringent requirements on delay bounds can be largely satisfied by fair queueing schedulers or the EDF scheduler. However, most real-time applications do not strictly require such a guarantee on the transmission delay bound for all its packets. For these applications, playback quality is tolerable in spite of some packet losses. This means that networks with mechanisms for delay bound guarantees provide

services with higher quality than what clients are willing to pay. To efficiently utilize network resources, therefore, a frequent technique has been to deliberately reduce the allocated resource to real-time flows based on the statistical pattern of the incoming network traffic [59]. Thus, even though the network is over-provisioned from the view of guaranteed service, flows are still serviced with good quality most of the time. Although networks are better utilized by implementing such statistical guaranteed services schedulers, flows are not protected well enough against each other. Once an unexpected burst of one flow arrives at the system, other flows have to suffer a degradation in service quality since most of the resource is temporarily consumed by the burst. To avoid such degradation, we need a scheduling discipline which does not over-allocate bandwidth resource but provides good protection from bursty traffic. Further, even though real-time traffic always needs an assurance on delay bound, it benefits from transmission with a low delay. Many playback algorithms nowadays have incorporated mechanisms which adapt playback delays to the current condition of a network.

4.1.2 Fairness Issues in Scheduling Real-Time Traffic

Some real-time applications have stringent requirements on timely deliveries, while others do not require services with strict guarantee. According to the service requirements, real-time applications are usually classified into hard and soft applications. Hard real-time applications require strict on-time completion of the transmission of each packet. Such applications need complete guarantee on the delay bound from the network. Soft real-time applications can tolerate sparse packet losses and overdue transmissions. Their quality is not degraded significantly as long as most packets (not necessarily all) arrive on time. In the following, we will consider a packet that is delayed beyond its deadline as a lost packet. Soft real-time applications have a QoS requirement that is frequently captured by two parameters, D and x : packet transmissions should achieve a delay bound of D with a

loss rate of $x\%$.

How we compute the packet loss rate in the case of soft real-time applications is very different from that in the case of non-real-time applications. When a data file is transmitted through the network, the loss rate of the transmission refers to the portion of data from the file which is lost in the network. This kind of loss rate is evaluated over the long term such as over the entire session or the entire transmission of the file. Soft real-time applications impose a requirement on the short-term loss rate, i.e., the loss rate needs to be less than $x\%$ over smaller intervals of time as well and not just over the entire length of the session. For example, in a voice-over-IP application, a loss rate of up to 5% is acceptable as long as the losses are uniformly distributed during the session but a 5% loss rate is not acceptable if all the losses occur in a burst all at once.

Many video and audio communications, except for secure communications, can be classified as soft real-time applications. Nowadays, many real-time interactive communication applications, such as *vic* and *vat*, are designed to adapt the current condition of the network and adjust their playback delays. For example, if the network is not congested and the delays experienced are low, the application will use a smaller playback delay to enhance the quality of interactive communications. They benefit from low average delays and can present satisfactory quality even with scattered losses of data. Many coding/decoding schemes can reconstruct the original information using received data with scattered losses. However, the acceptable level of sparse losses depends on the frame size used for coding/decoding within each specific application. Therefore, the network should provide assurances to soft real-time applications with various different requirements on delay and losses.

We now introduce our metrics and notation for quantifying the delay and short term loss rate requirements of flows. For a packet scheduler, the transmission delay of a packet is defined as the length of the time interval between a packet's arrival time and the time it completes its transmission. In [60], a framework is proposed to quantify and stipulate

QoS requirements regarding short-term losses. This framework specifies QoS requirements based on (m, k) -firm deadlines, where the quality of service delivered by the network is acceptable to a flow if at least m packets within a window of k consecutive packets are transmitted before their deadlines. Note that the window is a sliding window which means it can start at any packet and end k packets later. As discussed before, the values of m and k will depend on the specific application.

Borrowing from [60], consider a queueing system with N flows. Flow i has a requirement of (m_i, k_i) -firm deadline. The system maintains a record of the recent service results on deadlines met or missed for each flow. Let x_i^j denote the transmission result of packet j from flow i , where the result can be a *miss* or a *meet* depending on whether packet j missed or met its deadline. A packet is considered as missing the deadline if it is dropped by the system. The state of flow i is determined by the k_i -tuple, $(x_i^{j-k_i+1}, \dots, x_i^{j-1}, x_i^j)$, where j is the index of the most recent packet served by the system. We call those states with fewer than m_i meets as *failing states* for flow i . For each flow, a state transition diagram can be drawn. In the transition diagram, a flow becomes “closer” to a failing state when a packet misses its deadline. In [60], the rate at which a flow experiences failure is considered as the metric of quality of service. Thus, the lower the rate, the higher the quality. The scheduling discipline proposed in [60] attempts to keep flows away from failing states. It assigns higher priority to flows which are closer to failing states. In this specific approach, the priority of a flow equals to the minimum number of consecutive misses required to move the flow from its current state to a failing state. If we define the distance of a non-failing state from a failing state as the number of transitions between them, the priority of a flow with a non-failing states also equals the minimum distance to any failing state. Thus, the approach is named as *Distance-Based Priority* (DBP) assignment.

4.1.3 Fairness of Existing Schedulers with QoS Assurance

While the basic requirements and the associated metrics for scheduling soft real-time applications described above are a good starting point, they are not sufficient to serve as a guide toward designing schedulers. An important requirement on the schedulers should be that they maximize the utilization of network resources while also providing a similar level of delays and loss rates to every flow. This requirement is similar to the idea of fair queueing schedulers in the best-effort service model. Fair queueing schedulers attempt to provide an equal opportunity to flows sharing the same transmission link. Similarly, when serving real-time traffic, the level of QoS assurance should be fairly allocated such that no flow suffers from degraded service while another flow receives premium services.

Since the delay requirements from various applications can differ greatly, the fairness principle should normalize the impact of different delay requirements. We define the *normalized delay* of a packet as the packet's delay divided by the delay bound required by the flow which that packet belongs to. Therefore, we claim that a fair scheduler for soft real-time flows with (m, k) -firm deadlines should reduce the failure rate of flows and serve packets with equal normalized delays (as far as possible).

We now analyze the fairness of some existing schedulers which are used to provide QoS assurance to soft real-time traffic. The *Earliest Deadline First* (EDF) scheduler achieves the minimum actual delay bounds of packets by transmitting the packet with the earliest deadline in the system. Clearly, the EDF scheduler does not consider the impact of packet loss or a missed deadline on the QoS for a flow. Therefore, it cannot satisfy the requirements of scheduling soft real-time traffic. Fair queueing schedulers control the transmission delay by the amount of bandwidth assigned to each flow. Similar to the EDF scheduler, they do not take the packet loss rate into account for QoS.

The DBP scheduler [60] combines packet loss rates and delay constraints. However, it cannot achieve a low failure rate and equal normalized delay at the same time. To illustrate

its unfair scheduling discipline, consider a system of two flows, A and B , with periodic arrivals as in most real-time traffic. Assume both flows stay at states with the same distance of one to a failing state, which indicates both flows have priority 1 at the moment. Suppose flow A 's head packet has a deadline earlier than flow B 's head packet. Suppose that flow A 's head packet misses the deadline by a small amount. The system will update flow A 's state and promote flow A to a higher priority, i.e. priority 0 in this case. Now, the new head packet of flow A can be transmitted immediately. However, if the previous head packet of flow A missed its deadline only by a small amount, the current head packet in flow A may still have enough time to meet its deadline while the head packet in flow B may have very limited time to meet its deadline. In this scenario, after flow A is promoted to a higher priority, its packets are served with very low delay. Packets from flow B , however, will experience long delays until flow A has no packet waiting for service. The DBP scheduler, therefore, does not achieve the desired fairness in the delays experienced by different flows.

There are several variants of the DBP scheduler proposed in [61–63] which attempt to reduce the work complexity of the DBP scheduler. A different approach which incorporates delay and loss constraints is proposed in [64], known as *Dynamic Window-Constrained Scheduling* (DWCS). This method uses fixed windows instead of sliding windows to measure the packet loss rate. This notion can be used to schedule flows whose packets have known logical relationships and can be grouped into segments. Therefore the QoS described by it forms a subset of those by the (m, k) constraint. Despite this difference in the constraints, the DWCS scheduler uses a similar method to determine the scheduling order. It gives a higher priority to flows with a tighter window-constraint. In addition, it alleviates the unfairness in the priority queueing discipline by constantly updating the flow states. However, it does not take any action to further tune the transmission order so as to achieve better fairness.

We propose two packet schedulers for soft real-time traffic that achieve the fairness goal while also reducing the number of packets that are dropped. The first scheduler uses

the method of priority queueing with promotions to reduce the unfairness in the traditional priority queueing method. The second one consists of a mechanism for maintaining a transmission timetable and a conditional priority queueing scheduler. It further improves the fairness achieved in scheduling soft real-time traffic.

4.2 SRTS-PQP: Soft Real-Time Scheduler Using Priority Queueing with Promotions

According to the discipline of priority queueing, a flow with low priority can transmit its packets only if no high priority flow is backlogged. Therefore, a low priority flow cannot update its state if high priority packets are waiting for service. Furthermore, if high priority queues are continuously backlogged, low priority flows cannot be promoted to a higher priority even though their packets have missed the deadlines. Since the DBP scheduler uses priority queueing scheduling, the probability of failure is greatly affected by the traffic arrival characteristics. Some flows may end up with higher failure probability than others simply because they encounter a bursty period from flows with higher priorities.

One way to solve the problem just mentioned is to update states of flows more frequently than traditional priority queueing. With additional promotion chances, it is possible for a low priority flow to acquire a higher priority before its packets have been delayed for too long. In the following, we call this method *Priority Queueing with Promotions* (PQP). A somewhat similar method is used in a different context within the DWCS scheduler [64].

In designing a scheduler to exploit the above method, we maintain the dynamic priority assignment approach in the DBP scheduler and replace the traditional priority queues with the PQP queues. As a result, the scheduler is able to serve soft real-time traffic with a reduced probability of failure (packet loss) and relatively lower delays in comparison to the DBP scheduler. We name this scheduler as *Soft Real-Time Scheduler with PQP* (SRTS-PQP), which we describe in greater detail below.

The SRTS-PQP scheduler attempts to reduce the probability of packet losses by giving

```

Dequeue:
  while (the system is not empty) do
     $H \leftarrow$  the nonempty queue with the highest priority;
     $Q \leftarrow$  the packet with the earliest deadline in  $H$ ;
    TransmitPacket( $Q$ );
    Update the related flow state;
    Update the related priority queue;
    Update TreeOfDeadlines;
    while (TreeOfDeadlines.earliestD <  $t$ ) do
       $F_i \leftarrow$  the flow with the earliest deadline in TreeOfDeadlines;
      Drop the head packet of queue  $i$ ;
      Remove  $F_i$  from the original priority queue;
      Update  $F_i$ 's state and calculate its new priority;
      Insert  $F_i$  to the right priority queue;
      Update TreeOfDeadlines;
    end while;
  end while;

```

Figure 4.1: Pseudo-code of *Dequeue* routine in the SRTS-PQP scheduler

promotion chances to low-priority flows. The scheduler consists of the data structure for the DBP scheduler. After each transmission, the scheduler checks for any packets that have missed their deadline. If a flow just has an overdue packet (i.e., it has missed its deadline), that flow should have a higher priority. The scheduler promotes such flows to higher priorities and then begins the next transmission from the currently highest priority flows. In order to find overdue packets in a short amount of time, the scheduler maintains a binary search tree, *TreeOfDeadlines*, of all head packets sorted by their deadlines. Whenever the head packet with the earliest deadline in the system is overdue, the scheduler drops the packet, updates the state and priority of that flow, and adds the deadline of its new head packet to *TreeOfDeadlines*. Then, among those packets with the highest priority, the scheduler selects one with the earliest deadline and begins to transmit this packet. The pseudo-code of the dequeuing routine is shown in Figure 4.1.

The SRTS-PQP scheduler has a per-packet work complexity of $O(\log N)$, where N is the number of flows in the system.

4.3 SRTS-CPQ: Soft Real-Time Scheduler Using Conditional Priority Queueing System

Even with the improvement devised in the SRTS-PQP scheduler, a higher priority packet is still transmitted earlier than a lower priority packet even though there is enough time to transmit the lower priority packet first and ensure a lower overall loss rate. As a result, packets from lower priority flows have long delays and these flows become closer to failing states unnecessarily, which in turn results in a higher probability of failure and higher normalized delay. To avoid this inefficiency, we propose a different scheduler using conditional priority queueing to achieve better fairness. We call this scheduler *Soft Real-Time Scheduler using Conditional Priority Queueing* (SRTS-CPQ). In the SRTS-CPQ scheduler, the flows also have (m, k) -firm deadline requirements. With the same definition of the state of a flow in the DBP scheduler, we further categorize flow states into three groups. When the number of packets transmitted before the deadline is less than m among past k packets, the flow is said to be in a *failing state*. A flow in the *edge state* is at a distance of one to the failing state. A flow in the *inner state* is at a distance larger than one to the failing state. The SRTS-CPQ scheduler uses a new discipline of conditional priority queueing to organize the transmission order based on these flow states. It assigns higher conditional priority to failing state flows than edge state flows, and similarly, a higher priority to edge state flows than inner state flows.

4.3.1 Conditional Priority Queueing

Conditional priority queueing is based on a simple principle that a higher priority flow should not necessarily be favored over a lower priority flow at all times. When the traffic load is light, a flow with low priority is treated equal to a flow with higher priority, i.e.,

the flow with lower priority can transmit packets before the higher priority one if the transmission will not result in any dropped packets from the higher priority ones. In general, we say that an edge state flow has higher *conditional priority* than an inner state flow. To simplify the discussion, in the rest of this dissertation, when we say “high priority”, we will mean high conditional priority. In the SRTS-CPQ scheduler, flows at failing states have the highest conditional priority and flows at inner states have the lowest.

4.3.2 System Structure

The primary goal in maintaining conditional priority queueing is to ensure a fair distribution of the opportunity to transmit to both the higher and lower priority packets. To achieve this goal, the system keeps a record of the latest transmission time for packets with higher priority. With the knowledge of the latest time to transmit a higher priority packet without causing a packet loss, the scheduler can readily determine whether or not to transmit a packet with lower priority. Thus, the system sorts the head packets from all flows with low and high priorities based on their deadlines and attempts the transmission of the packet with the earliest deadline. If the packet with the earliest deadline has higher priority, it is transmitted. Otherwise, the scheduler examines the latest time that a higher priority packet transmission may begin without causing a loss, and checks whether the transmission of the lower priority packet would delay the transmission of higher priority packets. If it will, then the lower priority packet is dropped, the associated flow updates its priority and obtains the correct position for its new head packet. If not, the lower priority packet is then transmitted.

To ensure efficiency in the operation of the scheduler, packets are sorted using a minimum heap, called *DeadlineHeap*. A record of the latest transmission times of high priority packets is updated and maintained in a table with entries corresponding to these packets. This table, henceforth called the *timetable*, is not arranged by simply sorting the deadlines

of high priority packets. Since the length of any transmission is not negligible, transmitting a packet at the last moment may cause other packets with later deadlines to be dropped. Therefore, it is necessary to appropriately arrange the transmissions of all higher priority packets. Here, we propose a data structure which can accomplish the function of the timetable in $O(\log N)$ time. The data structure is described in Section 4.3.3. The packet with the earliest deadline is selected first. A packet with low priority can be transmitted only if its transmission will not result in a higher priority packet missing its deadline. This is determined by looking up the timetable. If the earliest starting deadline in the timetable is later than the finishing time of the transmission, that packet can be sent. Otherwise, the packet is dropped.

Once a packet is sent or dropped, the flow it belongs to should update its state information. The new head packet of this flow is then inserted into the *DeadlineHeap*. If this flow is in edge state, then the timetable should also be updated to include the deadline of the new head packet. The pseudo-code for the SRTS-CPQ scheduler is presented in Figures 4.2, 4.3 and 4.4. In the pseudo-code, the current time is denoted by t . The reader is referred to Appendix B for a description of the routines used by the pseudo-code.

4.3.3 Maintaining the Timetable

Structure of the Timetable

To describe the algorithm used to maintain the timetable, we first introduce some definitions. The priority of a packet is determined based on the flow state. In the SRTS-CPQ scheduler, there are a total of three (3) levels of priority, namely priority 0, 1 and 2. Flows in the failing state have priority 0, which is the highest priority. Flows in the edge state have priority 1 and flows in the inner state have priority 2. Priorities 0 and 1 are the same as in the DBP scheduler [60], while all priorities lower than and equal to 2 are combined into priority 2.

```

Enqueue: (Invoked when  $P$  arrives at flow  $F_i$ )
Assign deadline to  $P$ ;
if (Queue  $i$  is empty) then
    Update the state of flow  $F_i$ ;
    if ( $F_i.priority \leq 1$ ) then
         $T_1.InsertPacket(P)$ ;
        if ( $F_i.priority = 0$ ) then
             $T_0.InsertPacket(P)$ ;
        end if;
    end if;
     $AddPacketToDeadlineHeap(P)$ ;
end if;
 $AddPacketToQueue(i, P)$ ;

```

Figure 4.2: Pseudo-code of *Enqueue* routine in SRTS-CPQ

The *deadline* of a packet x , denoted as D_x , is the latest time when the transmission of packet x should complete. The *starting deadline* of packet x , SD_x , is defined as the latest time when the transmission of packet x should begin to meet the deadline. Then, $SD_x = D_x - L_x/R$, where L_x is the length of this packet, and R is the link capacity. The timetable checks the deadlines of high priority packets, arranges the latest transmission time of high priority packets in order to efficiently determine the latest time for the system to start transmitting an edge state packet. Since the deadlines can be spread over time, we design a data structure, described later, to maintain the tracking information efficiently and to update the timetable quickly.

The purpose of maintaining a timetable for high priority packets is to provide the information on the latest time to start a transmission of a high priority packet. Since there can be more than one packet with high priority, and their deadlines may not be far apart enough to fit the transmission of the packet with the later deadline, the scheduler should be able to arrange the transmission deadlines so that enough time is left for the transmission of all


```

Dequeue:
while (DeadlineHeap is not empty) do
   $P \leftarrow \text{MinOfDeadlineHeap};$ 
  if ( $t + P.\text{TxTime} \leq P.D$ ) then
    if ( $P.\text{priority} = 0$ ) then
      TransmitPacket( $P$ );
    else if ( $P.\text{priority} = 1$  AND ( $T_0$  is empty OR  $t < T_1.\text{earliestSD}$ )) then
      TransmitPacket( $P$ );
    else if ( $P.\text{priority} = 1$  AND  $T_0$  is not empty AND  $t + P.\text{TxTime} < T_0.\text{earliestSD}$ ) then
      TransmitPacket( $P$ );
    else if ( $P.\text{priority} = 1$  AND  $T_0$  is not empty AND  $t + P.\text{TxTime} \geq T_0.\text{earliestSD}$ ) then
      DropPacket( $P$ );
    else if ( $Q.\text{priority} > 1$  AND  $T_1$  is empty) then
      TransmitPacket( $P$ );
    else if ( $Q.\text{priority} > 1$  AND  $T_1$  is not empty AND  $t + P.\text{TxTime} < T_1.\text{earliestSD}$ ) then
      TransmitPacket( $P$ );
    else if ( $P.\text{priority} > 1$  AND  $T_1$  is not empty AND  $t + P.\text{TxTime} \geq T_0.\text{earliestSD}$ ) then
      DropPacket( $P$ );
    end if;
  else /*  $P$  will miss its deadline */
    DropPacket( $P$ );
  end if;
end while;

```

Figure 4.3: Pseudo-code of *Dequeue* routine in SRTS-CPQ

high priority packets.

To facilitate further discussion of the SRTS-CPQ scheduler, we define several terms in the following:

Definition 9 A packet of priority p^* is a packet from a flow with a priority higher than or equal to p . If packet x is of priority p^* and $k \geq 1$, an assembly of k packets of priority p^* beginning at packet x , denoted by $y_p(x, k)$, consists of packet x and $k - 1$ other packets which satisfy the following conditions:

1. Each packet is of priority p^* and has a deadline later than packet x .
2. If the k packets of $y_p(x, k)$ are numbered as $x, x + 1, \dots, x + k - 1$ in order of their

```

TransmitPacket(P):
  Transmit  $P$ ;
  if ( $P.priority = 0$ ) then
     $T_0.ReleasePacket(P)$ ;
     $T_1.ReleasePacket(P)$ ;
  else if ( $P.priority = 1$ ) then
     $T_1.ReleasePacket(P)$ ;
  end if;
  Update the related flow state and DeadlineHeap;
  Update  $T_0$  and  $T_1$  if necessary;

DropPacket(P):
  Remove  $P$  from its queue;
  Update the state of its flow and DeadlineHeap;
  Update  $T_0$  and  $T_1$  if necessary;

```

Figure 4.4: Pseudo-code of *TransmitPacket* and *DropPacket* routines in the SRTS-CPQ scheduler

deadlines, there exists a relation on their deadlines and packet lengths as

$$D_{x+k-1} - D_x \leq \sum_{i=x+1}^{x+k-1} L_i. \quad (4.1)$$

From Definition 9, an assembly is a group of packets whose deadlines are close to each other. If the system only begins to transmit packet x at SD_x , some deadlines of packets in $y_p(x, k)$ may be missed. In the following, we define some of the properties of assemblies that we will use in our algorithm description and analysis.

Definition 10 *The size of an assembly $y_p(x, k)$, denoted by $|y_p(x, k)|$, is defined as the number of packets contained in it, i.e.*

$$|y_p(x, k)| = k.$$

The length of an assembly $y_p(x, k)$, $L\{y_p(x, k)\}$, is the total length of the packets contained

by $y_p(x, k)$, i.e.

$$L\{y_p(x, k)\} = \sum_{i=x}^{x+k-1} L_i.$$

The deadline of an assembly $y_p(x, k)$, $D\{y_p(x, k)\}$, is the latest deadline among all packets in $y_p(x, k)$, which is actually the deadline of packet $x + k - 1$, i.e.

$$D\{y_p(x, k)\} = \max_{x \leq i \leq x+k-1} D_i = D_{x+k-1}.$$

The starting deadline of an assembly $y_p(x, k)$, $SD\{y_p(x, k)\}$, is the latest time to start transmission of packets of $y_p(x, k)$ so as to meet their deadlines. $SD\{y_p(x, k)\}$ is obtained by subtracting the length of the assembly from the deadline of the assembly, i.e.

$$SD\{y_p(x, k)\} = D\{y_p(x, k)\} - L\{y_p(x, k)\}.$$

According to Definition 9, assemblies with different sizes can begin at the same packet. For each packet, we call the assembly with the maximum size as *max-assembly*. We define it formally as follows.

Definition 11 A max-assembly of priority p^* beginning at packet x , $Y_p(x)$, is the one with the maximum size among those assemblies of priority p^* beginning at packet x .

Figure 4.5 shows an example of assemblies beginning at packet x . Now we can partition a group of packets into a set of assemblies according to Definitions 9 and 11. Here we are interested in a special set of max-assemblies which includes all packets in the group and such that, any given packet is contained in one and only one max-assembly. This set is constructed through the following procedure. Let \mathbf{A} be a group of k packets of priority p^* . Each of them is numbered according to the order of their deadlines, and the packet with the earliest deadline is numbered as 1 while the one with the latest deadline as k . $\mathbf{Y}_p^{\min}(\mathbf{A})$ is a set of max-assemblies of \mathbf{A} which contains all packets in \mathbf{A} . If $\mathbf{Y}_p^{\min}(\mathbf{A}) = \{Y_p(a_1), Y_p(a_2), \dots, Y_p(a_m)\}$, where a_1, a_2, \dots, a_m are the indices of the

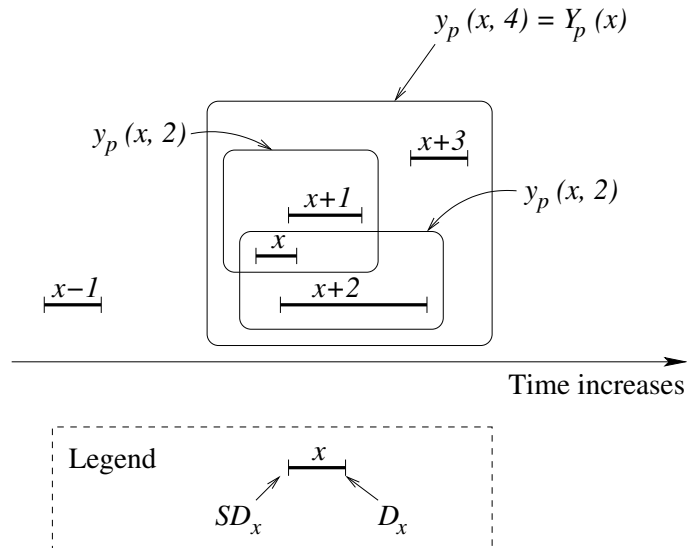


Figure 4.5: An illustration of assemblies beginning at packet x

beginning packets and satisfy

$$\begin{cases} a_1 = 1 \\ a_i = a_{i-1} + |Y_p(a_{i-1})|, 2 \leq i \leq m \\ a_m + |Y_p(a_m)| - 1 = k, \end{cases} \quad (4.2)$$

$\mathbf{Y}_p^{\min}(\mathbf{A})$ is called the *max-assembly partition* of \mathbf{A} .

As long as each max-assembly is served no later than its starting deadline, it is possible for the system to meet all high priority deadlines and at the same time serve low priority packets whenever possible.

As packets are arriving and being transmitted, $\mathbf{Y}_p^{\min}(\mathbf{A})$ changes constantly. In order to track $\mathbf{Y}_p^{\min}(\mathbf{A})$ and therefore to compute the latest starting time, we propose a data structure of a timetable. A timetable consists of *occupied periods* and *vacant periods*, definitions of which are presented below.

Definition 12 Let \mathbf{A} be the set of all packets of priority p^* awaiting service in the SRTS-CPQ system. $\mathbf{Y}_p^{\min}(\mathbf{A}) = \{Y_p(a_1), Y_p(a_2), \dots, Y_p(a_m)\}$ denotes the max-assembly por-

Table 4.1: Variables in a *VacancyElement*

b	The beginning time of the vacant period
e	The ending time of the vacant period
len	The length of this vacant period
$leftLen$	The total length of vacant periods in its left subtree
$rightLen$	The total length of vacant periods in its right subtree
Pointers	Used for data structure maintenance
$pktLen$	The total length of packets sitting between this <i>VE</i> and its predecessor.

tion of \mathbf{A} whose elements satisfy the relations in (4.2). The occupied period of $Y_p(a_i)$ is the latest possible transmission time for $Y_p(a_i)$, for $1 \leq i \leq m$. The vacant period of $[Y_p(a_{i-1}), Y_p(a_i)]$ is the time interval between the occupied periods of $Y_p(a_{i-1})$ and $Y_p(a_i)$.

The timetable of priority p , denoted as T_p , consists of occupied periods of $Y_p(a_i)$ from $\mathbf{Y}_p^{\min}(\mathbf{A})$, $1 \leq i \leq m$, and vacant periods of $[Y_p(a_{i-1}), Y_p(a_i)]$, $2 \leq i \leq m$. T_p starts with the earliest occupied period of all head packets with priority p^* , and ends with the latest occupied period among the same group of packets. In the middle of T_p , occupied periods and vacant periods are interleaved with each other. To keep track of the timetable, the basic unit in the data structure is an object named *VacancyElement*, or in short *VE*. Each *VE* represents a vacant period and the occupied period before it. The object contains several variables, which are listed in Table 4.1. The vacant period starts at $VE.b$ and ends at $VE.e$. The occupied period before it can be obtained from $VE.b$ and $VE.pktLen$. To search fast among vacant periods, we implement a binary search tree of *VEs*, called *VacancyTree*. Necessary pointers are stored in *VEs*. To facilitate the appropriate tracking of the timetable, each *VE* also contains the length of the vacant period within itself and the total length of the vacant periods within its left and right subtrees.

Besides *VacancyTree*, the timetable keeps records of the beginning and ending time in *earliestSD* and *latestD* respectively. Since *VEs* in *VacancyTree* only contains the occupied periods before each vacant period, the timetable needs to track the last occupied period

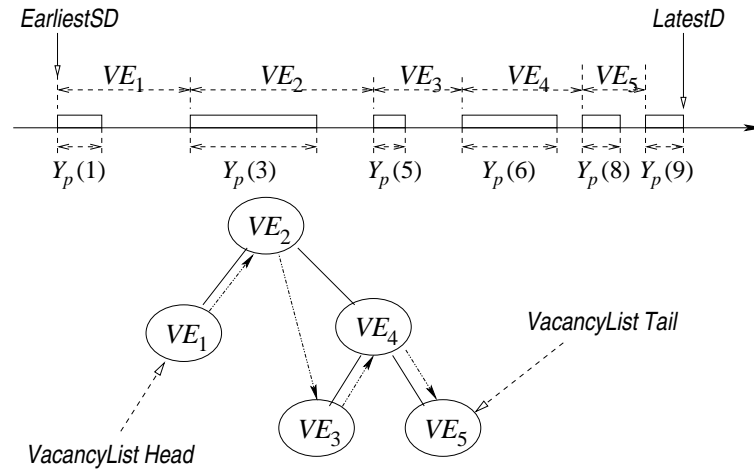


Figure 4.6: An illustration of the structure of a timetable of priority p . The size of each max-assembly is: $|Y_p(1)| = |Y_p(3)| = |Y_p(6)| = 2$, $|Y_p(5)| = |Y_p(8)| = |Y_p(9)| = 1$

within it, which is done by recording the total length of packets behind the last VE , denoted as $pktLenAtTail$. Figure 4.6 illustrates the structure of a sample timetable.

Arranging the Timetable

Arranging the timetable includes inserting a packet and releasing a packet. When we insert a packet into the timetable, the system first locates the occupied period for this packet. Let this occupied period be E . The length of E is increased and therefore the vacant periods before E will be shrunk. As a result, some vacant periods may disappear and the occupied periods between them will be combined into one long occupied period. The insertion operation consists of a chain of adjustment among VE s. On the other hand, releasing a packet is much easier to finish. A packet is released from the timetable when it is transmitted or dropped. In either case, the packet is within the earliest occupied period. Since the length of a period is the only value of concern in the timetable, it suffices to only adjust the length of the first occupied period and the value of $earliestSD$ accordingly.

From the scenario of inserting a packet, one needs an efficient way to finish the chain

of adjustments. Since *VacancyTree* is a binary tree structure, it has a potential to finish these adjustments with $O(\log N)$ work complexity. There are two cases as regards how the adjustments are made, depending on whether or not the *VacancyTree* is empty. In the following, we analyze and explain these cases in more detail.

Case 1: *VacancyTree* is empty, which means the timetable has up to one occupied period. If the timetable is empty, i.e. no occupied period at all, an occupied period is created. If one occupied period exists, two possible actions may be taken depending upon the deadline of the inserted packet. If the starting deadline falls within the existing occupied period, the system only adjusts the length of that period. Otherwise, a *VE* is generated and *VacancyTree* becomes non-empty. Finally, *earliestSD* and *latestD* are updated accordingly.

Case 2: *VacancyTree* is not empty. The system first locates the position of the inserted packet in the timetable. If the latest possible transmission time is outside the range of *VacancyTree*, a new *VE* may be created if needed and related updates are accordingly made. However if the latest possible transmission is within the range of *VacancyTree* and specifically inside the range of one *VE*, denoting that *VE* as *V*, the system starts a search along the tree from *V*. The search procedure contains up to two stages, searching upward and downward. The search stops completely when it finds enough amount of vacant time to hold the inserted packet or when it finds out that the vacant time in *VacancyTree* is not enough to hold the inserted packet. In the latter case, the packet is inserted into the timetable temporarily and all *VEs* before *V* are deleted. The *earliestSD* is pushed to an earlier time. If the new *earliestSD* is earlier than current time, the packet dropping mechanism is triggered. A fair dropping mechanism is discussed later.

Now we look into the two stages of the search action. In the first stage the system searches upward in the *VacancyTree*. Since the inserted packet may advance the starting deadline of some max-assembly with earlier time, only those vacant periods before *V* can be affected. By searching upward, the system finds out the highest level in the tree to be adjusted and determines whether to search downward. If the *VE* at the highest level

provides enough vacancy, the search may stop. Otherwise, the system starts to search downward in the left subtree. Once the system finds the earliest VE , denoted as U , whose vacant period together with all vacant periods after it but before V can accommodate the expansions on max-assemblies caused by the inserted packet, all those vacant periods are deleted from $VacancyTree$. The occupied periods among those deleted vacant periods are left and combined into one occupied period.

Deleting nodes from a binary tree one by one is certainly not an efficient method. Actually, during the search action, the system can determine which node and/or subtree should be deleted. The complexity of such deletion is $O(\log m)$ if m is the total number of VEs . This is proved in the following lemma.

Lemma 2 *When inserting a packet into the timetable, if at least one VE is deleted during the search action, at most one VE is deleted with $O(\log m)$ time, where m is the number of VEs in $VacancyTree$, while all other VE deletions take $O(1)$ time.*

Proof: The proof is better explained with the search action. When searching upward along the tree branch, each VE is examined on $VE.len$ and $VE.leftLen$. Suppose the inserted packet needs additional ΔT to accommodate its transmission. The result of the examination may be one of the following cases:

1. If $VE.len > \Delta T$, VE is not deleted.
2. If $VE.len = \Delta T$, only VE is deleted and the search stops.
3. If $VE.len + VE.leftLen > \Delta T$, the upward search stops and the downward search starts while ΔT is reduced by $VE.len$. VE is deleted in this case.
4. If $VE.len + VE.leftLen = \Delta T$, VE and its left subtree are deleted. The search stops.
5. If $VE.len + VE.leftLen < \Delta T$, VE and its left subtree are deleted. After reduce ΔT by the sum of $VE.len$ and $VE.leftLen$, the upward search moves on to the parent of VE .

Now we consider a VE on the route of downward search. Similarly ΔT represent the additional time needed to accommodate the inserted packet. The result of the examination is one of the following cases:

6. If $VE.rightLen > \Delta T$, the downward search moves on to the right child of VE .
7. If $VE.rightLen = \Delta T$, the right subtree of VE is deleted and the downward search stops.
8. If $VE.rightLen < \Delta T$ and $VE.rightLen + VE.len > \Delta T$, same as case 7.
9. If $VE.rightLen + VE.len = \Delta T$, both VE and its right subtree are deleted. The downward search also stops.
10. If $VE.rightLen + VE.len < \Delta T$, both VE and its right subtree are deleted. The downward search moves on to the left child of VE .

Clearly, throughout the search operation, only the VE at the highest level may be deleted by itself alone, as in case 2 and 3. In other cases, a VE is deleted with either its left or right subtree, which can be finished within $O(1)$ time. And deleting a node from a binary tree only needs $O(\log m)$ if m is the total number of nodes in the tree. Thus the lemma is proved. ■

Lemma 3 *When inserting a packet into the timetable, deleting used vacant periods is done in $O(\log m)$ time, where m is the total number of nodes in the VacancyTree.*

Proof: Suppose that the search starts at node V and stops at node U . Since the search route is along the shortest path between V and U , the number of nodes examined is of $O(\log m)$. Therefore, the number of node deletions is also $O(\log m)$. As illustrated in the proof for Lemma 2, at most one node deletion among those examined nodes requires $O(\log m)$ time. Each of the rest of the node deletions is done in $O(1)$ time. Therefore, the total time needed to delete used vacant periods is of $O(\log m)$. ■

4.3.4 Dropping Packets

From previous discussions, in the SRTS-CPQ scheduler, a packet with a lower priority may be dropped if its transmission conflicts with the starting deadlines of max-assemblies of packets with a higher priority. The reason for this kind of dropping is to reduce the probability of dynamic failure according to the (m, k) -firm deadline requirements. The decision is made at the boundary of packet transmissions when the scheduler is searching a packet for the next transmission opportunity. The principle of dropping under this circumstance is straightforward: if a lower priority packet cannot finish its transmission before the earliest starting deadline of higher priority packets, it is dropped.

Such a conflict in transmissions may also emerge through the arrangement of the timetable. After inserting a packet into the timetable, the earliest starting deadline may be pushed ahead. If the new earliest starting deadline is even earlier than current time, some packet would have to be dropped to ensure other packets' chances of meeting deadlines. Under this circumstance, it is critical to ensure fairness in the selection of a packet to drop. Here, we do not apply a complex dropping scheme; the scheduler just attempts to transmit the packet with the earliest deadline in the timetable. Once it finds out the packet from the timetable cannot be transmitted before its deadline, that packet is dropped and the scheduler proceeds onto the next packet in the timetable.

To achieve fairness in the probability of dynamic failure, we need a fair dropping discipline to correctly select a packet to drop. A possible approach is to maintain a record of past dropping history for each flow. Once it is determined that a packet should be dropped, the system selects the flow with the lowest dropping rate and the head packet of that flow bearing a deadline within the first occupied period in the timetable. However, this approach cannot guarantee that dropping the selected packet can ensure that the rest of the packets will meet their deadlines. It is possible that the selected packet and the packets with earlier deadlines do not form any assembly. Thus, the earliest starting deadline may not be post-

poned enough after dropping that packet. As a result, the system may need to drop another packet hoping that *earliestSD* would retreat to an instant later than current time. This will take $O(N)$ time and therefore, is not a scalable option.

4.4 Analysis of the SRTS-CPQ Scheduler

Lemma 4 *In the SRTS-CPQ scheduler, it takes $O(\log m)$ time to insert a packet into or release a packet from a timetable, where m is the total number of nodes in the *VacancyTree*.*

Proof: As shown in Figure B.2, inserting a packet into a timetable requires one to locate the range of vacant periods to be occupied, delete the occupied vacant periods, and update *VacancyTree* as regards the vacancy length. The first two operations are actually done at the same time and only take $O(\log m)$ time to finish since the locating procedure is along the shortest path between the earliest and latest *VEs* within *VacancyTree*.

To consider the procedure of updating the rest of *VacancyTree* on vacancy length, we assume that, among those *VE* affected by the insertion, w is at the highest level in the tree. Once the range of vacant periods is located, the total length of vacancy which becomes occupied is recorded during the locating procedure. Only the ancestor nodes of w should update their records of the vacancy length within their subtrees. Such updating can be completed by traversing the tree along the shortest path from w to the root, and thus only takes $O(\log m)$ time to finish. Within the subtree rooted at w , only the ancestors of the affected *VEs* have to adjust their records on vacancy lengths which can be finished during the search procedure. Since the search procedure is done within $O(\log m)$ time, updating records on vacancy length is also on the same order. This proves the lemma. ■

The DBP scheduler, on the other hand, consists of two major parts, one to decide the priority of each flow and one to sort the deadlines within each priority. The priority of a flow is directly related to the transmission history. A record of it is maintained at all times and updated only at the time when a packet is transmitted or dropped. Thus it has $O(1)$ per

packet work complexity.

The other part of the DBP scheduler is basically a priority queueing scheduler. However within each priority, the service order is determined by the earliest deadline first discipline. Maintaining the sorted order of packets based on deadlines has $O(\log k)$ complexity if k packets have the same priority. In the worst case, k equals to N . On the other hand, when a packet is dropped because its deadline is missed, the system will need to update the state of the related flow and insert the new head packet into the sorted order of packets with the same priority. Thus the complexity of dropping depends on the number of packets dropped. Since the incoming traffic is not known, one cannot estimate how many packets are awaiting service in a higher priority queue, and therefore the number of packets dropped from a lower priority queue cannot be predicted. Thus, the complexity of the scheduler is unbounded.

Theorem 6 *In a SRTS-CPQ scheduler for a system with N flows, the per-packet processing complexity is $O(\log N)$.*

Proof: Processing a packet consists of enqueueing and dequeueing routines. In the enqueue routine, when a packet arrives, it is added to the queue for its flow and its deadline is assigned based on that flow's traffic parameter. If the flow is previously empty, the flow state is reset and this packet is inserted into *DeadlineHeap*. And the same packet may be inserted to the timetable of the scheduler if its flow has a high priority. Since the insertions of *DeadlineHeap* and the timetable are of $O(\log N)$ each, the enqueue routine of the SRTS-CPQ scheduler has a work complexity of $O(\log N)$ for each arriving packet.

In the dequeue routine, the packet with the earliest deadline is taken from *DeadlineHeap*. The scheduler checks the priority and transmission time of the packet and determines whether to send or drop the packet. If the scheduler decides to transmit the packet, then after the transmission the scheduler should update the flow state, delete the transmitted packet from the data structure, and add the new head packet, if there is a new one, into

the heap and the timetables. Since the deletion and the insertion can be finished within $O(\log N)$ time, transmitting a packet only takes $O(\log N)$ time. On the other hand, if the scheduler decides to drop the packet, it takes $O(\log N)$ time to update the heap and timetables. Therefore, the theorem is proved. ■

4.5 Discussions and Evaluation

4.5.1 Discussions

In this chapter, SRTS-PQP and SRTS-CPQ schedulers have been presented. Both schedulers improve the performance of the DBP scheduler. The SRTS-PQP scheduler uses a similar method as the Dynamic Window-Constrained Scheduling (DWCS) method. In comparison to the DBP scheduler, the SRTS-PQP scheduler reduces the probability of dynamic failure for flows and, in addition, its work complexity is not high. However, since the scheduling decision is based on priority queueing discipline, it cannot avoid the unfairness inherent in priority queueing.

The SRTS-CPQ scheduler uses a new scheduling discipline, conditional priority queueing. It provides a framework for treating real-time traffic with both QoS and fairness requirements. Even though the data structure used for SRTS-CPQ is not simple, its per-packet work complexity is $O(\log N)$ if totally N flows are sharing one output link.

4.5.2 Evaluation Based on Simulations

We use simulation experiments to evaluate the performance of the SRTS-CPQ scheduler. The comparison is made among SRTS-CPQ, DBP and SRTS-PQP schedulers. In the simulation, each scheduler is fed by traffic from six Voice-over-IP (VoIP) flows and the probability of dynamic failure is recorded. The VoIP traffic traces used here are obtained from a public data made available by a research project on modeling real-time multimedia traffic [65]. The packet lengths and generation time instants were recorded by VoIP appli-

Table 4.2: Settings for traffic sources from VoIP traces

Source	1	2	3	4	5	6
L_{avg} (bytes)	36	36	36	36	32 or 17	36
r_{avg} (Bps)	460	812	98	810	807	727
r_{peak} (Bps)	936	1002	1234	1050	1062	1188
ΔD (ms)	10	10	10	10	10	10
Link Capacity	3.75×10^3 Bps					
Total Time	50 seconds					

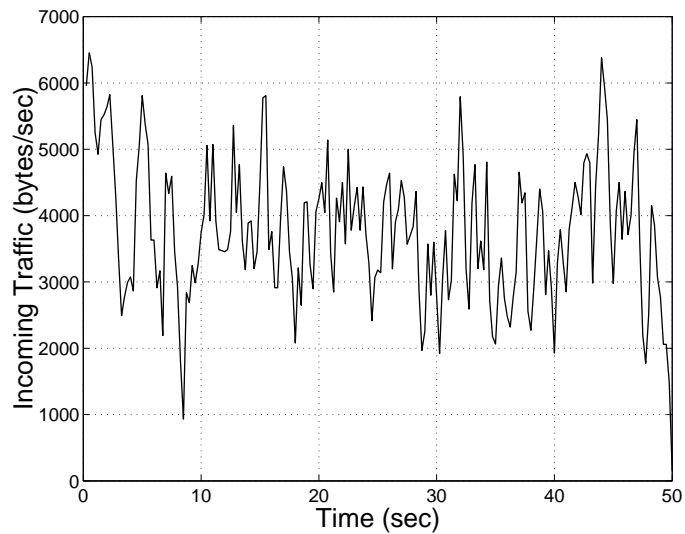


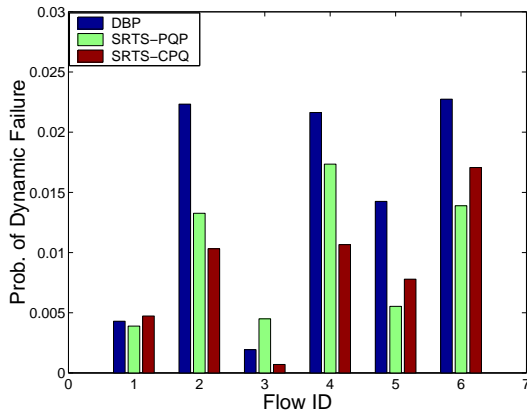
Figure 4.7: Incoming traffic throughout the simulation.

cation softwares. Table 4.2 lists the traffic traces used in our simulation. As shown in the table, most flows have constant-sized packets except for Flow 5. Figure 4.7 plots the total rate of incoming traffic throughout the simulation.

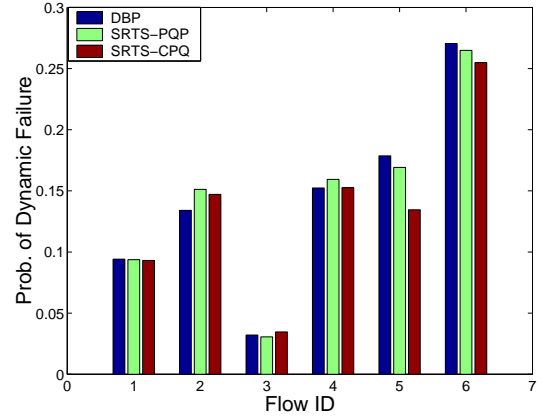
To show the validity of SRTS-CPQ, the value of (m, k) is chosen such that m is smaller than k by at least two. We present two sets of results. One is from a system with flows of $(2, 4)$ -firm deadlines and the other is with flows of $(6, 8)$ deadlines. Note that we do not reduce $(2, 4)$ to $(1, 2)$ for the purpose of SRTS-CPQ. We first compare the probability

of dynamic failure in each of the three schedulers. Figures 4.8 (a)–(d) show the failure probability of each flow under different traffic loads in both experiments. Figures 4.8 (e) and (f) show the average failure probability over all flows versus traffic load. As observed from those figures, SRTS-CPQ has the lowest failure probability with heavy traffic load. For flows with (2, 4)-firm deadlines, the average failure probabilities of the SRTS-PQP scheduler is close to that of the SRTS-CPQ scheduler, and even lower than that of SRTS-CPQ when traffic load is about 70%. Since (6, 8) deadlines are more stringent than (2, 4), failure probabilities with (6, 8) deadlines are higher than those with (2, 4) deadlines and the differences between schedulers are less. With (2, 4) deadlines, the SRTS-CPQ scheduler reduces the failure probability by about 40% from the DBP scheduler, while with (6, 8) deadlines the reduction is about 5%. However, the SRTS-PQP scheduler produces failure probabilities very close to the DBP scheduler under heavy traffic load with (6, 8) deadlines. This indicates that the SRTS-CPQ scheduler maintains a relatively good performance even with tighter loss and delay constraints.

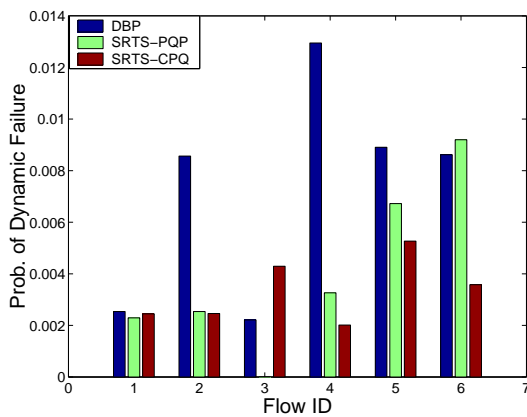
Now we study the average delay achieved by each of the schedulers. Figures 4.9(a)–(d) plot the average delay of each flow under different traffic loads. Here we also present results from the systems of (2, 4) deadlines and (6, 8) deadlines. In these graphs, the value of average delays is normalized by the guaranteed delay bound corresponding to the flows. As shown by these graphs, both SRTS-PQP and SRTS-CPQ reduce average delays in comparison to the DBP scheduler. SRTS-PQP achieves slightly lower delays than SRTS-CPQ. However, delays of different flows in the SRTS-PQP scheduler have larger differences between each other. To compare these differences with quantities, we compute the Gini index of every set of average delays and list the results in Table 4.3. One can see that SRTS-CPQ has a lower Gini index than SRTS-PQP. It verifies that SRTS-CPQ achieves a more fair distribution of the delays among the flows.



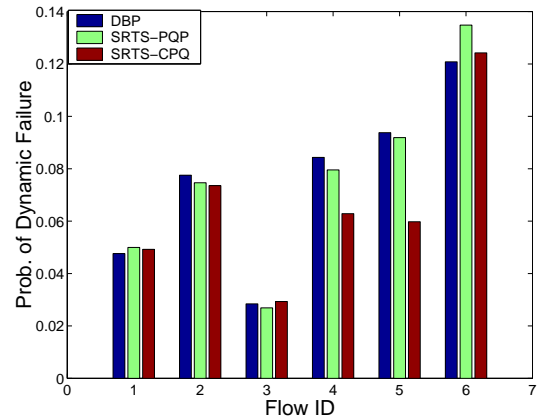
(a) (2, 4); Traffic load > 99%



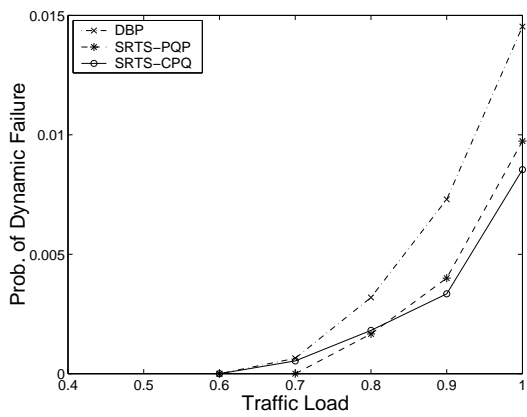
(b) (6, 8); Traffic load > 99%



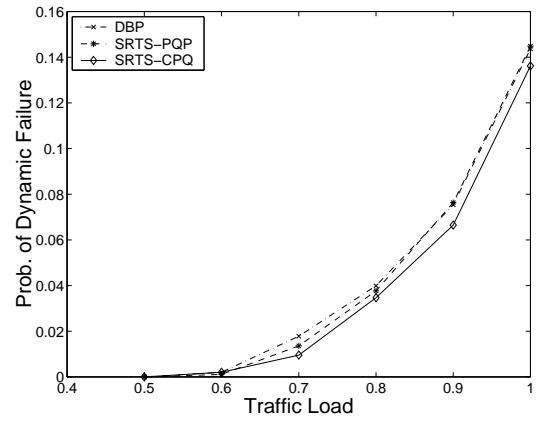
(c) (2, 4); Traffic load ≈ 90%



(d) (6, 8); Traffic load ≈ 90%



(e) (2, 4)-firm deadlines



(f) (6, 8)-firm deadlines

Figure 4.8: The probability of failure with different traffic loads

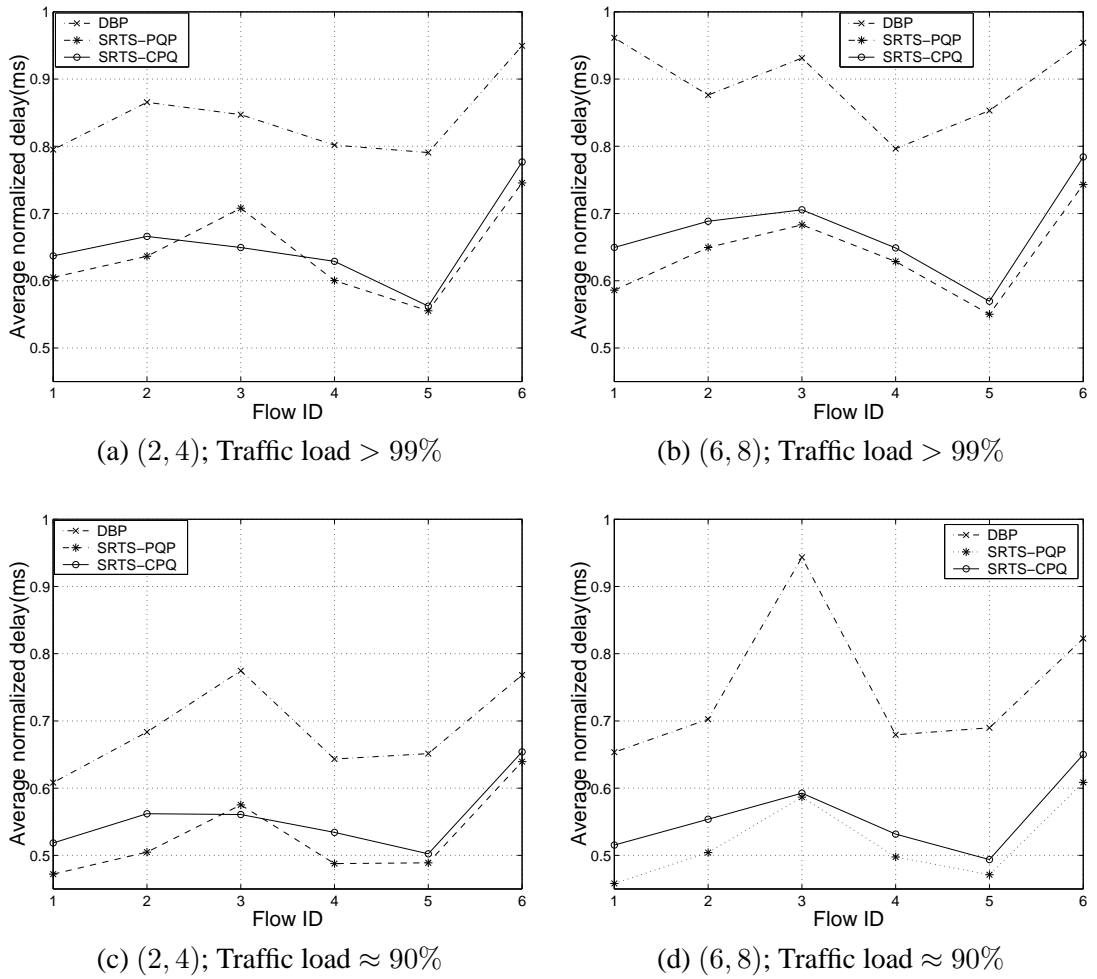


Figure 4.9: The average normalized delay of each flow with different traffic loads

Table 4.3: Gini index of average delays among flows

	(2, 4)-firm deadlines		(6, 8)-firm deadlines	
	Traffic > 99%	Traffic \approx 90%	Traffic > 99%	Traffic \approx 90%
DBP	0.0347	0.0500	0.0366	0.0702
SRTS-PQP	0.0565	0.0586	0.0554	0.0589
SRTS-CPQ	0.0509	0.0458	0.0528	0.0517

Chapter 5. Concluding Remarks and Future Work

5.1 Summary and Concluding Remarks

A number of emerging real-time and multimedia Internet applications will rely on scheduling algorithms in switches and routers to guarantee performance and an acceptable level of quality of service. Based on requirements of heterogeneous applications and benefits of the network, packet scheduling strategies are designed to satisfy both network users and network operators. This dissertation proposes novel packet schedulers for best-effort service, guaranteed (bandwidth and delay) services, controlled load service and soft real-time service. These service models and their variants cover the vast majority of services requested by applications today.

We have proposed a novel scheduler, *Greedy Fair Queueing (GrFQ)*, for fair queueing that can also serve as a low-latency scheduler for guaranteed bandwidth services. The per-packet dequeuing complexity of GrFQ is $O(\log N)$ with respect to the number of flows. This scheduler achieves better fairness (as measured by traditional metrics such as the normalized lag) than other schedulers of equivalent complexity. In this dissertation, we further argue that existing measures of fairness do not accurately capture the actual fairness achieved at most instants of time, and therefore, do not represent a true measure of the ability of a scheduler to successfully deliver end-to-end quality for real-time applications. To correctly evaluate the overall fairness performance, we borrow from the field of economics and propose a new measure of fairness based on the *Gini index*. This measure captures the instantaneous fairness of a scheduler and, unlike other measures based on bounds, also captures the fairness of the scheduler in its handling of flows during idle periods. With the Gini index as the measure of instantaneous fairness, we use real video traffic traces and real gateway traffic traces to show that the GrFQ scheduler achieves better fairness than

any other known scheduler at virtually all instants of time. We further propose a simplified version of the scheduler, called GrFQ-lite, which avoids the emulation of a fluid flow system and has a per-packet work complexity of $O(1)$ in the computation of the timestamps. Using real traffic traces again, we demonstrate that GrFQ-lite is also able to achieve close or better fairness than most other schedulers including those that are significantly more computationally intensive in their emulation of the ideally fair fluid-flow GPS system. The GrFQ and GrFQ-lite schedulers can be applied to switches and routers to achieve better bandwidth allocation among flows of traffic.

The Integrated Services framework, however, defines two kinds of services: guaranteed service and controlled load service. The GrFQ scheduler may also be used for guaranteed services (for both bandwidth guarantees and delay guarantees). This dissertation, therefore, next considers the controlled load service and develops a novel packet scheduler to meet the unique requirements of such a service model. The controlled load service requires source points to regulate the traffic and mark packets that are sent in violation of the traffic contract. One of the requirements we define is that the additional delay of unmarked packets caused due to the transmission of marked packets should be bounded. A $O(1)$ scheduler to achieve this bound is non-trivial. In this dissertation, we have proposed the $CL(\alpha)$ scheduler, which bounds this extra delay to α or less.

The principle used in the $CL(\alpha)$ scheduler may also be used to schedule flows with multi-level priorities, such as in some scalable real-time video streams as well as in other emerging service models of the Internet that mark packets to identify drop precedences [3, 38, 39]. In such cases with multiple levels of drop precedences, the principle of the $CL(\alpha)$ scheduler would have to be applied in a hierarchical manner to bound the impact of each lower priority layer on the delays experienced by higher priority layers. For example, consider flows of packets with three priority levels labeled as type 1, type 2 or type 3 packets, with type 1 at the highest priority level. In transmissions using such layered coding, one may get tolerable quality from receiving just type 1 packets. The quality of

the received video and audio deteriorates if a type 1 packet is delayed or dropped, but not as much if a type 2 packet is delayed or dropped, and even less when a type 3 packet is delayed or dropped. The extra delay of a type 1 packet due to the transmissions of type 2 packets could be required to be less than a certain quantity $\alpha_{1,2}$ and that due to transmissions of type 3 packets could be required to be less than another quantity $\alpha_{1,3}$. Similarly, the extra delay of a type 2 packet due to transmissions of type 3 packets may be bounded by $\alpha_{2,3}$. One may infer *ED* values corresponding to each of these three relationships through maintaining three different *EDD* queues, with each queue managed similarly as in the case of the $CL(\alpha)$ scheduler presented in this paper. Trade-offs between scheduler complexity, desired quality and bandwidth capacity may be achieved by adjusting the α values and the number of relationships for which an α value is defined.

Finally, in order to satisfy the diverse QoS requirements of real-time communication applications, we investigate their operational characteristics at end-systems. The common quality of service desired by such applications, besides fairness, is an upper bound on the loss rate with delay constraints. We apply the basic idea of fairness in the context of scheduling real-time traffic with QoS assurances and derive the specific fairness requirements when scheduling such traffic. Using the notion of (m, k) delay criterion, we define that the service goal of scheduling soft real-time traffic is to achieve fairness in the dynamic failure rate for every flow awaiting service in the system.

Since several strategies have been proposed to schedule soft real-time traffic, we analyzed these strategies and showed that none of them achieve the goal of both QoS assurances and fairness. In this dissertation, we proposed two strategies which are designed to achieve the service goal we proposed. SRTS-PQP, one of the proposed strategies, has a simple data structure but its performance is not as good as the other strategy, SRTS-CPQ. SRTS-CPQ achieves better performance with a new data structure for maintaining packet information. Although the system implements complex operations to update required information, the processing complexity is limited to the same order, $O(\log N)$, of SRTS-CPQ.

Using real Voice-over-IP (VoIP) traffic traces, we have evaluated both strategies in simulation experiments. Our results show that they both achieve better performance than existing schedulers in both QoS assurance and fairness.

5.2 Future Work

This dissertation has primarily focussed on packet scheduling for the Internet core and therefore concentrated on requirements based on bandwidth and delay (packet losses are assumed to be packets delayed beyond a certain deadline). In wireless environments, however, new kinds of resources such as power become important and packet losses occur not just because of excessive delays but also simply due to errors caused by channel conditions. Some of the techniques developed in this dissertation, especially those in the GrFQ scheduler, may be used to distribute power and packet loss rates in a fair manner as well. However, wireless networks remain a challenge for scheduling techniques for controlled load or real-time services. For example, in developing a mechanism for controlled load service in a wireless network, not just the delay but also the packet losses of unmarked packets caused by marked packets need to be considered (since packet losses are likely in wireless networks and admission control cannot always guarantee that admitted traffic will achieve a certain throughput). Similar issues arise in the design of real-time schedulers where the packet losses are caused both by inordinate delays in the buffers as well as random errors. Incorporating these random errors into the packet scheduling algorithms is a potential area of research that is not yet investigated.

This dissertation has focused only on packet scheduling strategies (which, one might argue, affect quality of service more directly than almost any other mechanism). However, other supporting mechanisms play a significant and sometimes critical role in developing the overall mechanisms that a network service may require. Future work should include research on admission control, routing, congestion control and related mechanisms to support

the scheduling algorithms developed as part of this dissertation.

It is important, however, not to create a zoo of service models and mechanisms that become so specialized as to discourage deployment. In this dissertation, we have identified certain specific new service requirements; however, developing a parsimonious set of services that will potentially serve a large class of applications (including possibly even future applications) is an ambitious but achievable research goal. Many attempts have been made in this direction, especially in the context of defining per-hop-behaviors for Differentiated Services. However, a consensus on a small set of service models has not yet been achieved. Solutions based on employing economic incentives, however, suggest a potential mechanism. In such a solution, services are merely defined in terms of the amount and type of resources that a flow requires from the network. Further, prices are associated with the consumption of each type of resource while users only modulate the price they are willing pay. This research requires algorithmic advances in the dynamic and distributed determination of prices for each of the resources based on current demand. Innovative engineering mechanisms and network protocols are needed to achieve rapid communication and distribution of these costs for feedback and billing purposes. Of course, finally, a solution based in economics will also require novel technical solutions (for packet scheduling, routing, etc.) that operate within this pricing framework.

Bibliography

- [1] J. Wroclawski, “Specification of the controlled-load network element service,” IETF Request for Comments 2211, Sep. 1997.
- [2] S. Shenker, C. Partridge, and R. Gu erin, “Specification of guaranteed quality of service,” IETF Request for Comments 2212, Sep. 1997.
- [3] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, “An architecture for Differentiated Services,” IETF Request for Comments 2475, Dec. 1998.
- [4] J. Wroclawski, “The use of RSVP with IETF Integrated Services,” Sep. 1997.
- [5] J. Heinanen, F. Baker, W. Weiss, and J. Wroclawski, “Assured Forwarding PHB Group,” IETF Request for Comments 2597, Jun. 1999.
- [6] V. Jacobson, K. Nichols, and K. Poduri, “An Expedited Forwarding PHB,” IETF Request for Comments 2598, Jun. 1999.
- [7] K. Kilkki, *Differentiated Services for the Internet*, Macmillan Technical Publishing, Indianapolis, USA, 1999.
- [8] C. Dovrolis, D. Stiliadis, and P. Ramanathan, “Proportional differentiated services: Delay differentiation and packet scheduling,” in *Proc. ACM SIGCOMM*, Sep. 1999, pp. 109–120.
- [9] I. Stoica and H. Zhang, “Providing guaranteed services without per flow management,” in *Proc. ACM SIGCOMM*, Sep. 1999, pp. 81–94.
- [10] P. Hurley and J. Y. Le Boudec, “A proposal for an asymmetric best-effort service,” in *Proc. IEEE International Workshop on Quality of Service*, Jun. 1999, pp. 132–134.
- [11] S. McCanne, V. Jacobson, and M. Vetterli, “Receiver-driven layered multicast,” in *Proc. ACM SIGCOMM*, Stanford, CA, Aug. 1996, pp. 117–130.
- [12] U. Horn and B. Girod, “Scalable video transmission for the Internet,” *Computer Networks and ISDN Systems*, vol. 29, no. 15, pp. 1833–1842, Nov. 1997.
- [13] D. Wu, Y. T. Hou, W. Zhu, Y.-Q. Zhang, and J. M. Peha, “Streaming video over the Internet: Approaches and directions,” *IEEE Trans. Circuits Syst. Video Technol.*, vol. 11, no. 3, pp. 282–300, Mar. 2001.
- [14] V. Jacobson, “Congestion avoidance and control,” in *Proc. ACM SIGCOMM*, Aug. 1988, pp. 314–329.

- [15] L. L. Peterson and B. S. Davie, *Computer Networks*, Morgan Kaufmann Publishers, Inc., 2000.
- [16] A. Demers, S. Keshav, and S. Shenker, “Analysis and simulation of a fair queueing algorithm,” in *Proc. ACM SIGCOMM*, 1989, pp. 1–12.
- [17] D. Bertsekas and R. Gallager, *Data Networks*, Prentice Hall, New Jersey, 2nd edition, 1992.
- [18] S. Keshav, *An engineering approach to computer networking: ATM networks, the Internet, and the telephone network*, Addison-Wesley, Massachusetts, 1997.
- [19] Z. Cao and E.W. Zegura, “Utility max-min: an application-oriented bandwidth allocation scheme,” in *Proc. IEEE INFOCOM*, 1999, vol. 2, pp. 793–801.
- [20] 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 Trans. Networking*, vol. 1, no. 1, pp. 344–357, June 1993.
- [21] S. R. Golestani, “A self-clocked fair queueing scheme for broadband applicaiotns,” in *Proc. IEEE INFOCOM*, Jun. 1994, pp. 636–646.
- [22] P. Goyal, H. M. Vin, and H. Cheng, “Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks,” *IEEE Trans. Networking*, vol. 5, no. 5, pp. 690–704, Oct. 1997.
- [23] J. C. R. Bennett and H. Zhang, “WF²Q: worst-case fair weighted fair queueing,” in *Proc. IEEE INFOCOM*, Mar. 1996, pp. 120–128.
- [24] M. Shreedhar and G. Varghese, “Efficient fair queueing using deficit round-robin,” *IEEE/ACM Tran. Networking*, vol. 4, no. 3, pp. 375–385, June 1996.
- [25] S. Floyd and V. Jacobson, “Link-sharing and resource management models for packet networks,” *IEEE/ACM Trans. Networking*, vol. 3, no. 4, pp. 365–386, Aug. 1995.
- [26] S. Floyd, “Notes on class-based-queueing and guaranteed service,” Unpublished notes: <http://www.aciri.org/floyd/cbq.html>, Jul. 1995.
- [27] S. S. Kanhere and H. Sethu, “Fair, efficient and low-latency packet scheduling using nested deficit round robin,” in *Proc. IEEE Workshop on High Performance Switching and Routing*, May 2001, pp. 6–10.
- [28] S. Tsao and Y. Lin, “Pre-order deficit round robin: a new scheduling algorithm for packet-switched networks,” *Computer Networks*, vol. 35, no. 2–3, pp. 287–305, Feb. 2001.
- [29] S. S. Kanhere, H. Sethu, and A. B. Parekh, “Fair and efficient packet scheduling using elastic round robin,” *IEEE Trans. Parallel and Distributed Systems*, vol. 13, no. 3, pp. 324–336, Mar. 2002.

- [30] S. S. Kanhere, *Design and analysis of fair, efficient and low-latency schedulers for high-speed packet-switched networks*, Ph.D. thesis, Drexel University, Philadelphia, PA 19104, USA, 2003.
- [31] D. Ferrari and D. C. Verma, "A scheme for real-time channel establishment in wide-area networks," *IEEE J. Selected Areas in Commun.*, vol. 8, no. 3, pp. 368–379, Apr. 1990.
- [32] D. Verma, H. Zhang, and D. Ferrari, "Guaranteeing delay jitter bounds in packet switching networks," in *Proc. Tricomm '91*. IEEE, Apr. 1991, pp. 35–46.
- [33] S. Golestani, "Congestion-free transmission of real-time traffic in packet networks," in *Proc. IEEE INFOCOM*, Jun. 1990, pp. 527–542.
- [34] L. Kleinrock, *Queueing Systems*, vol. II, John Wiley and Sons, 1976.
- [35] T. Nandagopal, N. Venkitaraman, R. Sivakumar, and V. Bharghavan, "Delay differentiation and adaptation in core stateless networks," in *Proc. IEEE INFOCOM*, Mar. 2000, pp. 421–430.
- [36] C. Dovrolis, D. Stiliadis, and P. Ramanathan, "Proportional differentiated services: Delay differentiation and packet scheduling," *IEEE/ACM Trans. Networking*, vol. 10, no. 1, pp. 12–26, Feb. 2002.
- [37] S. Bodamer, "A new scheduling mechanism to provide relative differentiation for real-time IP traffic," in *Proc. IEEE Globecom*, Nov. 2000, pp. 646–650.
- [38] D. Clark, "Explicit allocation of best-effort packet delivery service," *IEEE/ACM Trans. Networking*, vol. 6, no. 4, pp. 362–373, Aug. 1998.
- [39] W.-C. Cheng, D. D. Kandlur, D. Saha, and K. G. Shin, "Adaptive packet marking for maintaining end-to-end throughput in a Differentiated Services Internet," *IEEE/ACM Trans. Networking*, vol. 7, no. 5, pp. 685–697, Oct. 1999.
- [40] D. Stiliadis and A. Varma, "Efficient fair queueing algorithms for packet-switched networks," *IEEE/ACM Trans. Networking*, vol. 6, no. 2, pp. 175–185, Apr. 1998.
- [41] J. A. Cobb, M. G. Gouda, and A. El-Nahas, "Time-shift scheduling – fair scheduling of flows in high-speed networks," *IEEE Trans. Networking*, vol. 6, no. 3, pp. 274–285, June 1998.
- [42] Y. Zhou and H. Sethu, "On the relationship between absolute and relative fairness bounds," *IEEE Comm. Letters*, vol. 6, no. 1, pp. 37–39, Jan. 2002.
- [43] V. Jacobson, "Multimedia conferencing on the Internet," Tech. Rep., University College London, London, 1994.

- [44] D. Stiliadis and A. Varma, "Latency-rate servers: a general model for analysis of traffic scheduling algorithms," *IEEE Trans. Networking*, vol. 6, no. 5, pp. 611–624, Oct. 1998.
- [45] J. C. R. Bennett and H. Zhang, "Hierarchical packet fair queueing algorithms," *IEEE/ACM Trans. Networking*, vol. 5, no. 5, pp. 675–689, Oct. 1997.
- [46] H. Shi and H. Sethu, "An evaluation of timestamp-based packet schedulers using a novel measure of instantaneous fairness," in *Proc. of the IEEE International Performance, Computing, and Communications Conference (IPCCC)*, Apr. 2003, Available at <http://www.ece.drexel.edu/CCL/pubs.html>.
- [47] F. A. Cowell, *Measuring inequality: Techniques for the social sciences*, John Wiley & Sons, New York, 1977.
- [48] A. W. Marshall and I. Olkin, *Inequalities: Theory of majorization and its applications*, Academic Press, New York, 1979.
- [49] A. Kumar and J. Kleinberg, "Fairness measures for resource allocation," in *41st Annual Symposium on Foundation of Computer Science*. Nov. 2000, pp. 75–85, IEEE.
- [50] J. E. Stiglitz, *Economics*, W. W. Norton and Co., 1993.
- [51] Telecommunication Networks Group, "MPEG-4 and H.263 video traces for network performance evaluation," <http://www-tkn.ee.tu-berlin.de/research/trace/trace.html>.
- [52] National Laboratory for Applied Network Research, "Passive measurement and analysis," <http://pma.nlanr.net/PMA/>.
- [53] S. Jamin, S. J. Shenker, and P. B. Danzig, "Comparison of measurement-based admission control algorithms for controlled-load service," in *Proc. IEEE INFOCOM*, 1997, vol. 3, pp. 973–980.
- [54] R. Guérin and V. Peris, "Quality-of-service in packet networks: Basic mechanisms and directions," *Computer Networks*, vol. 31, no. 3, pp. 169–179, February 1999.
- [55] K. Siriwong and R. Ammar, "QoS using delay-synchronized dynamic priority scheduling," in *Proc. IEEE Symposium on Computers and Communications*, July 2001, pp. 276–281.
- [56] H. Zhang, "Service disciplines for guaranteed performance service in packet-switching networks," *Proc. IEEE*, vol. 83, no. 10, pp. 1374–1396, Oct. 1995.
- [57] D. Stiliadis and A. Varma, "Rate-proportional servers: A design methodology for fair queueing algorithms," *IEEE/ACM Trans. Networking*, vol. 6, no. 2, pp. 164–174, Apr. 1998.

- [58] C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *J. ACM*, vol. 20, pp. 46–61, Jan. 1973.
- [59] V. Sivaraman and F. Chiussi, "End-to-end statistical delay guarantees using earliest deadline first (EDF) packet scheduling," in *Proc. Globecom*, Dec. 1999, vol. 2, pp. 1307–1312.
- [60] M. Hamdaoui and P. Ramanathan, "A dynamic priority assignment technique for streams with (m, k) -firm deadlines," *IEEE Trans. Computers*, vol. 44, no. 12, pp. 1443–1451, Dec. 1995.
- [61] G. Bernat and A. Burns, "Combining (n, m) hard deadlines and dual priority scheduling," in *Proc. IEEE Real-Time Systems Symposium (RTSS)*. IEEE, Dec. 1997, pp. 46–57.
- [62] G. Quan and Y. Hu, "Enhanced fixed-priority scheduling with (m, k) -firm guarantee," in *Proc. IEEE Real-Time Systems Symposium (RTSS)*. IEEE, Nov. 2000, pp. 79–88.
- [63] P. Ramanathan, "Overload management in real-time control applications using (m, k) -firm guarantee," *IEEE Trans. Parallel and Distributed Systems*, vol. 10, no. 6, pp. 549–559, June 1999.
- [64] R. West, K. Schwan, and C. Poellabauer, "Scalable scheduling support for loss and delay constrained media streams," in *Proc. IEEE Real-Time Technology and Applications Symposium*. IEEE, Jun. 1999, pp. 24–33.
- [65] C. Chuah, "Workload model for multimedia applications," <http://www.ece.ucdavis.edu/~chuah/research/multimedia/index.html>.
- [66] K. Xu, "How has the literature on the gini index evolved in the past 80 years," to appear in *China Economic Quarterly*, 2003, Available at <http://is.dal.ca/~kxu/xkresearch.htm>.

Appendix A. Gini Index as a Measure of Fairness

In this appendix, we present several computational methods of the Gini index and discuss the properties of the Gini index in comparison with some other measures of inequality. In order to extend the Gini index as a measure of fairness for packet schedulers, we make some modifications based on the specific characteristics of fair packet schedulers.

A.1 Computational Methods of the Gini Index

The Gini index was proposed more than 80 years ago. It has been thoroughly studied by economists and widely applied in economic policy research. There are several different methods to compute the Gini index. A survey on various interpretations of the Gini index is provided in [66]. Here we present three kinds of computational methods: geometric approaches, Gini's mean difference approach and the covariance approach. We also discuss the relationships between them.

The geometric approaches of computing the Gini index are based on the Lorenz curve. Consider k quantities, $g_1 \leq g_2 \leq \dots \leq g_k$. Define $d_0 = 0$, and $d_i = d_{i-1} + g_i$, for $1 \leq i \leq k$. A plot of d_i against i is a concave curve, known as the *Lorenz curve* [50]. Figure A.1 shows two examples of the Lorenz curve. Note that if there is perfect equality in these k quantities, the Lorenz curve will be a straight line starting from the origin. The Gini index measures the area between the concave curve and the straight line. In Figure A.1, the concave curve from point 0 to A split the triangle $\Delta 0Ak$ into two parts. Denote the area between the straight line and the concave curve as α and the area under the concave curve as β . The Gini index is computed as

$$\text{Gini} = \frac{\alpha}{\alpha + \beta} = 1 - \frac{\beta}{\alpha + \beta} \quad (\text{A.1})$$

To represent this method in a mathematical format, define $F_i = i$, $L_0 = 0$ and $L_i =$

$\sum_{j=1}^i g_j, 1 \leq i \leq k$. β can be computed as

$$\beta = \frac{1}{2} \sum_{i=0}^{k-1} (F_{i+1} - F_i)(L_{i+1} + L_i)$$

Therefore,

$$\text{Gini} = 1 - \frac{1}{k^2 \bar{g}} \sum_{i=0}^{k-1} (F_{i+1} - F_i)(L_{i+1} + L_i) \quad (\text{A.2})$$

where \bar{g} is the mean value of $g_i, 1 \leq i \leq k$, i.e. $\bar{g} = \frac{1}{k} \sum_{i=1}^k g_i$. Since $F_0 = L_0 = 0, F_k = k$ and $L_k = k\bar{g}$, (A.2) can be further simplified as

$$\begin{aligned} \text{Gini} &= 1 + \frac{1}{k^2 \bar{g}} \left[\sum_{i=0}^{k-1} (F_i L_{i+1} - F_{i+1} L_i) - \sum_{i=0}^{k-1} (F_{i+1} L_{i+1} - F_i L_i) \right] \\ &= \frac{1}{k^2 \bar{g}} \sum_{i=0}^{k-1} (F_i L_{i+1} - F_{i+1} L_i) \end{aligned} \quad (\text{A.3})$$

A different way to compute the area β is

$$\beta = \sum_{i=1}^k [(k+1-i)g_i - \frac{1}{2}g_i] = \sum_{i=1}^k (k+1-i)g_i - \frac{1}{2}k\bar{g}$$

The Gini index can also be computed as

$$\begin{aligned} \text{Gini} &= 1 - \frac{2}{k^2 \bar{g}} \left[\sum_{i=1}^k (k+1-i)g_i - \frac{1}{2}k\bar{g} \right] \\ &= \frac{k+1}{k} - \frac{2}{k^2 \bar{g}} \sum_{i=1}^k (k+1-i)g_i \end{aligned} \quad (\text{A.4})$$

Therefore, the weight of each quantity is inversely associated with the value of the quantity.

In fact, (A.2) and (A.4) are equivalent to each other since

$$\begin{aligned} &\frac{1}{k^2 \bar{g}} \sum_{i=0}^{k-1} (F_i L_{i+1} - F_{i+1} L_i) \\ &= \frac{1}{k^2 \bar{g}} \sum_{i=1}^k (F_{i-1} L_i - F_i L_{i-1}) = \frac{1}{k^2 \bar{g}} \sum_{i=1}^k [F_i (L_i - L_{i-1}) - (F_i - F_{i-1})] \\ &= \frac{1}{k^2 \bar{g}} \sum_{i=1}^k (i g_i - \sum_{j=1}^i g_j) = \frac{1}{k^2 \bar{g}} \left(\sum_{i=1}^k i g_i - \sum_{i=1}^k \sum_{j=1}^i g_j \right) \\ &= \frac{1}{k^2 \bar{g}} \left[\sum_{i=1}^k i g_i - \sum_{i=1}^k (k+1-i) g_i \right] \end{aligned} \quad (\text{A.5})$$

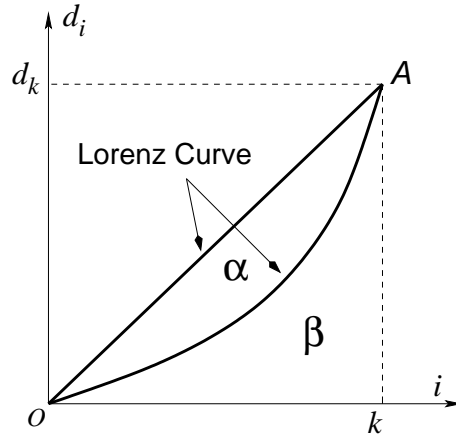


Figure A.1: An illustration of the Lorenz curve for computing the Gini index

$$\begin{aligned}
 &= \frac{1}{k^2 \bar{g}} \left[\sum_{i=1}^k (k+1)g_i - 2 \sum_{i=1}^k (k+1-i)g_i \right] \\
 &= \frac{k+1}{k} - \frac{2}{k^2 \bar{g}} \sum_{i=1}^k (k+1-i)g_i
 \end{aligned}$$

The Gini index can also be interpreted as the relative mean difference of a set of quantities. This relation was first given by Gini in 1912, which is the reason that the index is so named [66]. If we define the mean difference of g_1, g_2, \dots, g_k as

$$\Delta = \frac{1}{k^2} \sum_{i=1}^k \sum_{j=1}^k |g_i - g_j|$$

the Gini index is one-half of the relative mean difference which is the mean difference divided by the mean \bar{g} , i.e.

$$\text{Gini} = \frac{\Delta}{2\bar{g}} = \frac{1}{2k^2 \bar{g}} \sum_{i=1}^k \sum_{j=1}^k |g_i - g_j| \quad (\text{A.6})$$

Since

$$\sum_{i=1}^k \sum_{j=1}^k |g_i - g_j| = 2 \sum_{i=1}^k \sum_{j=1}^k \max(0, g_i - g_j) = 2 \sum_{i=1}^k \sum_{j \leq i} (g_i - g_j)$$

(A.6) can also be expressed as

$$\text{Gini} = \frac{1}{k^2 \bar{g}} \sum_{i=1}^k \sum_{j \leq i} (g_i - g_j) = \frac{1}{k^2 \bar{g}} \sum_{i=1}^k \left(i g_i - \sum_{j=1}^i g_j \right)$$

which has the same form as in (A.5). Therefore the expression (A.6) based on the relative mean difference is equivalent to expressions (A.2) and (A.4).

If we define the covariance between quantity value and its rank as

$$cov(g_i, i) = \frac{1}{k} \sum_{i=1}^k (g_i - \bar{g})(i - \bar{i})$$

where $\bar{i} = \frac{1}{k} \sum i = (k + 1)/2$, the Gini index can be computed by

$$\begin{aligned} \text{Gini} &= \frac{2cov(g_i, i)}{k\bar{g}} = \frac{2}{k^2\bar{g}} \sum_{i=1}^k (g_i - \bar{g})(i - \bar{i}) \\ &= \frac{2}{k^2\bar{g}} \sum_{i=1}^k ig_i - \frac{k+1}{k} \end{aligned} \quad (\text{A.7})$$

Since (A.4) can also be transformed as

$$\begin{aligned} &\frac{k+1}{k} - \frac{2}{k^2\bar{g}} \sum_{i=1}^k (k+1-i)g_i \\ &= \frac{k+1}{k} - \frac{2(k+1)}{k} + \frac{2}{k^2\bar{g}} \sum_{i=1}^k ig_i \\ &= \frac{2}{k^2\bar{g}} \sum_{i=1}^k ig_i - \frac{k+1}{k} \end{aligned}$$

(A.7) is equivalent to (A.4). Therefore, all these methods of computing the Gini index are equivalent to each other.

A.2 Comparison to Other Measures

As shown in the expression of the Gini index based on the relative mean difference, the Gini index incorporates the difference between any two quantities in the group. A similar measure might be the sum of the distances from the mean, divided by the mean. However, this metric gives more emphasis on extremely large or small quantities; it is hard to tell whether the inequality within an income distribution is due to the difference among most quantities or due to the difference between quantities with extremely large or small values. This ambiguous situation can be illustrated using examples in Figure A.2. Figures A.2 (a) and (b) plot two groups of k quantities. In both graphs, curve $0CD$ represents the value

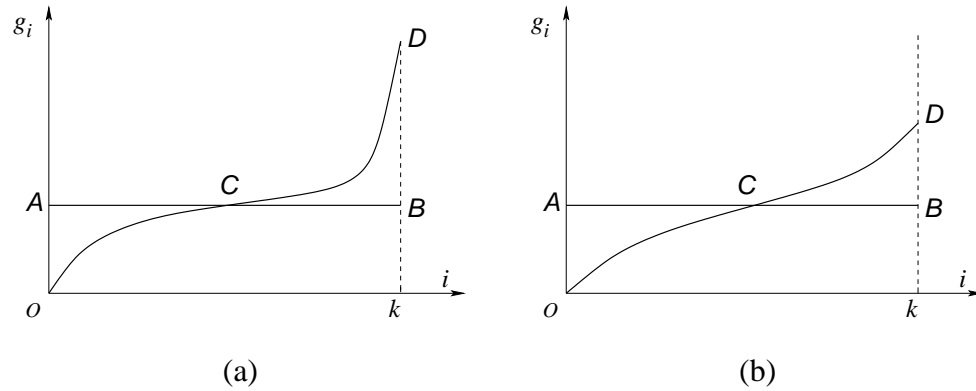


Figure A.2: An illustration of two distributions

of k quantities and line AB represents the mean value of the same k quantities. The sum of the distance from the mean is nothing but the area between line AB and curve $0CD$. We can observe that both groups have the same mean value and the area between line AB and curve $0CD$ is almost same in both graph. However in Figure A.2(a) most quantities are close to each other and close to the mean value while most quantities are different from each other in Figure A.2(b). Therefore, the sum of the distances from the mean cannot show how much difference there exists between majority of the quantities.

Another common metric used by statisticians is variance or standard deviation. It has the same shortcoming as the sum of distances from the mean. Therefore, variance cannot serve as a measure of inequality as well as the Gini index does.

A.3 Gini Index for Packet Schedulers

Since the Gini index captures the inequality among a group of unequal quantities, we use it to measure the inequality among session utilities and therefore measure the instantaneous fairness in bandwidth allocation. However the characteristics of session utilities require some modification of the original definition of the Gini index. Here we explain the rationale behind these modifications.

The first modification is about choosing the set of equal quantities. When we compute the Gini index for social income distribution, the straight line associated with the ideal equal distribution can be obtained by dividing the total income by the total number of persons. However, in a packet scheduler, we cannot use the same method since not all flows are continuously backlogged from the beginning of the system. The GPS virtual time records the normalized service a flow should receive in a GPS system, an ideal system which allocates service equally among all backlogged flows. Therefore, we use the GPS virtual time as the mean of session utilities and the straight line is associated with the GPS virtual time, which is also the normalized service in the GPS reference system.

Similar to the Gini index in the economic field, we define the fairness metric as the area between the Lorenz curves of the real scheduling system and the GPS reference system. Since the sum of normalized service in a real system may not equal that in a GPS system even though the sum of total service is equal in the two systems, the ending point of the Lorenz curve of a real system may not always meet the ending point of the curve from the GPS system. Therefore, under certain conditions, the Lorenz curve of a packet scheduling system may appear similar to that in Figure 2.4(b). This results in another modification of the Gini index. As recalled from Equation (A.1) and Figure A.1, the Gini index in the economic field divides the area α by the whole area under the Lorenz curve of an equal distribution, $\alpha + \beta$. For packet schedulers, we do not use such relative value since the total area under the Lorenz curve of the GPS system can increase as time elapses, while the area between Lorenz curves from the GPS and the real system will not increase significantly if the real system is close to being fair. In order to avoid a misleading result because of the relative area reducing as the system keeps executing, we only use the absolute area as the Gini index for packet schedulers.

Appendix B. Data Structure for the SRTS Scheduler

B.1 Data Structure for the Timetable

In this appendix, we present the data structure for the timetable. Fig. 4.6 plots an example timetable with six occupied periods and five vacant periods. The elements of a timetable include *VacancyTree*, *pktLenAtTail*, *earliestSD* and *latestD*. Except for *VacancyTree*, all other elements are registers. *VacancyTree* is basically an augmented binary search tree consisting of *VE* nodes. Each node contains a vacant period and the occupied period just before it. The lengths of both periods are stored inside the *VE* node, and the tuple (b, e) which defines the beginning and ending of the vacant period is used as the key for the binary tree. In addition, each *VE* node stores the total length of vacant periods covered by its left and right subtrees in registers *leftLen* and *rightLen*. Besides the tree organization, *VacancyTree* also combines a linked list structure of nodes. The order of this linked list conforms to the sorted order of *VacancyTree* and we denote the linked list as *VacancyList*. The pointers of each *VE* are listed in Table B.1.

To create a new *VE*, the beginning time, the ending time and the length of the occupied period before this *VE* are initialized based on the given information. The pointers for *VacancyTree* and *VacancyList* are initialized as null pointers and will be modified when this *VE* is inserted into the tree and the list. The pseudo-code is presented in Figure B.1.

The rest of this section describes the two basic operations: inserting a packet and releasing a packet. The next appendix section describes the details of the data structure of *VacancyTree*.

```

NewVE( $t_{\text{begin}}$ ,  $t_{\text{end}}$ ,  $L_{\text{occupied}}$ )
   $b \leftarrow t_{\text{begin}}$ ;
   $e \leftarrow t_{\text{end}}$ ;
   $\text{pktLen} \leftarrow L_{\text{occupied}}$ ;
   $\text{len} \leftarrow e - b$ ;
  All pointers are initialized as NULL;

```

Figure B.1: Create a new *VE*Table B.1: Pointers in a *VacancyElement*

<i>VacancyTree</i>	<i>parent; left; right;</i>
<i>VacancyList</i>	<i>front; back;</i>

B.1.1 Inserting a Packet

The insertion operations can be categorized into several cases and each case is independent of another. Here we summarize the cases in Table B.2 and provide the pseudo-code for each of the cases separately. Within the code, we assume that each packet has a deadline (D) and a starting deadline (SD).

B.1.2 Releasing a Packet

When a packet in the timetable is transmitted, it should be released from the timetable. Since the scheduling order is based on the order of deadlines, the released packet must from the earliest occupied period. Figure B.8 shows the pseudo-code of this operation.

Table B.2: Different cases when inserting a packet into a timetable.

	Description	Logic Condition
Case 1:	The timetable is empty.	$pktLenAtTail = 0$
Case 2:	The timetable is nonempty; <i>VacancyTree</i> is empty.	$pktlenAtTail \neq 0$ AND $VacancyTree.root = NULL$
Case 3:	<i>VacancyTree</i> is not empty; The inserted packet is outside the range of the timetable.	$VacancyTree.root \neq NULL$ AND $(P.SD \geq latestD + 1$ OR $P.D \leq earliestSD - 1)$
Case 4:	<i>VacancyTree</i> is not empty; The inserted packet is within the range of the timetable.	$VacancyTree \neq NULL$ AND $P.SD \leq latestD$ AND $P.D \geq earliestSD$

B.1.3 Searching

Searching the position of a packet to be inserted is locating the occupied period where the packet will reside. The operation returns the latest one among *VEs* which are affected by the packet if there exists one. If the packet is within the first occupied period, the operation returns the head of *VacancyList*. The pseudo-code is describe in Figure B.9.

B.2 Data Structure for *VacancyTree*

In this section, we present insertion and deletion operations for *VacancyTree*.

B.2.1 Insertion

Inserting a *VE* into the tree consists of inserting the *VE* into the tree and *VacancyList* while also updating the length of vacant periods at the same time. The pseudo-code is presented in Figure B.10.

B.2.2 Deletion

Deleting a *VE* from the tree procedure only operates on the pointers for *VacancyTree*. The structure of *VacancyList* remains untouched. The records of vacancy lengths are not ad-

justed except for the case when the deleted *VE* is replaced by its successor. The pseudo-code is presented in Figure B.11.

```

InsertPacket(P):
Case 1:
    earliestSD  $\leftarrow$  P.SD;
    latestSD  $\leftarrow$  P.D;
    pktLenAtTail  $\leftarrow$  P.length;

Case 2:
    if (P.SD > latestD + 1) then
        W  $\leftarrow$  NewVE(latestD + 1, P.SD - 1, pktLenAtTail);
        pktLenAtTail  $\leftarrow$  P.length;
        latestD  $\leftarrow$  P.D;
        InsertVEtoVacancyTree(W);
    else if (P.D < earliestSD - 1) then
        W  $\leftarrow$  NewVE(P.D + 1, earliestSD - 1, P.length);
        earliestSD  $\leftarrow$  P.SD;
        InsertVEtoVacancyTree(W);
    else if (P.SD  $\leq$  latestD + 1 AND P.D  $\geq$  earliestSD - 1) then
        pktLenAtTail  $\leftarrow$  pktLenAtTail + P.length;
        latestD  $\leftarrow$  max{latestD, P.D};
        earliestSD  $\leftarrow$  latestD - pktLenAtTail;
    end if;

Case 3:
    if (P.SD > latestD + 1) then
        W  $\leftarrow$  NewVE(latestD + 1, P.SD - 1, pktLenAtTail);
        pktLenAtTail  $\leftarrow$  P.length;
        latestD  $\leftarrow$  P.D;
        InsertVEtoVacancyTree(W);
    else if (P.SD = latestD + 1) then
        pktLenAtTail  $\leftarrow$  pktLenAtTail + P.length;
        latestD  $\leftarrow$  P.D;
    else if (P.D = earliestSD - 1) then
        H  $\leftarrow$  HeadOfVacancyList;
        H.pktlen  $\leftarrow$  H.pktlen + P.length;
        earliestSD  $\leftarrow$  P.SD;
    else if (P.D < earliestSD - 1) then
        W  $\leftarrow$  NewVE(P.D + 1, earliestSD - 1, P.length);
        earliestSD  $\leftarrow$  P.SD;
        InsertVEtoVacancyTree(W);
    end if;

Case 4:
    V  $\leftarrow$  SearchPositionInVacancyTree(P);
     $\Delta T$   $\leftarrow$  ComputeAdditionalVacancyNeeded(V, P);
    if ( $\Delta T$  > 0) then
        U  $\leftarrow$  V;
        downward  $\leftarrow$  UpwardSearch( $\Delta T$ , U, V, totalShrink);
        if (downward = TRUE) then
            DownwardSearch( $\Delta T$ , U, V);
        end if;
    end if;

```

Figure B.2: Pseudocode of *InsertPacket* routine

```

UpwardSearch( $\Delta T$ ,  $U$ ,  $V$ ,  $totalShrink$ )
  downward  $\leftarrow$  FALSE;
  while (TRUE) do
    OccupyVacancyLen( $\Delta T$ ,  $U$ ,  $V$ ,  $totalShrink$ , TRUE);
    while ( $U.parent \neq$  NULL AND  $U$  is a left child) then
       $U \leftarrow U.parent$ ;
       $U.leftLen \leftarrow U.leftLen - totalShrink$ ;
    end while;
    if ( $U$  is the root) then
      if ( $V$  is in the left subtree of  $U$ ) then
         $earliestSD \leftarrow earliestSD - \Delta T$ ;
      else if ( $U.len > \Delta T$ ) then
         $U.e \leftarrow U.e - \Delta T$ ;  $W \leftarrow U.back$ ;
        Delete VEs from  $W$  to  $V$  in  $VacancyList$ ;
      else if ( $U.len = \Delta T$ ) then
        DeleteVEfromVacancyTree $U$ ;
        Delete VEs from  $U$  to  $V$  in  $VacancyList$ ;
      else
         $\Delta T \leftarrow \Delta T - U.len$ ;
        if ( $U.leftLen \geq \Delta T$ ) then
           $U \leftarrow U.left$ ;  $downward \leftarrow$  TRUE;
          DeleteVEfromVacancyTree( $root$ );
        else /*  $U.leftLen < \Delta T$  */
           $\Delta T \leftarrow \Delta T - U.leftLen$ ;
           $earliestSD \leftarrow earliestSD - \Delta T$ ;
          Cut  $U$  and  $U$ 's left subtree from  $VacancyTree$ ;
          Delete VEs before  $V$  in  $VacancyList$ ;
        end if;
      end if;
      break;
    else
      OccupyVacancyLen( $\Delta T$ ,  $U$ ,  $V$ ,  $totalShrink$ , FALSE);
    end if;
  end while;
  return downward;

```

Figure B.3: Pseudocode of *UpwardSearch* routine

```

OccupyVacancyLen( $\Delta T$ ,  $U$ ,  $V$ , totalShrink, leftTree)
  if (leftTree = TRUE) then
     $vlen \leftarrow U.leftLen$ ;
  else  $vlen \leftarrow U.len$ ;
  end if;
  if ( $vLen \geq \Delta T$ ) then
     $totalShrink \leftarrow totalShrink + \Delta T$ ;
    UpwardUpdateOnVacancyLength( $U$ ,  $totalShrink$ );
    if (leftTree = TRUE) then
       $W \leftarrow U$ ;
       $U \leftarrow U.left$ ;
      DeleteVEfromVacancyTree( $W$ );
       $downward \leftarrow TRUE$ ;
    else
      if ( $vlen = \Delta T$ ) then
        DeleteVEfromVacancyTree( $U$ );
      else /*  $U.len > \Delta T$  */
         $U.e \leftarrow U.e - \Delta T$ ;
         $U \leftarrow U.back$ ;
      end if;
      Delete VEs from  $U$  to  $V$  in VacancyList;
    end if;
    break;
  else
     $\Delta T \leftarrow \Delta T - vlen$ ;
     $totalShrink \leftarrow totalShrink + \Delta T$ ;
    if (leftTree = TRUE) then
      if ( $U.left \neq NULL$ ) then
        Cut  $U$ 's left subtree from VacancyTree;
      else
        DeleteVEfromVacancyTree( $U$ );
      end if;
    end if;
  end if;

```

Figure B.4: Pseudocode of *OccupyVacancyLen* routine


```

DownwardSearch( $\Delta T$ ,  $U$ ,  $V$ )
  while ( $downward = \text{TRUE}$ ) do
    if ( $\Delta T > U.\text{rightLen} + U.\text{len}$ ) then
       $\Delta T \leftarrow \Delta T - U.\text{rightLen} - U.\text{len}$ ;
      Cut  $U$  and its right subtree in VacancyTree;
       $U \leftarrow U.\text{left}$ ;
    else if ( $\Delta T > U.\text{rightLen}$ ) then
       $downward \leftarrow \text{FALSE}$ ;
       $\Delta T \leftarrow \Delta T - U.\text{rightLen}$ ;
      Cut  $U$ 's right subtree from VacancyTree;
      if ( $U.\text{len} = \Delta T$ ) then
        DeleteVEfromVacancyTree( $U$ );
        Delete VEs from  $U$  to  $V$  in VacancyList;
      else /*  $U.\text{len} > \Delta T$  */
         $U.e \leftarrow U.e - \Delta T$ ;
         $W \leftarrow U.\text{back}$ ;
        Delete VEs from  $W$  to  $V$  in VacancyList;
      end if;
      break;
    else /*  $\Delta T < U.\text{rightLen}$  */
       $U.\text{rightLen} \leftarrow U.\text{rightLen} - \Delta T$ ;
       $U \leftarrow U.\text{right}$ ;
    end if;
  end while;

```

Figure B.5: Pseudocode of *DownwardSearch* routine

```

ComputeAdditionalVacancyNeeded(V, P)
   $\Delta T \leftarrow 0$ ;
  if (V = HeadOfVacancyList AND V.b > P.D) then
    V.pktLen  $\leftarrow$  V.pktLen + P.length;
    earliestSD  $\leftarrow$  earliestSD - P.length;
     $\Delta T \leftarrow 0$ ;
  else if (V = HeadOfVacancyList
    AND V.b  $\geq$  P.SD AND V.e > d) then
    V.pktLen  $\leftarrow$  V.pktLen + P.length;
     $\Delta T \leftarrow$  P.length - (P.D - V.b);
    earliestSD  $\leftarrow$  earliestSD -  $\Delta T$ ;
     $\Delta T \leftarrow 0$ ;
    V.b  $\leftarrow$  P.D + 1;
    totalShrink  $\leftarrow$  P.D - V.b;
    UpwardUpdateOnVacancyLen(V, totalShrink);
  else if (V.b < P.SD AND V.e > P.D) then
    W  $\leftarrow$  NewVE(b  $\leftarrow$  P.D + 1, e  $\leftarrow$  V.e);
    W.pktLen  $\leftarrow$  P.length;
    InsertVEtoVacancyTree(W);
    totalShrink  $\leftarrow$  V.e - P.SD;
    V.e  $\leftarrow$  P.SD;
    UpwardUpdateOnVacancyLen(V, totalShrink);
  else if (V.b  $\geq$  P.SD AND V.e > P.D) then
    V.pktLen  $\leftarrow$  V.pktLen + P.length;
     $\Delta T \leftarrow$  P.length - (P.D - V.b);
    totalShrink  $\leftarrow$  P.D - V.b;
    V.e  $\leftarrow$  P.D;
    UpwardUpdateOnVacancyLen(V, totalShrink);
    if ( $\Delta T > 0$ ) then
      V  $\leftarrow$  V.front;
      V.e  $\leftarrow$  V.e -  $\Delta T$ ;
       $\Delta T \leftarrow$   $\Delta T$  - V.len;
    end if;
  else
    if (V = TailOfVacancyList AND P.D > latestD) then
       $\Delta T \leftarrow$   $\Delta T$  - (P.D - latestD);
      latestD  $\leftarrow$  P.D;
    else  $\Delta T \leftarrow$  P.length;
    end if;
    V.e  $\leftarrow$  V.e -  $\Delta T$ ;
     $\Delta T \leftarrow$   $\Delta T$  - V.len;
  end if;
  return  $\Delta T$ ;

```

Figure B.6: Pseudocode of *ComputeAdditionalVacancyNeeded* routine

```

UpwardUpdateOnVacancyLen(V, totalShrink):
  W ← V.parent;
  while (W ≠ NULL) do
    if (V is a left child) then
      W.leftLen ← W.leftLen − totalShrink;
    else /* V is a right child */
      W.rightLen ← W.rightLen − totalShrink;
    end if;
    V ← W;
    W ← W.parent;
  end while;

```

Figure B.7: Pseudocode of *UpwardUpadteOnVacancyLen* routine

```

ReleasePacket(P):
  earliestSD ← earliestSD + P.length;
  if (VacancyTree is not empty) then
    H ← HeadOfVacancyList;
    H.pktLen ← H.pktLen − P.length;
    if (H.pktLen ≤ 0) then
      DeleteVEfromVacancyTree(H);
      DeleteVEfromVacancyList(H);
    else /* VacancyTree is empty */
      pktLenAtTail ← pktLenAtTail − P.length;
    end if;

```

Figure B.8: Pseudocode of *ReleasePacket* routine

```

SearchPositionInVacancyTree(P):
  W ← HeadOfVacancyList;
  X ← TailOfVacancyList;
  if (P.SD > latestD OR P.D < earliestSD OR W = NULL) then
    V ← NULL;
  else if (W ≠ NULL AND earliestSD ≤ P.D < W.b) then
    V ← W;
  else if (X ≠ NULL AND P.SD ≤ earliestSD AND P.D ≥ X.b) then
    V ← X;
  else
    W ← RootOfVacancyTree;
    while (W ≠ NULL) do
      X ← W.front;
      Z ← W.back;
      if (Z ≠ NULL) then
        if (W.b ≤ P.D < Z.b) then
          V ← W;
          break;
        else if (W.b ≥ P.D) then
          W ← W.left;
        else /* Z.b ≤ P.D */
          W ← W.right;
        end if;
      else if (X ≠ NULL) then
        if (X.b ≤ P.D < W.b) then
          V ← X;
          break;
        else if (W.b ≤ P.D) then
          V ← W;
          break;
        else
          W ← W.left;
        end if;
      else
        V ← W;
        break;
      end if;
    end while;
  end if;
  return V;

```

Figure B.9: Pseudocode of *SearchPositionInVacancyTree* routine

```

InsertVEtoVacancyTree(V):
   $W \leftarrow \text{RootOfVacancyTree};$ 
  if (VacancyTree is not empty) then
    while ( $W \neq \text{NULL}$ ) do
       $X \leftarrow W;$ 
      if ( $W.e > V.e$ ) then
         $W.\text{leftLen} \leftarrow W.\text{leftLen} + V.\text{len};$ 
         $W \leftarrow W.\text{left};$ 
      else
         $W.\text{rightLen} \leftarrow W.\text{rightLen} + V.\text{len};$ 
         $W \leftarrow W.\text{right};$ 
      end if;
    end while;
     $V.\text{parent} \leftarrow X;$ 
    if ( $V.e < X.e$ ) then
       $X.\text{left} \leftarrow V;$ 
      Insert  $V$  before  $X$  in VacancyList;
    else
       $X.\text{right} \leftarrow V;$ 
      Insert  $V$  behind  $X$  in VacancyList;
    end if;
  else /* VacancyTree is empty */
     $\text{RootOfVacancyTree} \leftarrow V;$ 
     $\text{HeadOfVacancyList} \leftarrow V;$ 
     $\text{TailOfVacancyList} \leftarrow V;$ 
  end if;

```

Figure B.10: Pseudocode of the insertion routine of *VacancyTree*

```

DeleteVEfromVacancyTree(V):
  if (V only has up to one child) then
    Replace V by its child;
  else
     $W \leftarrow \text{MinOfSubtreeWithRoot}(V.\text{right});$ 
     $X \leftarrow W.\text{parent};$ 
    while ( $X \neq V$ ) do
       $X.\text{leftLen}$  gets  $X.\text{leftLen} - W.\text{len};$ 
       $X \leftarrow X.\text{parent};$ 
    end while;
    DeleteVEfromVacancyTree(W);
    Replace V by W;
     $W.\text{leftLen} \leftarrow V.\text{leftLen};$ 
     $W.\text{rightLen} \leftarrow V.\text{rightLen} - W.\text{len};$ 
  end if;

```

Figure B.11: Pseudocode of the deletion routine in *VacancyTree*

Vita

Hongyuan Shi was born in Beijing, China. She graduated from Tsinghua University in Beijing, China, with a B.S. in Electronic Engineering. She joined the Department of ECE at Drexel University as a graduate student in 1998 and subsequently became a research assistant working under the supervision of Dr. Stewart Personick and later as a member of the Computer Communications Laboratory under the supervision of Dr. Harish Sethu. During her years at Drexel, she has contributed her research efforts in the architecture and design of high-speed optical routers, optical interconnection network topologies, quality-of-service issues under emerging Internet service models, and novel traffic management strategies. During her years at Drexel, Hongyuan has also served as a teaching assistant for a variety of courses in computer engineering. She also spent the summer of 1999 as an intern at Telcordia Technologies, developing component modules for a simulation tool on the transmission performance of optical networks. She is a student member of Society of Women Engineers, IEEE and IEEE Communications Society.

