Abstract

Title of Dissertation: Supporting Distributed Multimedia Applications on ATM Networks

Debanjan Saha, Doctor of Philosophy, 1995

Dissertation directed by: Professor Satish K. Tripathi
Department of Computer Science

ATM offers a number of features, such as high-bandwidth, and provision for per-connection quality of service guarantees, making it particularly attractive to multimedia applications. Unfortunately, the bandwidth available at ATM's data-link layer is not visible to the applications due to operating system (OS) bottlenecks at the host-network interface. Similarly, the promise of per-connection service guarantees is still elusive due to the lack of appropriate traffic control mechanisms. In this dissertation, we investigate both of these problems, taking multimedia applications as examples.

The OS bottlenecks are not limited to the network interfaces, but affect the performance of the entire I/O subsystem. We propose to alleviate OS's I/O bottleneck by according more autonomy to I/O devices and by using a connection oriented framework for I/O transfers. We present experimental results on a video conferencing testbed demonstrating the tremendous performance impact of the proposed I/O architecture on networked multimedia applications.

To address the problem of quality of service support in ATM networks, we propose a simple cell scheduling mechanism, named carry-over round robin (CORR). Using analytical techniques, we analyze the delay performance of CORR scheduling. Besides providing guarantees on delay, CORR is also fair in distributing the excess bandwidth. We show that albeit its simplicity, CORR is very competitive with other more complex schemes both in terms of delay performance and fairness.

Supporting Distributed Multimedia Applications on ATM Networks

by

Debanjan Saha

Dissertation submitted to the Faculty of the Graduate School
of The University of Maryland in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
1995

Advisory Committee:

Professor Satish K. Tripathi, Chairman/Advisor
Professor Ashok K. Agrawala
Assistant Professor Michael Franklin
Assistant Professor Richard Gerber
Professor Joseph JaJa
Professor Armand Makowski

© Copyright by
Debanjan Saha
1995

Dedication

To my parents.

Acknowledgements

I would like to express my earnest gratitude to my advisor, Professor Satish Tipathi, for motivating me to do a Ph.D., arranging for my funding, and supervising this thesis with care and interest. He has been the ideal advisor, one who allowed me to explore the space while providing me with experience and insight which I often lacked. I will always be grateful to him for believing in my abilities and for being on my side at every step of the way. Many thanks to Professor Ashok Agrawala for his support and encouragement. A special thanks to Professor Richard Gerber for his interest in my work and for many exciting and enlightening discussions we had during my stay in Maryland. I thank the members of my thesis committee, Professors Armand Makowski, Joseph JaJa, and Michael Franklin for carefully reading my thesis and making suggestions for improvement.

I would like to thank Drs. Dilip Kandlur at IBM Research, Dipak Ghosal at Bellcore, T.V. Lakshman and Hemant Kanakia in AT&T Bell Labs for their help during different stages of my dissertation research. A very special thanks goes to Sarit Mukherjee, Pravin Bhagwat, and Manas Saksena for their constructive criticism of my work and for many late night debates which influenced my brain waves. I am particularly grateful to Jennifer Yuin, Ibrahim Matta, and Krishnan Kailas for reading different parts of the thesis and for suggesting numerous improvements. I would like to thank the members of the Systems Design and Analysis Group, particularly Cengiz Alaettinoglu, Sedat Akyurek, Sanjeev Setia, Dheeraj Sanghi, Bao Trinh, Chia-Mei Chen, Partho Mishra, and Frank Miller for creating a wonderful atmosphere for research. Thanks to Nancy Lindley for making my life in the department a lot less complex than it would have been.

I would like to thank all my friends, particularly Kateel Vijayananda, Shailendra Verma, Shuvanker Ghosh, Stayandra Gupta, Gagan Agrawal, Kaustabh Duorah, Greg Baratoff, Sibel Adali, Claudia Rodriguez, and Koyeli Dey, for making my life inside and outside school a lot more fun.

Although my parents are on the other side of the Atlantic, they have been a constant source of encouragement. Without their support, this thesis would not have been complete. Finally, I would like to thank a sweet muffin for her love and affection which will never fade in my memory.

Table of Contents

Section **Page**

List of Figures vii

1 Introduction 1

1.1 Problem Description and Solution Approach 2

1.1.1 Application Throughput 2

1.1.2 Quality of Service 5

1.2 Summary of Contributions 8

1.3 Organization 9

2 Autonomous Device Architecture 10

2.1 Current State of Art 10

2.2 Guiding Principles 13

2.2.1 Device Autonomy 13

2.2.2 Connection Oriented I/O 14

2.3 System Components 15

2.4 Data Flow 16

2.5 Flow Control 18

2.6 Data Processing Modules 18

2.7 Summary 20

3 Experimental Evaluation **21**

3.1 System Platform 22

3.2 Base System: Implementation and Profiling 24

3.2.1 Implementation 24

3.2.2 Data Path Profiling 25

3.3 Optimizations with Autonomous I/O 28

3.3.1 Driver Extensions 28

3.3.2 Connection Specific Protocol Processing 30

3.3.3 Application Structure 30

3.3.4 Data Flow 31

3.3.5 Limitations 33

3.4 Performance Results 34

3.5 Summary 35

4 Traffic Shaping **36**

4.1 Traffic Shapers 37

4.2 Simple Shapers 38

4.3 Composite Shapers 40

4.3.1 Composite Leaky Bucket 42

4.3.2 Composite Moving Window 45

4.3.3 Composite Jumping Window 48

4.4 Summary 49

5 Carry-Over Round Robin Scheduling **51**

5.1 Current State of Art 51

5.1.1 Delay Guarantee 52

5.1.2 Throughput Guarantee . . . 52
5.2 Scheduling Algorithm . . . 54
5.2.1 Discussion . . . 57
5.3 Implementation in a Switch . . . 57
5.4 Basic Properties . . . 59
5.5 Summary . . . 62

6 Quality of Service Envelope **63**
6.1 Delay Analysis . . . 63
6.1.1 Single-node Case . . . 64
6.1.2 Multiple-node Case . . . 70
6.1.3 Comparison with Other Schemes . . . 75
6.2 Fairness Analysis . . . 78
6.2.1 Fairness of CORR . . . 79
6.2.2 Comparison with Other Schemes . . . 80
6.3 Summary . . . 81

7 Conclusions **82**
7.1 Contributions . . . 82
7.2 Future Directions . . . 83

List of Figures

Number **Page**

1.1 A video capture application. 3

2.1 Seperation of control and data flows. 14

2.2 Alternative data paths. 17

2.3 DMA data path. 19

3.1 RS/6000 architecture. 22

3.2 MMT adapter. 23

3.3 Transmit latency in the base system (in microseconds). 26

3.4 Receive latency in the base system (in microseconds). 26

3.5 Transmit and receive data path in the optimized system. 31

3.6 Comparison of transmit latencies in the base and optimized system (in microseconds). 32

3.7 Comparison of receive latences in the base and optimize system (in microseconds). . 32

3.8 Transmit and receive throughputs. 33

4.1 The network model. 37

4.2 An example of an MPEG coded stream. 40

4.3 Composite leaky bucket. 41

4.4 Shaping with multiple leaky buckets. 42

4.5 Traffic envelope after adding $(m+1)^{th}$ bucket. 44

4.6 Traffic envelope of a composite moving window. 46

4.7 Traffic envelope after adding the $(l+1)^{th}$ moving window. 47

4.8 Traffic envelope of a composite jumping window shaper. 49

5.1 Carry-Over Round Robin Scheduling. 55

5.2 An Example Allocation. 57

5.3 Architecture of the buffer manager. 58

6.1 Computing delay and backlog from the arrival and departure functions. 64

6.2 Nodes in tandem. 70

Chapter 1

Introduction

The introduction of digital audio and video to the desktop computing environment has opened the door to an array of exciting applications, such as multimedia conferencing, video-on-demand, tele-medicine, virtual reality, just to name a few. Along with the excitement and opportunity, the emergence of these multimedia applications have brought new challenges to all aspects of system design. Audio and video are fundamentally different from the classical media, such as text and graphics, in terms of their storage, processing, distribution, and display requirements. Consequently, the computing and communication subsystems are going through an era of reevaluation and redesign to rise up to the challenges posed by the emerging multimedia applications. In this dissertation our focus is on the communication subsystem.

From the network's perspective, multimedia applications are different from the traditional text and graphics applications in two fundamental ways: 1) they generate orders of magnitude more data, and 2) they often have to satisfy stringent timing requirements on the presentation of the data. Unfortunately, the existing networks, designed mainly for low-bandwidth, non-time-critical applications, provide neither the bandwidth nor the timing guarantees demanded by these applications. This lack of adequate network support is the main driving force behind the ambitious endeavor to develop a new generation of networking infrastructure. Among the set of alternative technologies proposed as the basic switching fabric of this infrastructure, Asynchronous Transfer Mode [21], popularly known as ATM, is a clear forerunner.

Architecturally, ATM networks consist of switches connected via point-to-point fiber links. ATM switches can scale from small multiplexers to very large switches in both the aggregate capacity and the number of access ports. A switch can accommodate access ports from low speeds (1.5Mb/s and 6Mb/s) to very high speeds (2.4 Gb/s). The number of access ports can vary from four to eight in local area switches to hundreds and thousands in wide area switches. Clearly, ATM has the potential to solve the bandwidth crisis in both local and wide area networks. Operationally, an ATM network is a connection oriented cell switching network. That is, a connection is established between the source and the destination before data transfer can begin. Once the connection is established, data is exchanged in fixed size cells, all of which follow the same path from the source to the destination. The connection oriented architecture of ATM makes it possible to reserve resources for each connection, and hence it provides a framework to develop an end-to-end service architecture that can provide performance guarantees on a per connection basis.

ATM has the potential to provide both high bandwidth and per connection service guarantees, making it particularly attractive to multimedia applications. Unfortunately, these benefits are still beyond the reach of the applications. ATM provides megabits of bandwidth at the data-link layer. However, due to protocol processing and operating system overheads, only a small fraction of this bandwidth is actually available to the applications. Similarly, the connection oriented architecture of ATM provides the necessary infrastructure for a quality of service architecture, but ATM does not really define the specific mechanisms required to guarantee service quality. Consequently, the end-to-end service guarantees are still elusive to the applications.

1.1 Problem Description and Solution Approach

The objective of this dissertation is to bring the potential benefits of ATM within the reach of the applications. More specifically, we address the problems of 1) making the bandwidth available at the data-link layer of ATM visible to the applications, and 2) developing traffic control policies and mechanisms for establishing an end-to-end quality of service architecture. In the following we briefly discuss both problems, and outline our solution approach.

1.1.1 Application Throughput

The job of the networking subsystem at the end-host is to move data between the network interfaces, the host operating system and the networked applications. The throughput of this data path depends primarily on the cost of network protocol processing, and that of moving data from the network interface to the applications through the operating system and vice versa. While the overhead of protocol processing depends largely on the on the processor bandwidth, the cost of data movement is determined by the memory bandwidth of the system. With the increasing gap between the CPU and memory speeds the cost of data movement has emerged as the most significant contributor of end-to-end latency. In many multimedia applications, the end-point of communication is often a device, such as a disk controller or a CODEC, rather than a process running on the CPU. End-to-end data path in such applications crosses the application and the operating system domains multiple times, increasing the cost of data movement even further, and consequently quenching achievable application throughput to an unacceptable level. To get a better

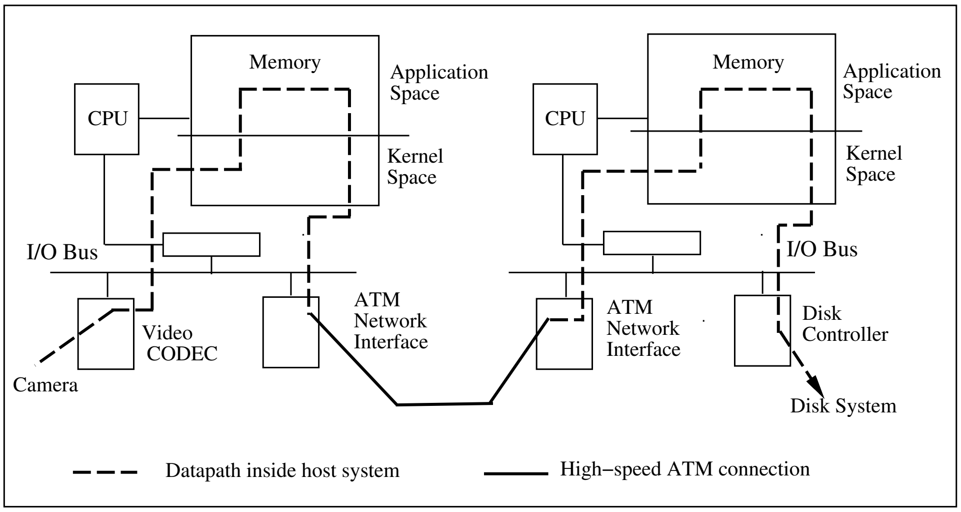

Figure 1.1: A video capture application.

understanding of the problem let us consider the following example.

Figure 1.1 shows an example of a video capture application. The video images captured by a camera and digitized and compressed by a CODEC on one host is transported over an ATM network and is stored on the disk in another host. The important thing to observe in this example is that the end-points of a network connection are not the network interfaces at the end-hosts or an application process, but the devices like the CODEC and the disk-controllers that actually generate and consume data. If we take a close look at the transmit-side data path, we observe that in order to move data from the CODEC to the ATM network interface, data is first copied from the CODEC's buffers to the kernel (operating system) buffers in the main memory. From there the data is copied into the buffers in the application address space. To move data across the network, one would then have to copy data from the application space back in the kernel space, and from there onto the buffer on the ATM network interface. Thus, the data path crosses domain boundaries twice (from kernel to application and back), and the data is copied four times before being sent out on the network. Similar steps are involved in moving data from the network interface to the disk controller at the destination. Besides the overhead of data copying, crossing of domain boundaries inflicts an additional cost of context switching between the application domain and the kernel domain. Also to be accounted in the end-to-end overhead, is the cost of network protocol processing. Hence, it is not very surprising that end-to-end throughput is only a fraction of the network bandwidth available at the data-link layer.

The problem of poor application throughput is not limited to the networked applications only.

All I/O [1] intensive applications suffer from the chronic I/O bottleneck at the end-host. The root cause behind this bottleneck is the inefficient data movements between the I/O devices, such as the network interface, the disk controller, the CODEC, the display controller, etc. In almost all of the existing systems, I/O data path crosses multiple address spaces, causing multiple data copies and context switches. Several research groups [33, 36, 12] identify data copying as the primary problem with respect to system performance. Frequent context switches is the second leading cause of the poor performance [32, 15, 18].

In most systems crossing of domain boundaries (e.g. user to kernel and kernel to user) cause physical copying data from the source domain to the destination domain. Several recent works have addressed the problem of data copying due to crossing of domain boundaries. There are two basic approaches to solving the problem —- *virtual copying* and *use of shared memory*. In the techniques [3, 37, 10, 44] using virtual copying, data is not physically copied when it moves from one domain to another, rather an association is maintained between the physical copy of the data and the domain (or domains) that currently owns it. A physical copy is made only at the application request [37] or when one of the domains attempts to modify the data item. The problem with virtual copying is that the overhead of maintaining the association between the data items and the domains is too high and often comparable with that of physical copying. In the shared memory approach [14, 15], two or more domains statically share a part of their address space and use this memory to transfer data. The cost of maintaining serializability of data accesses and the problem of providing protection against unauthorized accesses often cast doubt on their performance benefits.

In the above example, every copy of data buffer also requires a context switch between the user and the kernel domains. These context switches cannot be avoided because protection boundaries are crossed in copying data from the user to the kernel space and vice versa. The trend in the microprocessor technology towards a larger set of registers, deeper pipelines, and multiple instruction issues is going to further increase the cost of a context switch. The simplest way to avoid context switching (and data copying) is to avoid crossing domain boundaries. In [18], Fall et al. proposes a peer-to-peer I/O model where most of the data flow occurs through the kernel and does not require extra copies or context switches between the user and the kernel domains. Although peer-to-peer I/O transfers would lead to higher performance for I/O intensive applications, it also increases the complexity of the kernel and thus may adversely affect the performance of the kernel and other applications. An alternative approach to achieve the same goal is to contain the data path within the application domain. In [17] Engler *et al.* propose to eliminate all operating system abstractions and provide the applications almost direct access to the hardware. By giving applications more control over the hardware, it is possible to avoid transfer of control to the operating system for I/O services. An immediate consequence of this is a reduction in the frequency of domain boundary

[1]We consider the network interface to be an I/O device.

crossing. The authors also claim that by lowering the interface to the hardware, the management and abstraction of these resources can be customized by applications for simplicity, efficiency, and appropriateness. It is too early to comment on the potential of this approach. The problem of application portability and the overhead of providing low level protection can act against its success.

Although our main objective is to optimize the data path between the network interface and the other devices that consume and generate network bound data, the same principles and mechanisms apply to data transfer between any other I/O devices as well. We propose an I/O architecture that not only helps preserve the network throughput as seen by the applications, but also facilitates device-to-device data transfer in general. In our model, like the network connections in ATM, device-to-device I/O transfer is also connection oriented. That is, a connection is established between the I/O devices before data transfer can take place. Like a network connection, an I/O connection can be multihop. For example, an I/O connection carrying video data from a disk controller can pass through a decompressor on its way to the display. Also, an I/O connection can extend beyond a host, to the devices in other hosts. For example, in the video capture application described above, the I/O connection from the CODEC of the source can pass through the network interfaces of the source and the destination and the intermediate switches, if any, to the disk controller at the destination. Once the connection is established, devices can exchange data with minimal or no intervention from the operating system and the application. We call these devices, capable of transmitting and receiving data on their own, *active* or *autonomous devices*. The advantages of device autonomy, coupled with a connection oriented I/O architecture, are manifold. Most importantly, it helps optimize the I/O data path. Since the application and the operating system no longer need to control I/O transfers, I/O data path can bypass the application and kernel address spaces. A direct consequence of a limited operating system and application intervention is improved I/O throughput. In the context of network I/O, this translates to higher available bandwidth to the applications. A connection oriented I/O architecture also helps optimize network protocol processing using connection specific customizations. We have experimentally analyzed the performance impact of the proposed architecture on networked multimedia applications. On a video conferencing system built around IBM RS/6000s equipped with high-performance video CODECs and connected via 100 Mb/s ATM links, we have shown that a connection oriented autonomous I/O architecture can improve end-to-end network throughput by as much as three times that achievable using the existing architecture.

1.1.2 Quality of Service

Bandwidth does not solve the problem of guaranteeing service quality. The heart of a service architecture providing guarantees on end-to-end performance is the scheduling mechanism used for multiplexing traffic and the switching nodes. The manner in which multiplexing is performed has a profound effect on the end-to-end performance of the system. Since each network connection might have different traffic characteristics and service requirements, it is important that the multiplexing discipline treats them differently, in accordance with their negotiated quality of service. However, this flexibility should not compromise the integrity of the scheme, that is, a few connections should not be able to degrade service to other connections to the extent that the performance guarantees are violated. To protect the system from malicious and ill-behaved sources, it is also important that the access to the network is regulated. That is, each network connection should be associated with a traffic envelope describing the characteristics of the traffic it is carrying, and traffic generated by a source should be passed through a shaper or a regulator to prevent any violation of this traffic envelope. Besides policing, the shaper also smooths the traffic to a form that is easily characterizable.

In the last several years a number of multiplexing disciplines have been proposed [5]. Based on the performance guarantees they provide, these schemes can be broadly categorized into two classes: ones that provide guarantees on maximum delay and ones that guarantee a minimum throughput. The multiplexing disciplines providing delay guarantees [26, 22] typically use priority based scheduling to bound the worst case delay. Depending on the nature of the priority assignment, they can be further sub-divided into static priority schemes and dynamic priority schemes. In a static priority scheme [26], each connection is statically assigned a priority at the time of connection set up. When a cell arrives at the multiplexing node, it is stamped with the priority label associated with its connection and is added to a common queue. The cells are served according to their priority order. There are other alternative approaches to implement a static priority scheduler. In a dynamic priority scheduler, the priority assigned to the cells belonging to a particular connection can be potentially different, depending on the state of the server and that of the connections. Here again, cells are put in a common queue and served in the priority order. Knowing the exact arrival patterns of the cells from different connections, it is possible to bound the worst-case delay suffered by cells from a particular connection in both static and dynamic priority scheduling. One of the serious problems with the schemes described above is that they require traffic reshaping at each node. Priority scheduling completely destroys the original shape of the traffic envelope. Since these schemes require that the exact form of the traffic envelope be known at each node in order to guarantee worst-case delay bounds, traffic has to be reshaped into its original form as it exits a switching node.

The schemes offering throughput guarantees use weighted fair queueing [35, 48] and frame based scheduling [7, 23] to guarantee a minimum rate of service at each node. Knowing the traffic envelope, this rate guarantee can be translated into guarantees on other performance metrics, such as delay, delay jitter, worst-case backlog at a switch, etc. Based on implementation strategies, rate-based schemes can be further classified into two categories: 1) priority queue implementation, and 2) frame-based implementation. The most popular examples schemes using priority queue implementations are virtual clock [48], packet-by-packet generalized processor sharing (PGPS) [13, 35], self clocked fair queueing (SFQ) [25], etc. In all of these schemes, cells are stamped at their arrival with a priority label reflecting the service rate allocated to connections they belong to. They are then put in a common queue, and served in the priority order. While these schemes are extremely flexible in terms of allocating bandwidth in very fine granularity and fair distribution of bandwidth among active connections, they are costly in terms of implementation. Maintaining a priority queue in the switches is expensive. In some cases [35], the overhead of the stamping algorithm also can be quite high. In contrast, frame-based mechanisms are much simpler to implement. The most popular frame-based schemes are Hierarchical-Round-Robin (HRR) [7] and Stop-and-Go (SG) [23, 24]. HRR is equivalent to a non-work-conserving round robin service discipline. In HRR, each connection is assigned a fraction of the total available bandwidth and receives that bandwidth in each frame, if it has sufficient cells available for service. The server ensures that no connection gets more bandwidth than what is allocated to it, even if it has spare capacity and the connection is backlogged. Like HRR, SG also is a non-work-conserving service discipline. It tries to emulate circuit switching in a packet switched network. Due their non-work-conserving service policy, both SG and HRR fail to exploit the multiplexing gains of ATM. Another important drawback of SG and HRR, and all framing strategies for that matter, is that they couple the service delay with bandwidth allocation granularity. That is, the finer is the granularity of bandwidth allocation the higher is the delay suffered at the server.

In almost all the schemes discussed above, the shaper is assumed to enforce a specific rate constraints on the source, which is typically a declared peak or mean rate. In other words, the scheduler assumes a peak or mean rate approximation of the original source. Although a single rate characterization simplifies the task of the scheduler, it has a detrimental impact on system performance. Most multimedia applications generate inherently bursty traffic. Hence, enforcement of a mean rate results in a higher delay, while peak rate enforcement leads to a lower network utilization. To alleviate this problem we propose to use multirate shapers. A multirate shaper enforces different rate constraints over time windows of different lengths. For example, a dual-rate shaper can enforce a long term average rate and a short term peak rate. Multirate shaping allows a more precise characterization of bursty traffic which can potentially be used by the scheduler to exploit the multiplexing gains of ATM.

Although multi-rate shapers better characterize bursty traffic, it is the scheduler which determines how that information is used to improve system performance. As mentioned above, an ideal scheduler should also provide flexibility and protection, and should be simple enough to be implemented at the high-speed switches. We propose a simple work-conserving scheduling mechanism designed to integrate the flexibility and fairness of the fair queueing strategies with the simplicity of frame-based mechanisms. The scheduling mechanism, which we call Carry-Over Round Robin (CORR), is an extension of simple round robin scheduling. Very much like in round robin scheduling, CORR divides the time-line into allocation cycles, and each connection is allocated a fraction of the available bandwidth in each cycle. However, unlike slotted implementations of round robin schemes where bandwidth is allocated as a multiple of a fixed quantum, the bandwidth allocation granularity can be arbitrarily small in our scheme. Also, unlike the framing strategies like SG and HRR, ours is a work-conserving discipline, and hence unused bandwidth is not wasted but is fairly shared among the active connections. We have shown that when used in conjunction with multi-rate shaping, CORR is very competitive with much more complex mechanisms, such as PGPS and SFQ.

1.2 Summary of Contributions

In this dissertation we have addressed two important problems hindering the ubiquitous deployment of distributed multimedia applications —- 1) a lack of operating system support for network intensive applications, and 2) a lack of network support for quality of service guarantees. In the following, we briefly summarize our contributions in both of these areas.

To enhance operating system support for network intensive applications, we have proposed an I/O architecture. The proposed architecture limits the operating system involvement in I/O transfers by migrating some of the operating system's I/O functions to the I/O devices. By allowing autonomy to I/O devices and by introducing the notion of a connection oriented I/O architecture, we manage to limit the operating system and application involvement in I/O transfers. Consequently, it not only improves network throughput, but also addresses the more general problem of chronic I/O bottleneck of the current generation of operating systems. We have experimentally demonstrated the performance impact of the proposed I/O architecture on networked multimedia applications. We have developed a video conferencing system using the principles of device autonomy and connection oriented I/O transfers, and we have achieved a three-fold performance improvement over the system using the conventional I/O model.

To address the problem of quality of service support in ATM networks, we have proposed a simple cell scheduling mechanism, named carry-over round robin (CORR). Using analytical techniques, we have analyzed the delay performance of CORR scheduling assuming multi-rate sources. To the best of our knowledge, our source model is the most general among all the related studies published in the literature. We have derived closed form bounds for end-to-end delay when CORR is used in conjunction with multi-rate shapers. Besides providing guarantees on delay, CORR is also fair in distributing the excess bandwidth. We show that albeit its simplicity, CORR scheduling discipline is very competitive with more complex disciplines.

1.3 Organization

The rest of dissertation is organized as follows. In chapter 2 we introduce the concept of device autonomy. The autonomous devices are at the heart of the connection oriented I/O architecture proposed in this chapter.

Chapter 3 is devoted to the experimental validation of the I/O architecture proposed in chapter 2. In this chapter we describe in detail the architecture of a high-performance video conferencing system developed using the principles of device autonomy and connection oriented I/O. We present detailed performance results to demonstrate the impact of the proposed I/O architecture on the performance of the network subsystem in particular, and that of the I/O subsystem in general.

In chapters 4, 5, and 6 we discuss different aspects of quality of service management in ATM networks. In chapter 4, different shaping mechanisms used to regulate traffic at the edge of the network are presented. Here, we introduce the concept of multi-rate shaping and characterize the traffic envelopes defined by composite moving window, jumping window, and leaky bucket shapers.

In chapter 5, we present detailed algorithmic description of the Carry-Over Round Robin scheduling discipline and analyze some of its basic properties.

Chapter 6 is devoted to the evaluation of the shaping and scheduling mechanisms. We derive closed form bounds on the worst-case end-to-end delay when CORR is used in conjunction with multi-rate shapers. We also analyze the fairness properties of CORR scheduling.

We conclude in chapter 7 by noting the contributions of the work and laying a roadmap for future extensions.

Chapter 2

Autonomous Device Architecture

Multimedia applications typically involve moving large volumes of time-sensitive data between devices and peripherals attached to the same host or to different hosts. For example, in a video recording application, data captured from a camera is compressed and coded by a CODEC and stored on a disk, all of which can be done on the same host. Alternatively, images can be captured by a camera attached to one host, compressed by a CODEC in another, and stored on a disk in yet another host, all connected via a network. In either case, providing adequate system support requires an infrastructure that is capable of moving hundreds and thousands of megabytes of data from one end to the other in a timely and orderly manner. Although we have witnessed great gains in hardware performance in recent years, software performance has not improved commensurately. The inadequacy of the current generation of operating systems (OSs) in supporting I/O intensive applications is a major deterrent in the wide-spread deployment of multimedia applications. In this chapter, we propose a new architecture designed to alleviate the I/O bottleneck in the current generation of OSs.

The rest of the chapter is organized as follows. In section 2.1 we review related works. The guiding principles behind the proposed architecture are discussed in section 2.2. Section 2.3 is devoted to a description of the basic components of the architecture. Alternative data paths for device-to-device data transfer are discussed in section 2.4. Flow control and data processing issues are addressed in sections 2.5 and 2.6, respectively. We summarize the contributions of this chapter in section 2.7.

2.1 Current State of Art

Most of the conventional I/O system architectures stem from the Multics system of the late 1960's. Given the enormous changes in the application profile, it is not surprising that the OSs fail to provide the sheer performance required or the predictability of performance desired by multimedia applications. Although it is not immediately obvious, both of these limitations arise from the processor-centric viewpoint adopted by the current OSs. The processor-centric viewpoint is embodied in the notion of a process. A process, by definition, is an instantiation of the current state of a computation and a place to keep a record of resources reserved for the computation. The most important omission from this notion is the communications and the I/O operations that take place. Nothing is indicated about the resources to be used for communications and I/O and their expected usage pattern. Implicit resource demands of communications and I/O make it hard to design OSs which would provide predictable performance. Another aspect of the processor-centric viewpoint of current OSs is manifested in the unwarranted involvement of the processor in I/O operations. In the current systems, the processor is involved in initiating I/O operations and moving data, even when the applications perform no processing on the data.

In order to understand why I/O intensive multimedia applications suffer from the existing OS architecture, let us consider the typical activities that take place in the OS while supporting a basic multimedia application. Consider the video recording application described in the last chapter. Here, in order to move a data packet from the CODEC to the network interface, the application has to first make a system call to read data from the CODEC. As a result of the read call, the control is transferred to the kernel. In the kernel, the system call is translated into a sequence of device specific operations, ultimately resulting in copying of data from the device buffer to buffers in the application space. Once data is received in the application space, the application requests a network send. Once again the control is transferred to the kernel. The kernel translates the system call into device specific actions, and eventually data is moved from the application buffer to the buffer on the network interface. Hence, to move a data packet between two I/O devices, the control is transferred back and forth between the application and the kernel domains twice, resulting in four context switches and multiple data copies. This results in significant degradation of system performance by keeping the host bus busy, and consuming processor cycles and memory bandwidth.

Several recent works have addressed the problem of data copying due to crossing of domain boundaries. The simplest way of avoiding data copy is to use shared memory [14, 15]. In [37] inter-domain transfers are optimized by encapsulating data in a sequence of *pallets* (contiguous virtual memory address). Data is mapped into a receiving domain only when requested by the application. Mach [3] and its predecessor Accent [20] use a scheme known as *copy-on-write* to avoid unnecessary data copying. A number of techniques rely on the virtual memory system to provide copy-free cross domain transfers. *Virtual page remapping* [10, 44] unmaps the pages containing data units from the sending domain and maps it into the receiving domain. *Shared virtual memory* [41] employs buffers that are statically shared among two or more domains to avoid data transfers. The problems with using shared memory is that the unit of sharing is typically a page and data must be aligned to the page boundary. There are also tricky protection issues which often cast doubt on their viability.

Frequent context switches is another leading cause of the poor performance [32, 15, 18]. In the above example, every copy of data buffer also requires a context switch between the user and the kernel domains. These context switches cannot be avoided because protection boundaries are crossed in copying data from the user to the kernel space and vice versa. The trend in the microprocessor technology towards a larger set of registers, deeper pipelines, and multiple instruction issues is going to further increase the cost of a context switch. In order to reduce the frequency of context switches, Bershad et al. [6, 31] proposes building extensible kernels that would include some part of an application to run in the kernel. In [18], Fall et al. proposes a peer-to-peer I/O model where most of the data flow occurs through the kernel and does not require extra copies or context switch between the user and the kernel domains. Although peer-to-peer I/O transfers lead to higher performance for I/O intensive applications, it also increases the complexity of the kernel and, thus, may adversely affect the performance of the kernel and other applications.

Moving I/O data through the CPU and the memory subsystem also harms the cache performance. The processor's primary and secondary caches are filled with data that is used only once, leading to flushing of caches of other data. This would result in more processor stalls later for other programs. On the other hand, if the data transferred is allowed to bypass the cache, the networked application will experience an increased latency due to cache misses when processing message headers and control messages. The increasing gap between processor speed and memory bandwidth means that the cost of delivering data to a wrong place in the memory hierarchy would also rise proportionately. Carter et al. [8] proposes to address this problem by integrating the memory controller with the network interface.

One solution to poor I/O throughput in the system is to move the I/O data path away from the application and the kernel. With the availability of cheaper and faster microprocessors, most devices now a days are equipped with powerful on-board processors. Taking advantage of this trend, it is possible to make devices exchange data independently, instead of the application driven push-pull control. Data can be transferred from one device to another without processor or main memory getting in the way. Such devices would not only move data, but also do rate-matching, perform limited data processing, and recover from occasional data losses. Once the connection is established between two devices by an application program, there would be no further need to involve the main processor in the data transfer. In the context of the example shown above, a connection would be established between the camera controller and the network controller, and then frames would be captured at a rate at which the network agrees to send it out. Once the connection is established, the controllers would never need to interrupt the kernel until one desires to terminate the capture mode or wishes to exchange control information, such as controls for camera positioning, exception handling, etc.

Smart devices capable of direct data transfer address the problem only partially. A disk controller that stores a file would also require someone to manage disk space (allocate disk sectors, link them in a file system, etc.) If this is provided by an OS service, each data transfer would still require some action from a processor. We define a new term, *autonomous devices*, for a device that can be a source or a sink of data without repeated interactions with the main processor, that is, OS services. Several recent works on redesigning workstations [19, 4, 47, 27] have explored the notion of using direct transfers between certain types of devices such as a camera and a display. The I/O architecture proposed in this chapter is guided by the same philosophy. The only difference is that instead of redesigning the workstation hardware, we rely on a more evolutionary software approach.

2.2 Guiding Principles

We believe that a pragmatic approach to alleviate the I/O bottleneck in the OS is to move I/O devices out of the OS control. We call these devices, capable of handling I/O transfers without OS control, autonomous devices. With the autonomous devices as the sources and the sinks of data flows, we propose a connection oriented I/O architecture in an attempt to eliminate the I/O bottleneck in the OS. The central tenets guiding our design effort are:

- Allowing devices to operate autonomously and communicate with each other without operating system or application control.
- Developing a connection oriented architecture for I/O transfers between autonomous devices.

In the following, we elaborate on our ideology and explore some of the advantages of the new architecture.

2.2.1 Device Autonomy

A device is a source or a sink of data. Monitors, keyboards, disk and tape drives, network interfaces are all devices [1] generating and/or consuming data. In the conventional model of computation, an application proceeds by directing a device to do something. The service that is requested is at a high-level of abstraction (e.g., read a line from the named file, open a file for reading). This request is translated into a system call. The OS kernel handles the system call by breaking down the operation into a sequence of low-level commands to a device. Transferring large amounts of data, such as listing a file, thus involves repeated interaction with the kernel, as well as passing data through the processor. Flow control between devices is done by buffering data in the kernel

[1] Main memory can be treated as a device.

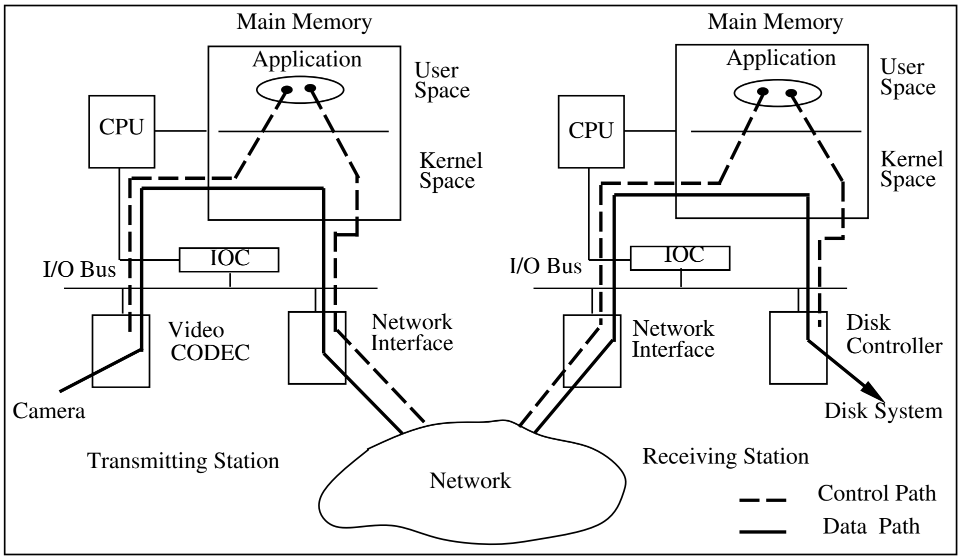

Figure 2.1: Seperation of control and data flows.

and sometimes in the application space as well. We refer to this type of device as a *passive* device. All devices are currently treated as passive devices.

An *active* device is defined as one that is capable of handling data transfers, buffering data, and matching data rates with flow control mechanisms. An autonomous device is defined as an active device that is capable of de-multiplexing traffic according to application-specific contexts provided to it by a processor. An *autonomous* device is an *active* device with simple data manipulation capabilities such as byte swapping, checksum operation, and more complex actions such as BITBLT operations used in bit-mapped display devices.

In an autonomous device architecture, the OS is viewed as the one that establishes the necessary contexts at devices at the beginning of a data transfer. The control may be transferred to the OS only for exception handling and not during the data flow.

2.2.2 Connection Oriented I/O

In our model, all I/O transfers are connection oriented. A connection is established between the source and the sink before the beginning of data transfer. The application and the OS participate in the connection setup phase only by setting up the context at the devices and initializing the connection states. Once the data transfer begins, it proceeds autonomously between the source and the sink, transparent to the kernel and the application. Besides lesser involvement of the OS and the application in data movement, the connection abstraction also helps build a quality of service architecture in the I/O subsystem. In order to support guaranteed quality of service, each connection has to be associated with a traffic envelope, and the connection setup procedure has to include an admission control test to determine if the new connection can be supported.

In many ways the connection oriented I/O architecture resembles the ATM network architecture. In some sense it is the generalization of connection oriented network I/O to other froms of I/O activity inside the end-host.

One of the major advantages of connection oriented architecture is that it allows separation of control and data flows. Consequently, data transfer mechanisms can be fast and dumb, while the control mechanisms can be as complicated as necessary. For a better understanding of its implications, consider the video capturing example. The problem with the application architecture shown in figure 1.1 is that both control and data flows between devices are through an application process. In this example, only the control flow needs to pass through the application. Since data and control cannot be separated in the existing architecture, both control and data flows pass through the application, severely limiting the throughput of the data path. In a connection oriented architecture, we can alleviate this problem by using separate connections for control and data flows. In the architecture shown in figure 2.1, data paths connect the devices that generate/consume data directly, while the control flow still passes through the application. That is, the application still remains in control of the data flow without being directly involved in moving data.

2.3 System Components

The proposed architecture is general enough to be applicable to a large class of systems. In the most general form we assume that a host system consists of one or more CPUs, memory subsystem, and I/O devices capable of performing autonomous operations. We do not make any assumption regarding how the devices are interconnected. They can be connected through a bus or a switch. We also do not make any assumption on the degree of autonomy exercised by the devices. The architecture is general enough to support autonomous, active, as well as passive devices.

In our system all forms of device to device and application to device data transfers are preceded by a connection setup procedure between the source and the sink. After a connection is established, the involvement of the OS in data transfer depends on the degree of autonomy exercised by the devices. If the source and the sink are passive, data still flows through the kernel. For active and autonomous devices, the responsibility of the OS can range from exception handling to interrupt processing, depending on the sophistication of the devices in terms of the functions they support.

To support notion of connections, devices and device interfaces need to be altered. We introduce the abstraction of I/O channels to support I/O connections at the device end. An I/O channel is a resource sub-unit of a device. Associated with each I/O channel are resources, such as buffers reserved for the channel, channel state variables, and channel handlers. Channel handlers are responsible for transmit and receive processing at the source and the sink devices, respectively. Since each channel can be associated with a different set of handlers, channel handlers can be used to customize I/O services on a per connection basis. The channel handlers can execute either on the microprocessor on the device or on the main processor depending on the type of the device or by design choice.

2.4 Data Flow

One of the important assumptions in the proposed I/O model is the availability of data streaming mechanisms between devices. In the following we discuss in detail different forms of hardware and software data streaming mechanisms (see figure 2.2) that can be used by devices with different degrees of autonomy.

Hardware Streaming: In this approach, data is transferred directly from the source to the sink bypassing the main memory. Supporting hardware streaming requires appropriate hardware support for device-to-device data transfers including demultiplexing facility and support for necessary data format conversion. Consequently, passive devices cannot use hardware streaming. Several systems have been developed with support for some variety of hardware streaming. IBM's MicroChannel bus is a popular example of a streaming bus. It defines peer-to-peer transfers as data streaming occurring between two bus masters. One master acts as a controlling bus master and the other as a slave. This feature has been used in the AURORA [11] testbed. There are several other examples of hardware streaming between devices using customized hardware interfaces [1, 4, 47].

Hardware streaming has encountered resistance because of its lack of integration with the host system. Since the applications have no access to the data, they are constrained by the functionality provided by the adapters. The adapters that support a fixed set of capabilities provide applications with little flexibility in terms of data processing.

DMA Streaming: A second approach to data streaming is to use DMA transfers between devices. The CPU may control data transfers but does not participate in the data movement. The main advantage of DMA streaming over hardware streaming is that it does not require any special hardware, except for DMA support at the source and sink devices. All active and autonomous devices can take advantage of DMA streaming. During the course of data transfer, the source DMAs data into the main memory buffer, which is then DMAed into the sink device. With the help of an intelligent I/O controller, buffering in the main memory

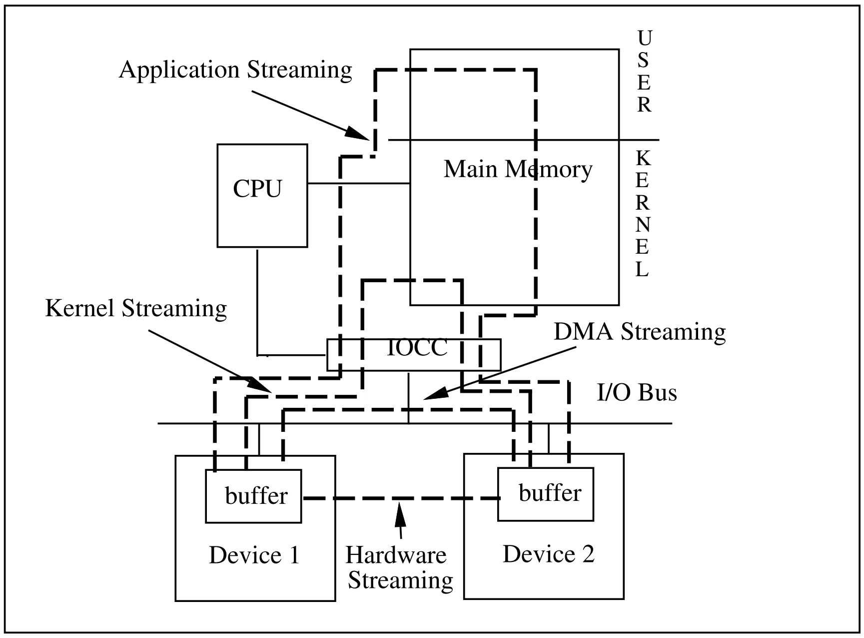

Figure 2.2: Alternative data paths.

can be substituted by buffering in the I/O controller itself. This improves the sustainable throughput since the DMA transfer does not have to compete with the concurrent CPU activity for main memory accesses. Two I/O bus trips are required for DMA streaming, and consequently throughput is bounded by half of the I/O bus bandwidth.

DMA streaming suffers from the same shortcomings of the hardware streaming. Since the data path is transparent to the CPU, data processing is limited to functions performed by the adapters.

Kernel Streaming: Both hardware streaming and DMA streaming suffer from the lack of flexibility in data manipulation. In order to provide applications with complete control on the data, we need to move data through the kernel and one or more application domains. Clearly, data must pass through the CPU/cache at least once. In most systems, crossing of domain boundaries require data copying, leading to further reduction in achievable throughput. If all data manipulation can be performed in the kernel mode, overhead due to kernel-to-user domain boundary crossing can be avoided. We refer to in-kernel but application transparent data transfer between devices as kernel streaming. All devices are capable of using kernel streaming.

Although kernel-streaming offers full programmability, data manipulation has to be performed in the kernel. Consequently, applications are limited by the functionalities provided by the kernel for data manipulation. Some of the newer OSs are trying to address this problem by providing interfaces to execute application code in the kernel domain. The problem of protection against faulty and malicious user code casts doubt on this approach.

Application Streaming: We refer to data transfer between devices through kernel and one or more application domains as application streaming. Most of the existing applications follow this model of data streaming. Clearly, application streaming provides the most flexible interface for data manipulation, but only at the cost of data transfer throughput.

2.5 Flow Control

Flow control refers to the task of speed-matching between the data source and the data sink. In the traditional application streaming, the application controls the flow by exercising a push-pull control on the source and the sink. That is, a source is blocked until the sink has consumed at least a part of the the outstanding data. Data is buffered in the application and in the kernel to absorb temporary disparity in the rates of data generation and consumption. Unfortunately, application driven control is not an available option in application transparent data streaming. Also, a push-pull control is not the most ideal form of flow control for high-speed data transfers.

We propose to use source rate control for controlling autonomous data flows. In our model, each I/O connection is associated with a traffic envelope which describes the characteristics of the data flow between the source and the sink of the connection. For example, a flow envelope may specify the peak and the mean rates of the flow. As a part of connection setup, an admission control test is performed to check if the resources available at the sink are sufficient to consume the data generated by the source. If sufficient resources are available, the connection is admitted, and the traffic envelope is communicated to the source. It is the responsibility of the source to conform to this traffic envelope. If sufficient resources are not available at the sink, the connection is aborted.

The advantage of source rate control is that the overhead of flow control during data transfer is minimal. Hence, it is extremely suitable for high-speed data streaming. The flip side of this approach is that once the resources are committed to a connection, they cannot be used for other connections, even when they are under-utilized.

2.6 Data Processing Modules

The proposed architecture is an exact fit for the applications that move large volumes of data but perform very little processing on it. Although most of the multimedia applications fall into this model, there are many which do not. Hence, it is important to add support for data manipulation

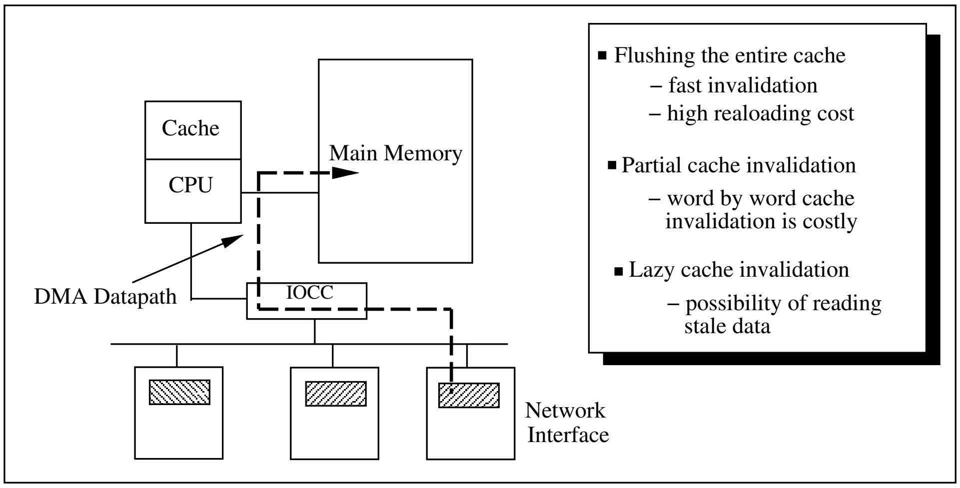

Figure 2.3: DMA data path.

capability to enhance the generality of the model. We use channel handlers for this purpose.

The handlers associated with each channel can be used to process data both at the source and at the destination in a channel specific way. In order to offer full flexibility in data processing, we need to provide an interface for applications to specify the handlers to be used with a particular channel. Many modern operating systems, such as IBM's AIX and SUN's Solaris provide dynamic loading facility to load code modules into a protected address space. This facility is widely used to load device drivers and other kernel modules selectively depending on hardware availability. It can be easily extended to load user code to an attached peripheral device rather than the kernel address space. Hence, user specified data processing modules can be attached to I/O connection. However, the problem of protection against erroneous and malicious code module is still remains to be addressed.

Even when application code cannot be used as processing modules, connection handlers could still be quite useful for connection specific data processing. For example, consider the scenario sketched in figure 2.3. Video data received over the network is DMAed into main memory for subsequent display. In many systems, such as IBM's RS/6000, the DMA data path bypasses the cache. Hence, to maintain cache consistency, after a DMA operation the network device driver flushes the data cache. However, the chances that a stale cache line is accessed by the CPU if the cache is not flushed are quite small. While it is important to flush the cache where high data fidelity is mandatory, avoiding cache flushing not only saves the overhead of the flush operation, but also improves cache performance. If we identify network connections where data corruption does not lead to catastrophic consequences, we can improve system performance by avoiding cache flushing [16]. The video application described above is a perfect candidate for such optimizations.

2.7 Summary

In this chapter we have proposed a connection oriented autonomous I/O architecture. Our approach of delegating more responsibility to the devices is fundamentally different from most of the solutions proposed in the literature. Device-to-device autonomous data transfers with minimal OS and application intervention have the potential to eliminate the I/O bottleneck in the operating system. The notion of I/O channels not only provides a uniform abstraction for all I/O activity in the system, but also enables channel specific customization of I/O services. It also establishes an infrastructure for performance guarantees on I/O operations.

Chapter 3

Experimental Evaluation

We have designed and implemented a high-performance video conferencing system using the principles of device autonomy and connection oriented I/O transfers proposed in chapter 2. In our system, video images captured from a camera is digitized and compressed in real-time by a high-performance prototype CODEC. The compressed video data is then transferred over an ATM network to the destination(s), where it is decompressed and displayed in real-time. Even with specialized hardware for video/audio processing and high-speed network connectivity, our first prototype system using conventional application streaming to move audio and video data between the CODEC and the ATM network interface did not meet our expectations in terms of audio/video quality. After a careful profiling we identified the two most important factors limiting system performance: (1) the data path crosses domain boundaries (application/kernel) several times, leading to unnecessary data copying and context switching, and (2) protocol processing overheads of UDP/IP [1]. To improve system performance, we have implemented a second prototype where the CODEC and the ATM network interface communicate with each other in a 'semi-autonomous' fashion. The optimized data path bypasses application address space and hence eliminates unnecessary data copies and context switches. To optimize network protocol processing overhead, we use a light weight native ATM protocol stack instead of UDP/IP. The optimized system can support two-way and multi-way conferences using full-motion, very high resolution video and CD quality audio.

The rest of the chapter is organized as follows. In section 3.1 we describe the system platform used for the experiments. In section 3.2 we present the details of the implementation and performance profiling of the base system. Architectural and implementation details of the optimized system are discussed in section 3.3. We compare the performance the base and the optimized system in section 3.4. The contribution of this chapter is summarized in section 3.5

[1]We use UDP/IP running over ATM for data transfers between the hosts.

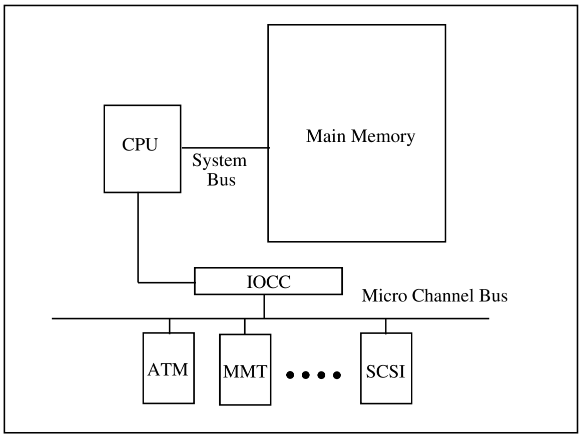

Figure 3.1: RS/6000 architecture.

3.1 System Platform

The system platform for our experiments is the IBM RS/6000 running AIX 3.2.5 [2]. Each machine is equipped with an IBM Turboways ATM network interface [2], and the MMT [3] prototype adapter [1]. The ATM adapter implements the ATM adaptation layer 5 (AAL5), which is responsible for all segmentation and reassembly of datagrams and detection of transmission errors and dropped cells. The MMT adapter performs video and audio capture, compression, decompression, and playback.

Figure 3.1 shows the architecture of the RS/6000 system. The CPU and the main memory are connected through the system bus. The I/O adapters, including the ATM adapter and the MMT adapter, sit on the MicroChannel I/O bus. Data transfer from the I/O adapters to the main memory and vice versa goes through the I/O controller (IOCC). Data transfer on the MicroChannel can be in strides of 8, 16, or 32 bits with a cycle time as low as 20ns. In the streaming mode both data and address buses can be used for data transfer giving rise to 64-bit wide data path with a cycle time of 10ns. In the following section we briefly describe the architecture of the MMT and ATM adapters.

MMT Adapter. The MMT adapter (figure 3.2) consists of a video/audio capture (VAC) subsystem and a compression/decompression (CODEC) subsystem. The VAC subsystem accepts NTSC video and microphone or line audio input and generates YUV422 digital video and PCM/ADPCM audio and vice versa. The CODEC subsystem accepts YUV422 digital video at the video input for compression and generates YUV422 digital video at the video

[2] AIX is a UNIX like Operating System.

[3] MMT stands for Multimedia Multi-party Tele-conferencing.

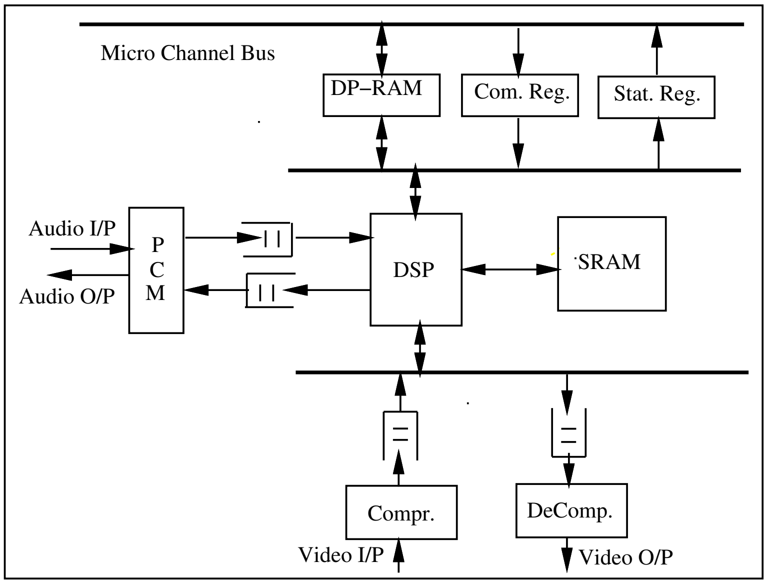

Figure 3.2: MMT adapter.

output after decompression. It supports full-duplex real-time video compression and decompression at frame rates up to 30 frames/second. The whole system is controlled by a dedicated DSP[4] processor. The DSP has access to a 256 Kbyte SRAM and a 16 Kbyte dual port memory (DP-RAM). The SRAM is used for smoothing and mixing of video and audio streams. The dual port memory is used to communicate with the system.

The adapter currently supports the ISO Motion-JPEG [46] standard. The video processing unit consists of two motion JPEG engines, one for compression and the other for decompression. They can support different frame rates and resolutions. The video input is fed through the video frame rate control logic to the compression engine. The data rate of the compressed data stream can be controlled by programming the quantizer to as low as 128 Kbits/sec and as high as 10 Mbits/sec [1]. The CODEC is capable of mixing up to 32 video streams [42] in the compressed domain and presenting them in multiple video windows.

ATM Adapter. The IBM ATM adapter [2] is responsible for performing the AAL5 functionalities. It features a dedicated i960 processor and a specialized chipset to handle AAL5 segmentation and reassembly in hardware. The adapter is equipped with a DMA master and 2MB of on-board memory. It can support up to 1024 network connections with an aggregate throughput of 100 Mb/s.

[4]DSP stands for Digital Signal Processor.

3.2 Base System: Implementation and Profiling

In this section, we describe the architecture and the performance of the first prototype implementation of the conferencing system. Our objective behind this effort is to understand the limitations of the conventional I/O architecture. In the following description we limit our attention to the Video Audio Support Unit (VASU), the subsystem responsible for moving audio and video data. For a detailed description of the entire system refer to [39].

3.2.1 Implementation

The first prototype, referred to as the base system, uses classical UNIX model of peer-to-peer communication. Peer VASUs running on different hosts open communication channels to each other using datagram sockets running over UDP/IP on ATM AAL5. In the base system, no modification to the devices or the device interfaces are made. We have written AIX drivers for MMT and ATM adapters following the standard UNIX paradigm for character and network devices, respectively. A simple optimization is incorporated in the MMT driver to reduce data copying. In the following we describe some of the implementation details.

MMT Device Driver. The MMT device driver follows the well-known *config-open-close-read-write-ioctl* UNIX paradigm. The config call initializes the MMT device by loading appropriate micro-code and initializing the device state. The open and close calls are standard. To optimize data movement between device buffers and applications we map MMT buffers into kernel address space. This allows us to move data directly from the device buffer to the application space, and vice versa, without copying into intermediate kernel buffers in the main memory. This optimization saves one data copy for each read and write operation. We have implemented several device specific ioctl (I/O control) calls. These include registering user processes with the device so that asynchronous call-backs can be made to the appropriate process when data is ready in the device, or when the device is ready to accept data. There are ioctl interfaces to register asynchronous exception handlers, to change device configurations, such as quantization, frame rate, etc. We have also added ioclt calls for turning device profiling on and off and generating device statistics.

ATM Interface Driver. The ATM device driver consists of several sub-layers in which the lowest layer interfaces with the ATM adapter and the highest layer interfaces to the AIX network subsystem. The standard interface for the ATM device is the IP network interface and the device supports the classical IP over ATM model [29]. In addition to this IP interface, the ATM device driver also provides a low-level UNIX device interface. Hence, the VASU can access the ATM network using either one of these two mechanisms. While using the low-level interface, the device is opened via the open system call. Before data can be sent or received (write and read respectively), virtual channels (VC) have to be established. This is done via ioctl calls. There are ioctl interfaces for opening and closing connections. In opening a VC its traffic characteristics (peak cell rate, sustainable cell rate, burst length) and other connection parameters (simplex/duplex, service priority, AAL type) are specified. In addition, there are ioctl calls for resetting the sender and receiver entities for a certain VC.

The VASUs use the file I/O interface to open, close, read, write data from the MMT and use the socket API to open and close network connections and send and receive system calls to exchange video and audio data over the network. On the transmitting side, audio and video data is captured, digitized and compressed by the MMT. Compressed data is packetized by the DSP and an interrupt is sent to the driver indicating that data is ready to be read. The driver, in turn, sends a signal to the VASU. The VASU, upon receipt of the signal, reads the data and sends it over the UDP/IP socket to its peer. On the receive side, VASU receives data on the UDP socket. Once data is received from the network interface, the VASU writes it into the MMT buffer using the write system call provided by the MMT driver.

3.2.2 Data Path Profiling

The quality of video and audio in the base system was far below our expectations. Clearly, neither the network nor the the CODEC was the bottleneck. In order to identify the system bottlenecks we instrumented the transmit and receive data paths and performed a thorough profiling of the system.

Figures 3.3 and 3.4 show the transmit and receive latencies in the base system. These measurements have been taken on an RS/6000 Model 530H with a 32 Mbyte memory and a 50 MHz processor. The measurements were taken using the system's real-time clock which is an integral part of the RS/6000 architecture. This clock can be accessed by any process by reading two 32-bit clock registers with microsecond granularity. We used a two-instruction assembly language function to read the clock registers with negligible overhead. To store the statistics gathered, we allocated temporary buffers in the kernel. We also added ioctl calls so that the applications can access the statistics.

As shown in figure 3.3, the latency on the transmit side comprises of two major components – MMT read and network send, each of which is a system call. The MMT read overhead can be further broken down into the cost of context switching and the overhead due to data copying across domain boundaries. Note that, the data copying in this particular case is a copy from the MMT adapter to the main memory across the I/O bus. The network send overhead include the cost of

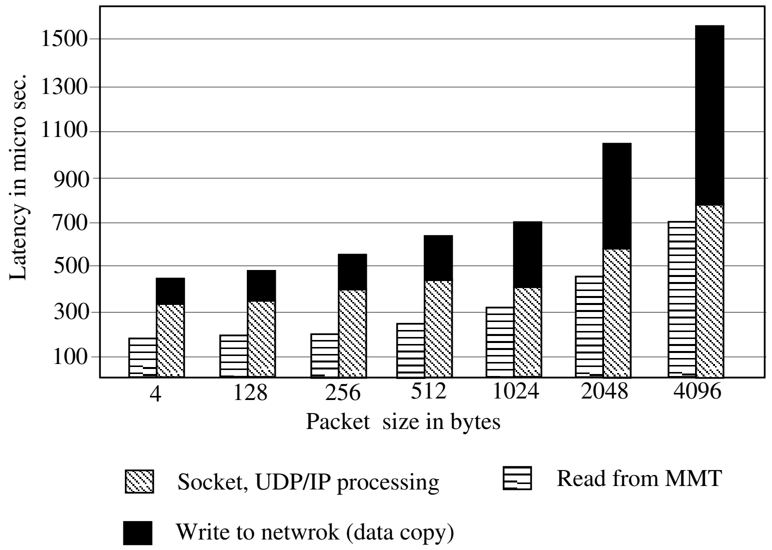

Figure 3.3: Transmit latency in the base system (in microseconds).

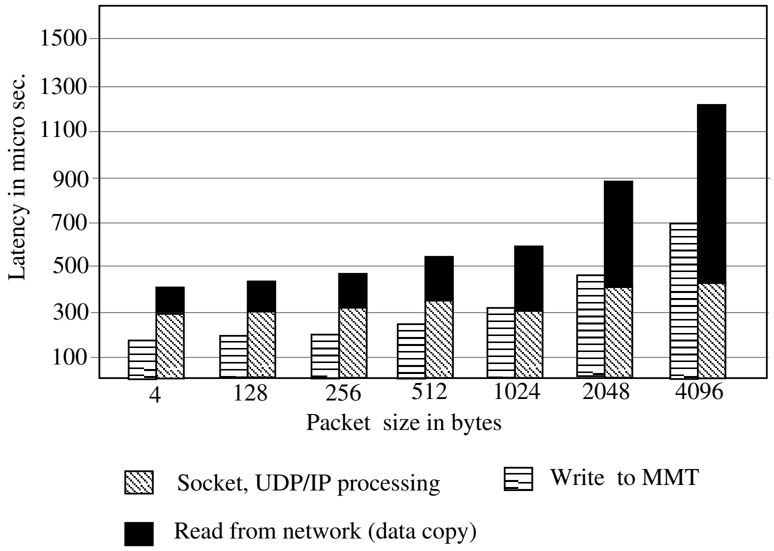

Figure 3.4: Receive latency in the base system (in microseconds).

context switching, two data copies, and network protocol processing. First, data is copied from the user buffer to kernel mbufs [5], a main memory to main memory copy. The second data copy is a copy from the kernel mbufs to buffers on the adapter across the I/O bus. Hence, the entire transmit operation involves two context switches and three data copies, one main memory to main memory and two between the main memory and the memory on the I/O adapters. Similarly, the

[5] mbufs are kernel managed buffers [45].

receive data path consists of two system calls (see figure 3.3), one each for network receive and MMT write.

Quite expectedly, the overheads incurred by read and write calls to MMT are approximately the same. They are primarily due to context switching and data copies across domain boundaries. However, the socket sends and receives are significantly more expensive than MMT read and write calls. Besides the cost of data copying across domain boundaries and between the adapter buffer and main memory, they also include protocol processing overheads. In MMT read and write operations we save one data copy by copying data directly from the adapter buffer to application buffer and vice versa. When data is moved between the application buffer and the ATM interface buffer, data is first copied into to a kernel buffer in the main memory, and then copied into the application buffer or the adapter buffer depending on the direction of data flow. This extra copy into the kernel buffer cannot be avoided since the CPU performs network protocol processing on the data.

If we convert these latency figures into equivalent throughput, we observe that only a very small fraction of the network bandwidth is actually available to the application. A large portion overhead can be attributed to the circuitous data path between the MMT and ATM adapters. There is no reason for video and audio data generated by the MMT to pass through the buffers in the kernel and the application address space on its way to the ATM network interface. Similarly, moving network data from the ATM adapter through kernel and application buffers in the main memory to the MMT adapter is not an example of the most optimal data path. Unified flow of control and data and lack of autonomous device operation in the current generation of OSs is the only reason why data has to pass through the application. By taking advantage of I/O connections we have separated control and data flows in the optimized system. This allows us to set up a MMT to ATM direct data path bypassing VASU.

From figures 3.3 and 3.4 we also observe that one third of the overhead in the transmit and receive latencies is contributed by the protocol and socket processing overheads. A large component of this overhead stems from the protocol redundancy in the existing layered protocol architecture, such as the Internet protocol family. The ATM adaptation layer, implemented in hardware, provides a rich set of functionality, such as connection management, flow control, segmentation, and reassembly. However, to maintain the same interface to all the link layers, the Internet protocol suite (TCP/IP and UDP/IP) ignores the special features provided by ATM, and many of the functionalities are replicated in higher layers. In order to exploit the rich set of functionality provided by the ATM network, in the optimized system we use a native ATM stack running over AAL5 instead of UDP/IP. The connection oriented I/O architecture provides the necessary infrastructure for an elegant implementation of the native ATM stack.

3.3 Optimizations with Autonomous I/O

Based on the lessons learned from the implementation and evaluation of the first prototype, we have designed and implemented the second prototype with two primary objectives:

- Optimize the data path between MMT and ATM adapters by using DMA-streaming. Although hardware streaming would have been the best option in terms of performance, we opted for DMA streaming mainly due to logistic limitations [6].
- Optimize protocol processing overhead by using a light weight native ATM protocol stack instead of UDP/IP.

To facilitate autonomous data streaming and connection oriented I/O, we introduced the notion of I/O channels. A channel is a resource sub-unit on particular device. An application opens I/O channels to devices in order to access their services. Channels can be spliced to establish application transparent direct data paths between autonomous devices. The I/O architecture implemented in the optimized system is structured around the notion of I/O channels. Following we describe the architectural and operational details of the system.

3.3.1 Driver Extensions

In order to incorporate the notion of device autonomy and I/O channels we had to rewrite the device drivers for MMT and ATM adapters. The extended drivers support data streaming and channel specific data processing along with the services provided by standard UNIX devices. Following we briefly describe some of the major changes made to the MMT and ATM device drivers.

Data Structure: To incorporate the notion of I/O channels we extended the device driver data structure to include channel state information for each open connection. The channel state for a connection consists of a pointer to a buffer pool reserved for the channel, pointers to asynchronous connection handlers, and pointer to a data structure consisting of device dependent channel state information.

Entry Points: In order to maintain backward compatibility we use the standard UNIX interface for the drivers. The new functionalities are accessed using the extension parameters with the standard calls and several new ioctl calls. In the following we briefly explain some of the important features of the extended driver.

[6] This would have required hardware modifications and changes to device micro-codes and were beyond the scope our project.

open: The *open* entry point is used to open an I/O channel to the device. We use the extension parameter to specify the channel state information and channel handlers. As a response to the open call, the kernel opens a new channel with the specified characteristics. If sufficient resources are not available, the request to open a channel is refused. The following example explains the actions in more detail.

```
int rc;
int devfd;              /* Device file pointer */
struct CHANEXT {
   int chhandle;        /* Channel handle  */
   int *ch_rcv();
   int *ch_snd();
   int *ch_ctl();
   struct *ch_state;
} ext;                 /* Channel handlers */


    mapext.ch_rcv = rxhandler;       /* Receive handler */
    mapext.ch_snd = txhandler;       /* Transmit handler */
    mapext.ch_ctl = sxhandler;       /* Status handler */

    rc = open(devfd, devflag, &ext);  /* Registration */
```

In the pseudo code above, `rxhandler`, `txhandler`, and `sxhandler` are the asynchronous handlers for receive, transmit, and status handling, respectively. These handlers are specified by the application and are used for connection specific data processing. The structure ch_state is used to initialize the channel state variables. It is also used for specifying channel parameter and for admission control. For example, for ATM interface controller, we specify the channel characteristics in terms of the bandwidth requirement and traffic envelope (e.g. the peak arrival rate, burst size etc.). Depending on current state of the device, the request to open a new connection may or may not be complied with.

close: The *close* call is used to close a previously open I/O channel. It de-allocates the resources reserved for the channel and unloads the channel handlers.

read: The applications use the *read* entry point to move data from the device to application space. The application can use the receive handlers to perform channel specific processing on the data as a part of the read call.

write: The applications use the *write* interface of the device driver to move data from the application space to the device. The channel specific transmit handler is used to process the write request.

ioctl: We added several *ioctl* calls to facilitate data streaming. To enable streaming mode transfers between two devices the application first open I/O channels to both devices and set up the channel handler in such a way that data generated from the source is moved directly to the sink. We have added ioctl calls to initiate and terminate streaming mode transfer. We have also added ioctl calls for modifying the channel parameters and loading and unloading the channel handlers.

Interrupt Handler: We have extended the interrupt handlers for the MMT and ATM device to perform demultiplexing functions. For example, when the ATM interface controller interrupts the system upon receipt of a complete AAL5 packet, the interrupt handler looks at the packet header to determine the particular I/O channel it belongs to and then calls the receive handler associated with the connection. This feature is used for device-to-device autonomous transfers by appropriately programming the connection handler so that it can route data directly to the destination device. We also use this feature to implement connection specific protocol processing.

3.3.2 Connection Specific Protocol Processing

Connection handlers open up the opportunity for connection specific protocol processing. We exploit this feature to implement a variety of protocol stacks on top of ATM AAL5. When an application opens an ATM connection using the extended open interface provided by the ATM driver, it can choose the protocol stack it wants to use with that connection by specifying appropriate connection handlers. The I/O channel associated with the connection is instantiated with the appropriate handlers to implement the requested protocol functionality. When the network interface receives a packet, it uses the receive handler associated with the channel to process it. Similarly, when the application makes a network send request, the transmit handler associated with the channel is called to process the data. As a default we use a native ATM stack where protocol processing is limited to the protocol functionality provided by AAL5. By using the native ATM stack we eliminate much of the protocol processing overhead in the optimized system.

3.3.3 Application Structure

We modified VASU to exploit the benefits of the new I/O architecture. On the transmit side, VASU opens one control and one data channel to the MMT. The control channel is used to move control information, such as device failure, from the MMT to VASU. The data channel carries the

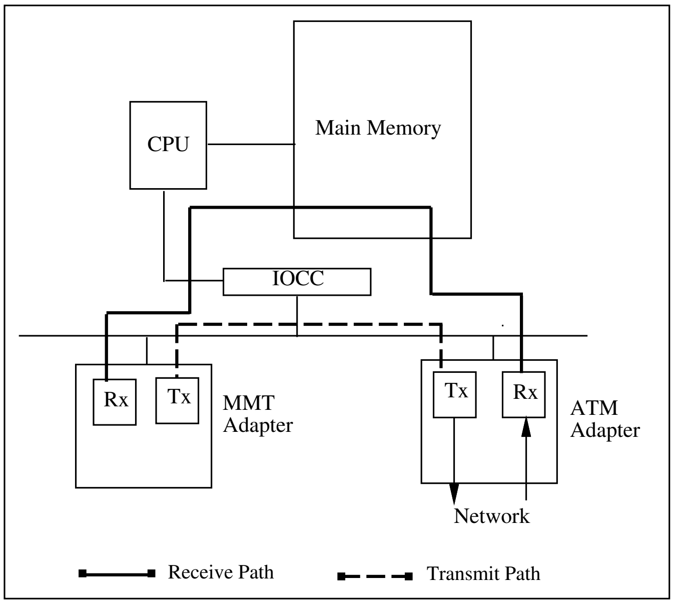

Figure 3.5: Transmit and receive data path in the optimized system.

video and audio data generated by MMT. VASU also opens a native ATM connection to the peer VASU at the destination. By appropriately programming the transmit handler associated with the MMT data channel, the data channels to MMT and ATM interface are spliced for the purpose of data streaming from the MMT to the ATM adapter.

Similarly, on the receive side, VASU opens two channels to MMT. One of them is used for control flow from MMT to VASU, the other is used for moving data to MMT. The data channel to MMT and the ATM network connection setup earlier by the peer VASU on the transmit side are spliced to form a direct data path from the ATM adapter to the MMT adapter. Data transfer starts when VASUs on the transmit and receive ends trigger data streaming through the ioctl interfaces to the MMT and ATM devices, respectively.

3.3.4 Data Flow

On the transmit side, when the MMT card has data to send, it interrupts the system. The interrupt is trapped by the interrupt handler which calls the appropriate transmit handler to process the packet. The send handler copies data directly into the buffer on the ATM adapter. It is the responsibility of the transmit handler of ATM to queue the datagram for transmission. In the transfer of data from MMT to the network interface the data path never crosses the kernel boundary and, hence, saves the cost of data copying and context switching. This mode of data transfer cannot still be called completely autonomous because the devices still interrupt the system.

However, the task of the interrupt handler is limited to calling the appropriate handler. Once the handler initiates the DMA, data transfer occurs autonomously between the device, transparent to the kernel and the application.

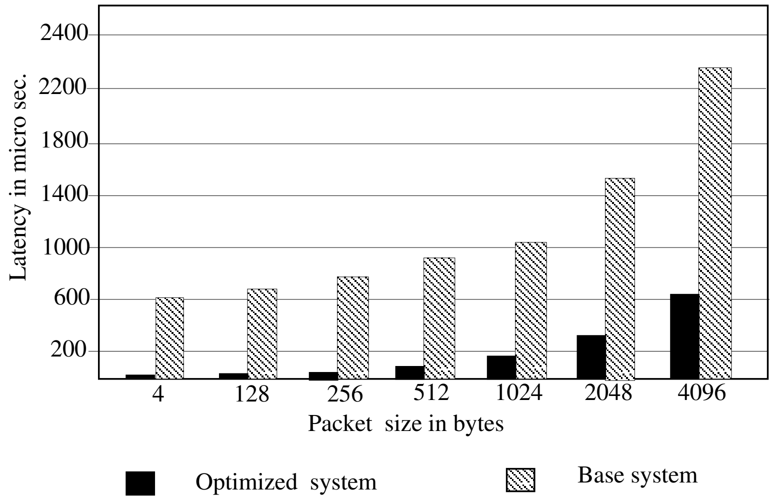

Figure 3.6: Comparison of transmit latencies in the base and optimized system (in microseconds).

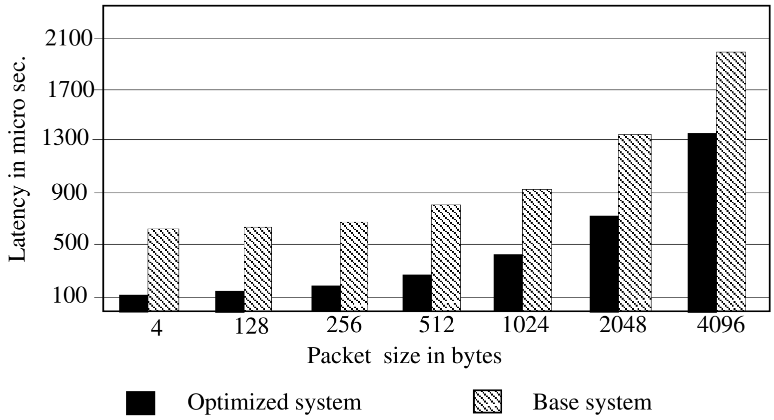

Figure 3.7: Comparison of receive latences in the base and optimize system (in microseconds).

On the receive side of the optimized system, whenever an (AAL5) packet is completely received, the network interface interrupts the system. In response, the interrupt handler calls the receive handler associated with the connection on which the packet is received. If the final destination of the data is the MMT adapter, the receive handler DMAs data in a mbuf chain in the main memory and invokes the appropriate receive handler on the MMT. It is now the responsibility of the MMT receive handler to copy the mbuf chain to the dual-port buffer on MMT adapter and submit the data for processing. Note that, the transmit and receive data paths are not symmetric. In the transmit side data generated by MMT is DMAed directly on to the buffer on the ATM adapter.

THROUGHPUT (in MB/sec)

Packet Size (in bytes)	TRANSMIT SIDE			RECEIVE SIDE		
	Optimized System	Base System	Improvement	Optimized System	Base System	Improvement
4	2.60	0.05	5200%	0.27	0.05	540%
128	30.56	1.60	1910%	6.78	1.70	398%
256	39.46	2.74	1440%	11.28	3.14	359%
512	45.71	4.65	983%	15.34	5.33	287%
1024	49.61	8.16	607%	19.73	9.37	210%
2048	51.84	10.90	475%	23.03	12.88	178%
4096	52.85	15.03	315%	24.65	17.42	141%

Figure 3.8: Transmit and receive throughputs.

However, on the receiving side, data received at the ATM interface is first DMAed into the main memory and then moved into the buffers on the MMT adapter. This asymmetry is due to the lack of adequate buffering on the MMT adapter. As mentioned earlier, MMT is a prototype device and has only 8KB of dual-port memory available for data buffering. It is divided into 4KB of transmit buffer and 4KB of receive buffer. The ATM card on the other hand has 2MB of on board dual-port buffer. Hence, when data is generated by MMT, it can be transferred directly on to the ATM adapter. However, while transferring data from ATM to MMT it has to buffered through the main memory since the single receive buffer on MMT is not always free. Figure 3.5 shows the transmit and receive data paths in the optimized system.

3.3.5 Limitations

There are a few limitations in the implementation of the autonomous device architecture described above. The devices still interrupt the system. To eliminate the interrupts and to make the data transfer completely autonomous, we need to modify the device micro code. Making these changes is beyond the scope of our project. We refer to this form of device-to-device transfers as "semi-autonomous" operations. In the next section, we will see that despite this shortcoming, the improvement in end-to-end throughput is more than three-fold. We believe that with fully autonomous data paths improvements in throughput will be even more dramatic.

3.4 Performance Results

In the optimized system, data paths never cross the domain boundaries. The in-kernel data transfer eliminates the system calls and consequently both i) the overhead due to data copying across domain boundaries and ii) the cost of context switching. The latencies in the optimized data path reflects the cost of moving data from the MMT adapter to the ATM interface and vice versa. On the transmit side, data is DMAed directly from the MMT adapter buffer to the buffer on the ATM adapter. On the receiving side, data received at the ATM network interface is first DMAed into a main memory buffer pool and then copied into buffer memory on the MMT card. Besides data path optimizations, we save some of the protocol processing overheads.

We profiled the transmit and receive data paths in the optimized system following the same procedures used in profiling the base system. Figures 3.6 and 3.7 compare the transmit and receive side latencies in the base and the optimized systems. Figure 3.8 summarizes all the measurements and shows the percentage improvement in throughput in the optimized system. The results speak for themselves. On the transmitting side, we observe throughput improvements ranging from 5200% for data segments of size 4bytes to 315% for data segments of size 4Kbytes. The dramatic improvement in throughput for smaller packet sizes reflects the fact that the fixed overhead associated with data transfer is negligible in the optimized system. Noting that the most significant part of the fixed overhead is contributed by the context switches, these results are quite expected. For larger data segments, the improvement in throughput can be attributed to the reduction in data copying cost. In the optimized system, data is copied only once, compared to three times in the base system. If we account for the savings due to the elimination of two context switches and two data copies, a three-fold improvement in throughput may appear a little anomalous. In fact, a three-fold improvement is expected due to the elimination of two data copies only. The reason behind this anomaly is that one of the two copies eliminated in the optimized data path is a main memory to main memory copy. A main memory to main memory copy is relatively less expensive than main memory to a I/O memory or a I/O memory to main memory copy. Hence, elimination of two copies out of three did not translate into a saving of two third of the overhead. We could not experiment with packet sizes larger than 4Kbytes since MMT's transmit buffer is 4Kbyte wide. Extrapolating from the measured data, we predict that the end-to-end throughput would saturate at around 55-60 Mb/s.

The improvements on the receiving side, although substantial, are not as good as those on the transmitting side. We observe throughput improvements ranging from 540% for 4byte packets to 141% for 4Kbyte packets, compared to 5200% and 315% for the corresponding packet sizes on the transmitting side. This difference can be attributed to the asymmetry in the send and receive data paths in the optimized system. We believe that with more buffering on MMT, it is possible to optimize the receive data path to yield throughputs comparable to those of the transmit data path.

3.5 Summary

In this chapter we discussed in detail the design, implementation, and evaluation of a video conferencing system developed using the principles of autonomous device operations. Our initial attempts to develop a desktop conferencing system supporting full-motion, high-resolution video and high-quality audio failed to meet our expectations despite adequate hardware support. After a through profiling of the system, we identified the operating system interfaces to the I/O devices and network interfaces as the bottlenecks. To alleviate the problem, we have implemented a prototype system where MMT and ATM adapters communicate with each other in a 'semi-autonomous' fashion with minimum application and kernel intervention. The experimental performance results presented above clearly demonstrates the effectiveness of the proposed architecture. We believe that the connection oriented autonomous device architecture is a powerful model with a potential to provide the sheer performance demanded by many multimedia applications.

Chapter 4

Traffic Shaping

In chapter 2 we proposed a connection oriented I/O architecture with the objectives to (1) improve network throughput as seen by the applications, and (2) establish an infrastructure for per-connection service guarantees. In chapter 3 we demonstrated the the performance impact of the proposed architecture on network throughput. The goal of this chapter, and that of chapters 5 and 6, is to make use of the proposed connection model to develop a service architecture that provides deterministic guarantees on end-to-end performance. In our model (see figure 4.1), autonomous devices are the sources and sinks of data that flows over the connection, called a virtual channel, joining them. Each virtual channel is associated with a *traffic envelope*, describing the characteristics (e.g. peak and average rates, burst lengths and periods) of the traffic it is carrying. Also associated with each channel is a *service envelope* specifying its service requirements, such as maximum tolerable delay, desired delay jitter etc. Our objective is to design a end-to-end service architecture that admits a connection only when it can guarantee to satisfy its service requirements.

In order to provide guaranteed services, resources, such as bandwidth, buffers, etc. have to be reserved a priori, so that the promised quality of service is not violated due to system overloading. The resource reservation algorithm at a multiplexing node takes into account the traffic envelopes and the service envelopes of all the connections passing though the node in order to determine how much resources to reserve for a particular connection, or whether to reject a connection when sufficient resources are not available. Besides traffic and service envelopes, resource reservation also depends on the multiplexing schemes used at the switching nodes. Hence, in order to design and develop an effective quality of service (QoS) architecture, we need to understand the interaction between the traffic characteristics and service requirements of different virtual channels and the multiplexing policy used at the switching nodes. In this chapter we focus on traffic characteristics. Multiplexing mechanisms are discussed in chapter 5. Chapter 6 is devoted to the evaluation of the service architecture.

The rest of the chapter is organized as follows. In section 4.1, we introduce the notion of traf-

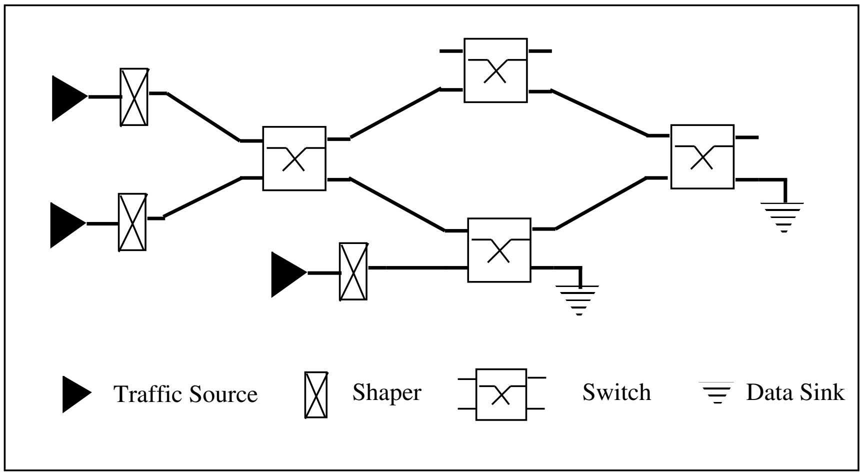

Figure 4.1: The network model.

fic shaping. We discuss simple shaping mechanisms in section 4.2. Composite shapers and the associated traffic envelope are discussed in section 4.3. We summarize in section 4.4.

4.1 Traffic Shapers

One of the most important components in a QoS architecture is the characteristics of the traffic source associated with each connection. The resource reservation algorithm at a multiplexing node needs to analyze the traffic envelopes associated with all the connections passing through the node in order to determine the service quality that can be offered to individual connections. Since the number of connections passing through a node may run into hundreds or even thousands, it is important that the traffic envelopes are succinct and simple. Unfortunately, traffic generated by most multimedia applications are very bursty and often difficult to model [28] and specify. To alleviate this problem, traffic generated by a source is passed through a traffic shaper which shapes the traffic to a form that is simple to specify and is easy to analyze. Besides shaping, a shaper (or regulator) also polices traffic so that a source may not violate the traffic envelope negotiated at the time of connection setup and degrade the service quality of other users. If the traffic generated by a source does not conform to the traffic envelope enforced by the shaper, the shaper can either drop the violating cells, tag them as lower priority traffic, or hold them in a reshaping buffer. In the rest of the discussion we assume that the shaper exercises the third option. In our model, traffic generated by a source (autonomous device) is first fed into a shaper buffer. From the shaper buffer the data is passed through a regulator and is released into the system/network in the form of fixed size cells satisfying the traffic envelope. The shape of the traffic envelope is determined by the shaping mechanism and the shaper parameters.

By introducing a shaper at the edge of the network we get a better control on the traffic entering the network. From the perspective of the network, the traffic generated by the shaper is really the traffic entering the network. In the rest of the chapter our objective is to study different shaping mechanisms and characterize the traffic envelope they enforce on a source. These traffic envelopes are used later to model the traffic arrival into the network.

4.2 Simple Shapers

Several shaping mechanisms enforcing different classes of traffic envelopes have been proposed in the literature. The most popular among them are *leaky bucket* [43], *jumping window* and *moving window* [38]. In the following we briefly describe their working principles and the traffic envelopes they enforce on a connection.

Leaky Bucket Shapers: A leaky bucket shaper consists of a token counter and a timer. The counter is incremented by one each t units time and can reach a maximum value b. A cell is admitted into the system/network if and only if the counter is positive. Each time a cell is admitted, the counter is decremented by one. The traffic generated by a leaky bucket regulator consists of a burst of up to b cells followed by a steady stream of cells with a minimum inter-cell time of t. The major attraction of leaky bucket is its simplicity. A leaky bucket regulator can be implemented with two counters, one to implement the token counter and the other to implement the timer.

Jumping Window Shapers: A jumping window regulator divides the time line into fixed size windows of length w and limits the number of cells accepted from a source within any window to a maximum number m. The traffic generated by a jumping window can have a worst-case burst length of $2m$. This happens when two bursts of size m each are released next to each other, the first one at the end of its window and the second one at the beginning of the next window. Like a leaky bucket, a jumping window regulator can also be implemented with two counters.

Moving Window Shapers: Similar to a jumping window, in a moving window, the number of arrivals in a time window w is limited to a maximum number m. The difference is that each cell is remembered for exactly one window width. That is, if we slide a window of size w on the time axis, the number of cells admitted within a window period can never exceed m irrespective of the position of the window. Hence, the worst-case burst size in a moving window regulator never exceeds m. Compared to a jumping window shaper, traffic generated by a moving window shaper is smoother. This smoothness, however, comes at the cost of added complexity in implementation. Since the departure time of each cell is remembered for a duration of one window period, the implementation complexity depends on m, the maximum number of cells released within a window period.

A good shaping mechanism should have the following properties:

- The traffic envelope it enforces on a source should be easy to describe.
- It should be simple to implement and easy to police.
- It should be able to capture a wide range of traffic characteristics.

Although the shaping mechanisms described above satisfy the first and second requirements, they do not quite meet the third one. Traffic envelopes defined by a single leaky bucket, jumping window, or moving window are often not adequate for precise characterization of the bursty and the variable rate traffic generated by multimedia applications. This mismatch between the original traffic form and that enforced by the shaper can have serious implications on system performance. Let us consider a real-life example to better understand the problem.

Example: Consider an MPEG [30] coded video stream. From uncompressed video data, an MPEG encoder produces a coded bit stream representing a sequence of encoded frames. There are three types of encoded frames: I (intracoded), P (predicted), and B (bidirectional). The sequence of encoded frames are specified by two parameters: M, the distance between I and P frames, and N, the distance between I frames. For example, when M is 2 and N is 5, the sequence of encoded frames is IBPBIBPB ...(see figure 4.2). The pattern IBPB repeats indefinitely. The interarrival time τ between two successive frames is fixed, and depends on the frame rate.

In general, an I frame is much larger than a P frame (in size), and a P frame is much larger than a B frame. Typically, the size of an I frame is larger than the size of a B frame by an order of magnitude. Let us assume that $|I|$, $|P|$, and $|B|$ be the sizes of I, P, and B frames, respectively [1]

Let us consider the problem of choosing a moving window shaper for this source. One good set of parameters is $w = \tau$ and $m = |I|$. Clearly, with this choice of parameters, there will be no delay in the shaper. However, this traffic envelope grossly over-estimates the traffic generated by the source. In fact, this is a peak rate approximation of the original source. An alternative set of parameters is $w = \tau$ and $m = (|I| + |P| + 2|B|)/4$. With this shaping envelope, the average rate of traffic generated by the original source and that estimated from the traffic envelope are the same.

[1] For simplicity we assume that sizes of all I frames are same and so are sizes of all P and B frames. In general, $|I|$, $|P|$, and $|B|$ are random variables. However, it is not unreasonable to assume that $|I|$ is larger than $|P|$ and $|P|$ is larger than $|B|$.

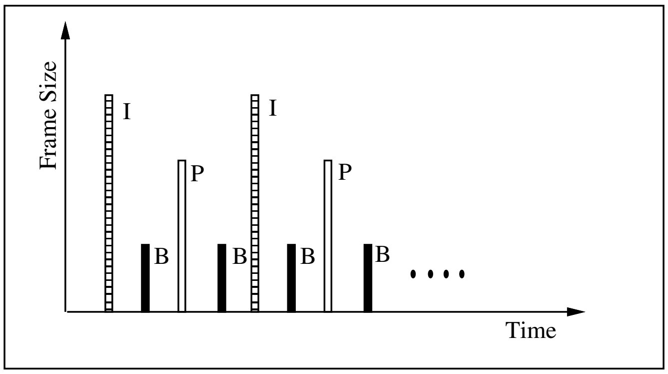

Figure 4.2: An example of an MPEG coded stream.

In other words, it is an average rate approximation of the source. It smooths the traffic generated by original source to an equivalent source with the same average rate by holding the frames in the shaper buffer. Here, an I frame may be delayed by $\tau\lfloor 4|I|/(|I| + |P| + 2|B|)\rfloor$ time units in the shaper buffer.

Clearly, none of the choices is satisfactory. With the peak-rate approximation of the source, there is no delay in shaper. However, this traffic envelope grossly over-estimates the resource requirement of the source, leading to loss of network utilization. With the average-rate approximation of the source, network resource utilization is high, but only at the expense of a higher shaper delay. Hence, it is important that the traffic envelope captures the characteristics of the original source as closely as possible. The traffic envelopes enforced by single leaky bucket, moving window, and jumping window are too simple for an accurate characterization of bursty sources. We can do better if we use multiple shapers to shape a traffic source.

4.3 Composite Shapers

Simplicity is the main attraction of the shapers described in the last section. They all are quite easy to implement, and define traffic envelopes that are easily analyzable. Unfortunately, as explained in the example above, they are often too simple to capture the true characteristics of real-life sources. All the shapers described in the previous section enforce a specific rate constraint on a source, typically a declared peak or average rate. However, most applications generate inherently bursty traffic. That is, the difference between the peak-rate and the mean-rate of traffic generation is quite high. Hence, enforcing an average rate results in higher delay in the shaper buffer, and a peak rate enforcement leads to over-allocation of system resources, and consequently lower utilization of the system. To alleviate this problem, or in other words, to enforce a traffic envelope that is close to the original shape of the traffic, and yet simple to specify and monitor, we can use multiple shapers

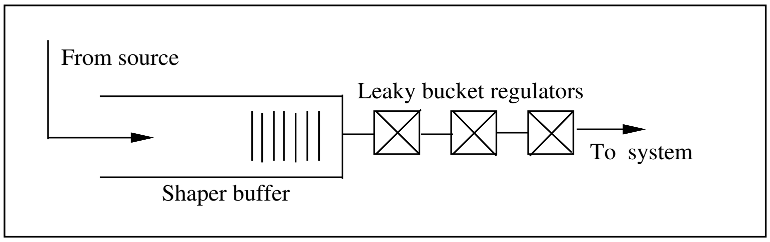

Figure 4.3: Composite leaky bucket.

to enforce multiple rate constraints. That is, we can choose multiple leaky buckets, moving windows and jumping windows enforcing different rate constraints over different time intervals. For example, two leaky buckets arranged in series can be used to enforce a short-term peak rate and a long-term average rate on a source. Composite shapers provide us with a much richer and larger set of traffic envelopes with marginal increase in complexity.

Example: To appreciate the effectiveness of composite shaper in better characterizing a bursty source let us consider the same example as above. A dual-moving-window shaper with parameters $(w_1 = 4 \times \tau, m_1 = |I| + |P| + 2|B|)$ and $(w_2 = \tau, m_2 = |I|)$ can capture the original characteristics of the source much more accurately than either one of moving windows alone. The first shaper enforces the longer term average rate while the second one controls the short term peak rate of the source. Now, the cells generated from the I frame would not be held up in the shaper buffer for $\tau \lfloor 4|I|/(|I| + |P| + 2|B|) \rfloor$ time units, as is the case when we use a single moving window with $w = \tau$ and $m = (|I| + |P| + 2|B|)/4$. Also, unlike the traffic envelope of the moving window with $w_2 = \tau$ and $m_2 = |I|$, the composite traffic envelope does not over-estimate the traffic generated by source. An even more precise characterization can be done with three moving window regulators with parameters $(w_1 = 4 \times \tau, m_1 = |I| + |P| + 2|B|)$, $(w_2 = 2 \times \tau, m_2 = |I| + |B|)$ and $(w_3 = \tau, m_3 = |I|)$. With a better characterization of a traffic source a more efficient utilization of the shared resources is possible.

Although developing techniques for choosing the right shaping mechanism and shaper parameters for a traffic source is an interesting and important subject of research, it is not the focus this dissertation. Our objective is to study the performance of the network subsystem, and for our purposes shapers are the traffic sources that feed the network. Hence, our interest is in characterizing the traffic generated by the shapers so that they can be used to model the traffic arrivals in the network. Also, since we are interested in the worst-case behavior, we concentrate on the worst-case bursty traffic generated by the shapers. In the following, we derive the exact forms of the worst-case traffic envelopes enforced by composite leaky bucket, moving window, and jumping window shapers.

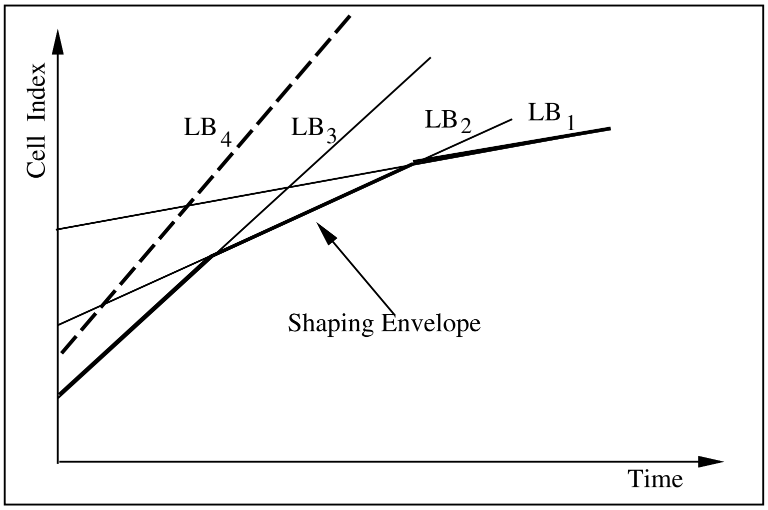

Figure 4.4: Shaping with multiple leaky buckets.

4.3.1 Composite Leaky Bucket

In a composite leaky bucket shaper, multiple simple leaky buckets are arranged in cascade (see figure 4.3). The data generated by the source is first buffered in the shaper buffer. Fixed length cells are dispatched into the system if and only if there is at least one token in each of the leaky buckets. A single leaky bucket generates the worst-case bursty traffic when the system starts with a full bucket of tokens, and dispatches a cell whenever there is a token.

This worst-case behavior of a leaky bucket is characterized by a traffic envelope that starts with a burst equal to the bucket size, followed by a straight line [2] of slope equal to the rate of token generation. The traffic envelope enforced by a composite leaky bucket is the intersection of the traffic envelopes of the constituent leaky buckets.

Example: In figure 4.4 a composite leaky bucket consisting of leaky buckets LB_1, LB_2, LB_3, and LB_4 is shown. The composite traffic envelope is marked by the dark line. The exact shape of the envelope depends on the number of components and the associated parameters. Inappropriate choice of shaper parameters may give rise to redundant components which may not have any role in defining the traffic envelope. For example, LB_4 is a redundant component in the composite shaper shown in figure 4.4. We call a set of leaky buckets an *essential set* if none of the buckets is redundant.

[2]Strictly speaking, the traffic envelope due to each leaky bucket is a burst followed by a stair case function. For ease of exposition we have approximated the stair case function by a straight line with the same slope. This simplification is only for the purpose of explanation. The results derived later takes into consideration the stair case function.

Let us consider n leaky buckets (b_i, t_i), $i = 1, 2, \ldots, n$. Without loss of generality we number the buckets in such a fashion so that $t_i \geq t_j$, for $i < j$. We will show that if these leaky buckets form an *essential set* then $b_i > b_j$ and $t_i > t_j$, for $i < j$.

Lemma 4.1 *If two leaky buckets (b_1, t_1) and (b_2, t_2) form an essential set, then $b_1 > b_2$ and $t_1 > t_2$.*

Proof: With the numbering scheme we have, we always have $t_1 \geq t_2$.

Case $t_1 = t_2$: If $b_1 < b_2$, then only bucket 1 is effective. If $b_1 = b_2$, the buckets are identical and one of them can be eliminated. If $b_1 > b_2$, then bucket 1 is redundant.

Case $t_1 > t_2$: If $b_1 < b_2$, then only bucket 1 is effective. If $b_1 = b_2$, then bucket 1 is effective. If $b_1 > b_2$, then the effect of both buckets can be observed. $\square$

Lemma 4.2 *If a set of leaky buckets (b_i, t_i), where $i = 1, 2, \ldots n$, form an essential set, then $b_1 > b_2 > \cdots > b_n$ and $t_1 > t_2 > \cdots > t_n$.*

Proof: By using lemma 4.1 repeatedly the result follows. $\square$

The following theorem characterizes the traffic envelope of a composite shaper consisting of an essential set of leaky buckets. To capture the worst case bursty behavior, we assume that all the buckets are full and cells are released to the network as long as there is at least one token in each of the buckets. Before presenting the theorem let us formally define a composite leaky bucket.

Definition 4.1 *An n-component composite leaky bucket is an essential set of n leaky buckets $(b_1, t_1), (b_2, t_2), \ldots, (b_n, t_n)$, where $b_i > b_j$ and $t_i > t_j$ for $i < j$. For the purpose of mathematical convenience we assume that the n-component shaper includes another pseudo leaky bucket (b_{n+1}, t_{n+1}), where $b_{n+1} = 0$ and $t_{n+1} = 0$.*

Theorem 4.1 *Consider an n-component composite leaky bucket. Let us define*

$$B_k = \begin{cases} \infty & k = 0, \\ \left\lfloor \dfrac{b_k t_k - b_{k+1} t_{k+1}}{t_k - t_{k+1}} \right\rfloor & k = 1, 2, \ldots, n, \\ 0 & k = n + 1. \end{cases}$$

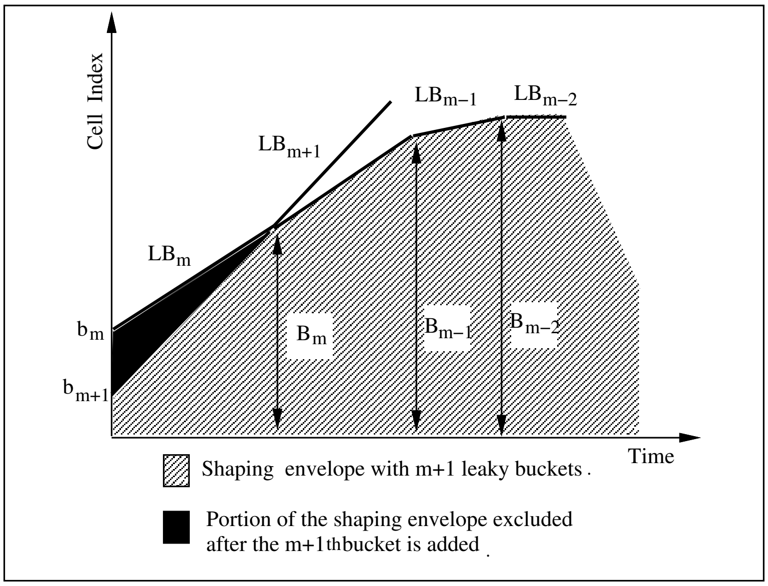

Figure 4.5: Traffic envelope after adding $(m+1)^{th}$ bucket.

Then the departure time of the i^{th} *cell from the composite shaper, denoted by* $a(i)$*, can be expressed as*

$$a(i) = \sum_{k=1}^{n+1} (i - b_k + 1)\ t_k\ [U(i - B_k) - U(i - B_{k-1})], \quad i = 0, 1, \ldots, \infty$$

where $U(x)$ *is the unit step function defined as*

$$U(x) = \begin{cases} 0 & x < 0, \\ 1 & x \geq 0. \end{cases}$$

Proof: We will prove this theorem by induction.

Base Case: For $n = 1$, we have $B_0 = \infty$, $B_1 = b_1$, and $B_2 = 0$. Therefore,

$$a(i) = (i - b_1 + 1)\ t_1\ U(i - b_1).$$

This allows a burst of size b_1 to depart at time 0 and a cell after every t_1 henceforth. Clearly, the traffic envelope captures the characteristics of the traffic generated by a leaky bucket with parameters b_1 and t_1. Hence the hypothesis holds in the base case.

Inductive Hypothesis: Assume that the theorem holds for all $n \leq m$. To prove that it holds for all n, we need to show that it holds for $n = m + 1$.

Figure 4.5 shows the traffic envelope before and after the addition of the $(m+1)^{th}$ bucket. Since the set of buckets constitute an essential set, $b_{i+1} < b_i$ for $i = 1, 2, \ldots m$. Therefore, the effect of $(m+1)^{th}$ bucket is observed from time 0 to the time when the traffic envelope due to bucket $m+1$ intersects the composite traffic envelope due to leaky buckets 1 through m (see figure 4.5). We observe that this cross-over point is really the point of intersection between the traffic envelopes of the m^{th} and the $(m+1)^{th}$ leaky buckets. Using simple geometry, we can find that they intersect at the departure instant of the $\lfloor (b_m t_m - b_{m+1} t_{m+1})/(t_m - t_{m+1}) \rfloor$th cell, which, by definition, is B_m.

In other words, the bucket $m+1$ excludes the segment marked by the solid shadow from the shaping envelope as shown in figure 4.5. The new segment can be expressed as

$$(i - b_{m+1} + 1)\ t_{m+1}\ [U(i - B_{m+1}) - U(i - B_m)]$$

The composite traffic envelope of buckets 1 through m is a piecewise linear function joining the points of intersections of the traffic envelopes of i^{th} and $(i+1)^{th}$ buckets, for $i = 1, 2, \ldots m-1$. By adding the new segment to that, we get the composite traffic envelope due to $m+1$ buckets.

$$\begin{aligned}
a(i) &= \sum_{k=1}^{m+1} (i - b_k + 1)\ t_k\ [U(i - B_k) - U(i - B_{k-1})] \\
&\quad + (i - b_{m+1} + 1)\ t_{m+1}\ [U(i - B_{m+1}) - U(i - B_m)] \\
&= \sum_{k=1}^{m+2} (i - b_k + 1)\ t_k\ [U(i - B_k) - U(i - B_{k-1})] \qquad (since\ \ b_{m+2} = 0)
\end{aligned}$$

This completes the proof. □

4.3.2 Composite Moving Window

A single moving window (w, m) generates the worst-case bursty traffic when a burst of m cells is released at the beginning of each window. Hence, the worst-case traffic envelope of a single moving window is characterized by a stair-case function with step height m and step width w. In a composite moving window shaper, multiple moving window regulators are arranged in cascade. The traffic envelope defined by a composite moving window is the intersection of the traffic envelopes of the constituent shapers. The following example explain it in more detail.

Example: Figure 4.6 shows the worst-case traffic envelope of a composite shaper consisting of moving windows $MW_1 = (w_1, m_1)$, $MW_2 = (w_2, m_2)$, $MW_3 = (w_3, m_3)$, where $w_1 = 3 \times w_2$, $w_2 = 4 \times w_3$ and $m_1 = 2 \times m_2$, $m_2 = 2 \times m_3$. In this particular example, traffic is shaped over three different time windows. The first moving window limits the number of cells dispatched in any time

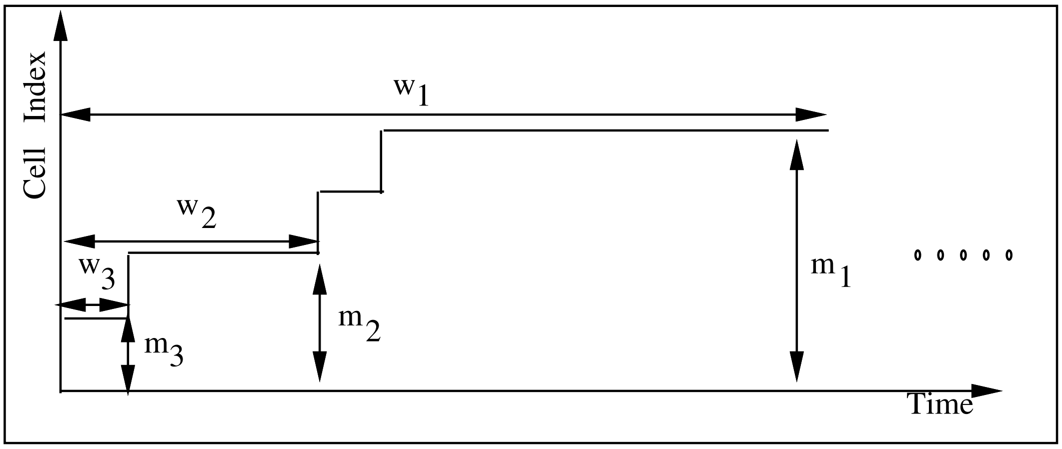

Figure 4.6: Traffic envelope of a composite moving window.

window of size w_1 to m_1. However, MW_1 does not impose any restriction on how these m_1 cells are dispatched. In the worst-case, they may be dispatched as a single burst of size m_1. Moving window MW_2 determines the burst size distribution within w_1. Window w_1 can be broken down into three windows of duration w_2 each. The maximum amount of traffic that can be dispatched during any time interval w_2 is limited to m_2 by MW_2. Hence, m_2 cells are dispatched in each of the first two w_2 intervals. Since $m_1 = 2 \times m_2$, no cell is dispatched in the third w_2 window to satisfy the constraint imposed by the first moving window. Similarly, the second moving window limits the maximum number of cells released in a window of length $w_2 = 4 \times w_3$ to $m_2 = 2 \times m_3$. Hence, m_3 cells are dispatched in each of the first two w_3 windows within a w_2 window. In the remaining two w_3 windows no cells are dispatched to satisfy the constraint imposed by MW_2.

The following theorem captures the shape of the worst-case traffic envelope of a composite moving window shaper. Before presenting the theorem let us first formally define a composite moving window shaper.

Definition 4.2 *An n-component composite moving window shaper consists of n simple moving windows* (w_k, m_k)*,* $k = 1, \ldots, n$*, where* $w_i \geq w_j$*,* $m_i \geq m_j$*, and* $m_i/w_i \leq m_j/w_j$*, for* $1 \leq i < j \leq n$*. For the sake of mathematical convenience we assume that an n-component composite shaper also includes another pseudo moving window* (m_0, w_0) *such that* $m_0/m_1 = 0$*. We also assume for simplicity of exposition that* m_{i+1} *divides* m_i*, and* w_{i+1} *divides* w_i*, for* $i = 1, 2, \ldots, n-1$*.*

Theorem 4.2 *Consider an n-component moving window shaper. If* $a(i)$ *is the departure time of the* i^{th} *cell from the shaper, then*

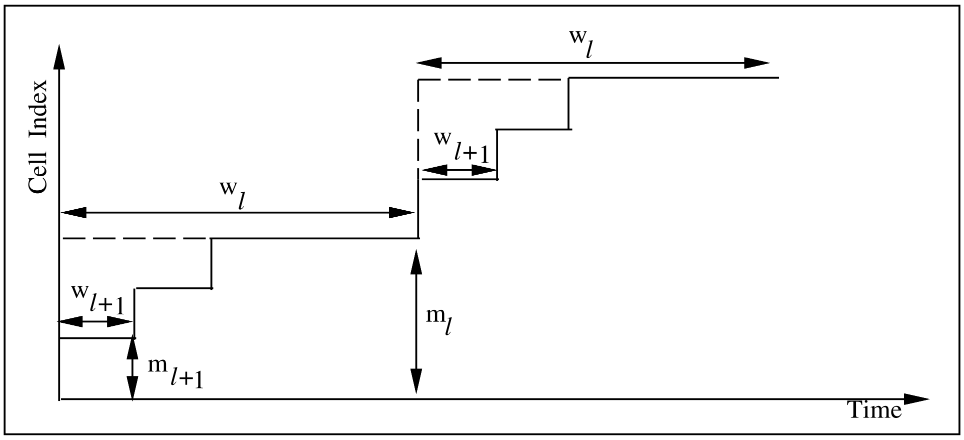

Figure 4.7: Traffic envelope after adding the $(l+1)^{th}$ moving window.

$$a(i) = \sum_{k=1}^{n} \left(\left\lfloor \frac{i}{m_k} \right\rfloor - \left\lfloor \frac{i}{m_{k-1}} \right\rfloor \frac{m_{k-1}}{m_k} \right) w_k, \qquad i = 0, 1, \ldots, \infty$$

Proof: We will prove this by induction.

Base Case: For $n = 1$, we have $a(i) = \left\lfloor \frac{i}{m_1} \right\rfloor w_1$. This means that a bursts of size m_1 appear at times kw_1, $k = 0, 1, \ldots, \infty$. Clearly, this represents the traffic envelope due to a single moving window with parameters (w_1, m_1). Hence, the premise holds in the base case.

Inductive Hypothesis: Assume that the premise holds for all $n \leq l$. To prove that it holds for all n, we need to show that it holds for $n = l + 1$.

Consider the effect of adding the $(l+1)^{th}$ moving window. In the worst-case, the burst always comes at the beginning of a window. Therefore, as shown in figure 4.7, bursts of size m_l cells appear at the beginning of each window of length w_l, for m_{l-1}/m_l windows. Now, from the hypothesis, the arrival time of the i^{th} cell is given by

$$a(i) = \sum_{k=1}^{l} \left(\left\lfloor \frac{i}{m_k} \right\rfloor - \left\lfloor \frac{i}{m_{k-1}} \right\rfloor \frac{m_{k-1}}{m_k} \right) w_k.$$

If a new shaper (w_{l+1}, m_{l+1}) is added, the burst appearing at the beginning of each w_l window will spread out into m_l/m_{l+1} bursts of size m_{l+1} each and separated by w_{l+1}, as shown in figure 4.7. Due this spreading out of the bursts, the arrival time of the i^{th} cell will be postponed by

$$\left(\left\lfloor \frac{i}{m_{l+1}} \right\rfloor - \left\lfloor \frac{i}{m_l} \right\rfloor \frac{m_l}{m_{l+1}} \right) w_{l+1}$$

Hence the arrival time of the i^{th} cell after the addition of the $(l+1)^{th}$ moving window is

$$\begin{aligned} a(i) &= \sum_{k=1}^{l} \left(\left\lfloor \frac{i}{m_k} \right\rfloor - \left\lfloor \frac{i}{m_{k-1}} \right\rfloor \frac{m_{k-1}}{m_k} \right) w_k \\ &\quad + \left(\left\lfloor \frac{i}{m_{l+1}} \right\rfloor - \left\lfloor \frac{i}{m_l} \right\rfloor \frac{m_l}{m_{l+1}} \right) w_{l+1} \\ &= \sum_{k=1}^{l+1} \left(\left\lfloor \frac{i}{m_k} \right\rfloor - \left\lfloor \frac{i}{m_{k-1}} \right\rfloor \frac{m_{k-1}}{m_k} \right) w_k \end{aligned}$$

This completes the proof. □

4.3.3 Composite Jumping Window

Although moving and jumping window shapers appear to be quite similar, their worst-case traffic envelopes can be quite different. In a single moving window shaper, two full size bursts are always separated by at least one window length. However, in a jumping window shaper, two bursts can appear next to each other, one at the end of a window and the other at the beginning of the next window. Similar cases are possible in composite jumping window shapers. Hence, unlike in moving window shapers, where the worst-case bursty behavior occurs when bursts appear at the earliest possible instants, in jumping windows the worst-case occurs when the first of the two consecutive bursts arrives at the end of its window, and the next arrives at the beginning of the next window.

Example: Figure 4.8 shows the traffic envelope of a composite shaper consisting of jumping windows $JW_1 = (w_1, m_1)$, $JW_2 = (w_2, m_2)$, $JW_3 = (w_3, m_3)$, where $w_1 = 3 \times w_2$, $w_2 = 4 \times w_3$ and $m_1 = 2 \times m_2$, $m_2 = 2 \times m_3$ in the interval $[0, 2w_1]$. If we had only one shaper (w_1, m_1), two bursts of size m_1 cells each could have appeared next to each other at w_1^- and and w_1. However, jumping window JW_2 mandates that the maximum size of the bursts can only be $m_2 = m_1/2$. Hence, each of the bursts of size m_1 has to be broken down into two bursts of size m_2 each. In the worst-case, they can appear at times $w_1 - w_2$, w_1^-,w_1 and $w_1 + w_2$. However, jumping window JW_3 restricts the size of each bursts to $m_3 = m_2/2$ cells only. Hence, each of the bursts of size m_2 has to be further broken down into bursts of size m_3. Consequently, in the worst-case, the first m_1 cells are released as four separate bursts of size m_3 cells each at times $w_1 - w_2 - w_3$, $w_1 - w_2$, $w_1 - w_3$, and w_1^-. The second batch of m_1 cells are released as four separate bursts of size m_3 each at times w_1, $w_1 + w_3$,$w_1 + w_2$, and $w_1 + w_2 + w_3$ (see figure 4.8). The subsequent batches of m_1 cells can be released starting at time $2w_1$ and following the constraints imposed by JW_is, for $i = 1, 2, 3$.

The following theorem defines the exact form of the worst-case traffic envelope defined by a composite jumping window. But before that, let us formally define a composite jumping window shaper.

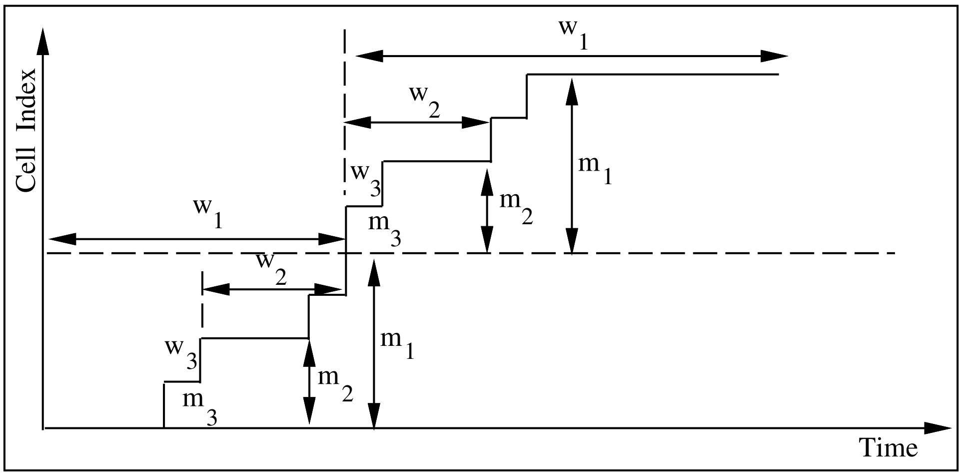

Figure 4.8: Traffic envelope of a composite jumping window shaper.

Definition 4.3 *An n-component composite jumping window shaper consists of n simple jumping windows* (w_k, m_k), $k = 1, \ldots, n$, *where* $w_i \geq w_j$, $m_i \geq m_j$, *and* $m_i/w_i \leq m_j/w_j$, *for* $1 \leq i < j \leq n$. *For the sake of mathematical convenience we assume that an n-component composite shaper also includes another pseudo jumping window* (m_0, w_0) *such that* $m_0/m_1 = 0$. *We also assume for simplicity of exposition that* m_{i+1} *divides* m_i, *and* w_{i+1} *divides* w_i, *for* $i = 1, 2, \ldots, n-1$.

Theorem 4.3 *The worst-case departure time* $a(i)$ *of the* i^{th} *cell,* $i = 0, 1, \ldots, \infty$, *from an n-component composite jumping window shaper is given by*

$$a(i) = \begin{cases} \displaystyle\sum_{k=1}^{n} \left\{ \left(\left\lfloor \frac{i}{m_k} \right\rfloor + 1 \right) - \left(\left\lfloor \frac{i}{m_{k-1}} \right\rfloor + 1 \right) \frac{m_{k-1}}{m_k} \right\} w_k, & 0 \leq i < m_1 \\ \\ \displaystyle\sum_{k=1}^{n} \left(\left\lfloor \frac{i}{m_k} \right\rfloor - \left\lfloor \frac{i}{m_{k-1}} \right\rfloor \frac{m_{k-1}}{m_k} \right) w_k, & m_1 \leq i < \infty. \end{cases}$$

Proof: The proof is similar to the proof of the last theorem. The only difference is that in the case of the jumping window, the first batch of bursts is released at the end of the first outer window. The rest of the batches are dispatched starting at the beginning of the subsequent outer windows. □

4.4 Summary

In this chapter, we discussed shaping traffic for the purpose of policing and characterization. We discussed simple shaping mechanisms such as leaky buckets, moving and jumping windows, and characterized the traffic envelopes defined by composition of multiple shapers. We will use these traffic envelopes to model input traffic to the network in the following chapters.

Shaping mechanisms have been proposed and analyzed by several authors (e.g. [38, 9]). However, most of these studies investigate different aspects of simple shapers such as a single leaky bucket or single jumping and moving window. The contribution of this chapter is in the characterization of the traffic envelopes defined by composite shapers. We believe that the characterization of traffic generated by composite shapers will be extremely useful in developing an end-to-end quality of service architecture, particularly with the recent standardization of multi-rate shaping of virtual connections by the ATM forum.

Chapter 5

Carry-Over Round Robin Scheduling

The heart of a quality of service architecture providing deterministic guarantees on performance is the multiplexing policy used at the switching nodes. Multiplexing is the allocation of link capacity to competing connections. The manner in which multiplexing is performed has a profound effect on the end-to-end performance of the system. Since each connection might have different traffic characteristics and service requirements, it is important that the multiplexing discipline treats them differently, in accordance with their negotiated quality of service. However, this flexibility should not compromise the integrity of the scheme, that is, a few connections should not be able to degrade service to other connections to the extent that the performance guarantees are violated. Also, the scheme should be analyzable since performance guarantees are to be given. Finally, it should be simple enough for implementation in high-speed switches. In this chapter we propose a multiplexing mechanism that attempts to achieve these daunting goals.

The rest of the chapter is organized as follows. In section 5.1 we review the current state of art. In section 5.2 we present a detailed description of the scheduling algorithm. An outline for a hardware implementation of the multiplexing mechanism is discussed in section 5.3. Some of the basic properties of the multiplexing discipline are analyzed in section 5.4. We summarize in section 5.5

5.1 Current State of Art

In the last several years a number of multiplexing disciplines have been proposed [5]. Based on the performance guarantees they provide, these schemes can be broadly categorized into two classes: ones that provide guarantees on maximum delay at the switching nodes and ones that guarantees a minimum throughput. In the following we briefly explain working principles of the representative schemes from each class and examine their merits and shortcomings.

5.1.1 Delay Guarantee

The multiplexing disciplines providing delay guarantees [26, 22] typically use priority based scheduling to bound the worst case delay encountered by a cell belonging to a connection at a particular switch. Depending on the nature of priority assignment, they can be further sub-divided into static priority schemes and dynamic priority schemes. In a static priority scheme [26], each connection is statically assigned a priority at the time of connection set up. When a cell from a certain connection arrives at the multiplexing node, it is stamped with the priority label associated with its connection, and is added to a common queue. The cells are served according to their priority order. There are other alternative approaches to implement a static priority scheduler. In a dynamic priority scheduler, the priority assigned to cells belonging to a particular connection can be potentially different, depending on the state of the server and that of the connections. For example, an Earliest Deadline First scheduler [22] uses a real or a virtual deadline as the priority label for a cell. Here again, cells are put in a common queue and served in the priority order. Knowing the exact arrival patterns of cells from different connections it is possible to bound the worst-case delay suffered by cells from a particular connection in both static and dynamic priority scheduling. The end-to-end queueing delay suffered by a cell passing through multiple multiplexing nodes, each employing deadline or priority scheduling, is the sum of the worst-case delays encountered at each node. One of the serious problems with the schemes described above is that they require traffic reshaping at each node. Priority scheduling completely destroys the original shape of the traffic envelope. Since these schemes require that the exact form of the traffic envelope be known at each node in order to guarantee worst-case delay bounds, traffic has to be reshaped into its original form as it exits a multiplexing node.

5.1.2 Throughput Guarantee

The multiplexing disciplines providing throughput guarantees [7, 23, 35, 48, 40] use weighted fair queueing and frame based scheduling to guarantee a minimum rate of service at each node. Knowing the traffic envelope, this rate guarantee can be translated into guarantees on other performance metrics, such as delay, delay jitter, worst-case backlog, etc. Unlike the disciplines providing delay guarantees, in rate based schemes, worst case end-to-end queueing delay is equal the delay suffered at the bottleneck node only, not the sum of the worst-case delays at each intermediate node. Rate-based schemes are also more fair in terms of distributing excess bandwidth. In a delay-based scheme employing priority scheduling, excess bandwidth is consumed by the connections with the highest priorities, whereas in rate-based schemes it can be distributed more evenly and predictably.

Based on implementation strategies, rate-based schemes can be further classified into two categories – 1) priority queue implementation, and 2) frame-based implementation.

The most popular examples of priority queue implementations are virtual clock [48], packet-by-packet generalized processor sharing (PGPS) [13, 35], self clocked fair queueing (SFQ) [25], etc. In virtual clock, every connection has a clock associated with it that ticks at a potentially different rate. When a cell from a certain connection arrives at the system, it is stamped according to an algorithm that is independent of the arrivals from other connections and dependent only on the history of arrivals in the connection concerned, and the rate of service allocated to the connection. The stamped cells enter a queue common to all connections, and are served in the order of stamped value. Both PGPS and SFQ are similar to virtual clock in the sense that they all stamp the cells at their arrival, put all cells in a common queue, and serve them in the order of stamped value. However, they differ in the stamping algorithms they use. While these schemes are extremely flexible in terms of allocating bandwidth in very fine granularity and fair distribution of bandwidth among active connections, they are costly in terms of implementation. Maintaining a priority queue in the switches is expensive. In some cases [35], the overhead of the stamping algorithm also can be quite high.

Frame-based mechanisms are much simpler to implement. The most popular frame-based schemes are Stop-and-Go (SG) [23, 24] and Hierarchical-Round-Robin (HRR) [7]. Both SG and HRR use a multi-level framing strategy. For simplicity, we just describe one-level framing. In a framing strategy, the time axis is divided into periods of some constant length, called a frame. Bandwidth is allocated to each connection as a certain fraction of frame time. In SG, at each multiplexing node, the arriving frames of each incoming link is mapped onto the departing frames on the outgoing links. All the cells from one arriving frame of an incoming link and going to a certain outgoing link are put into the corresponding departing frame on the outgoing link. In some sense, SG is emulates circuit switching on a packet switched network. One-level HRR is equivalent to a non-work-conserving round robin service discipline. Each connection is assigned a fraction of the total available bandwidth and receives that bandwidth in each frame, if it has sufficient cells available for service. The server ensures that no connection gets more bandwidth than what is allocated to it, even if it has spare capacity and the connection is backlogged. Both SG and HRR are non-work-conserving service disciplines and, hence, fail to exploit the multiplexing gains of ATM. Another important drawback of SG and HRR, and all framing strategies for that matter, is that they couple the service delay with bandwidth allocation granularity. The delay encountered by a cell in a SG and HRR is bounded by frame size multiplied by a constant factor (in SG the constant is in between 2 and 3, in HRR it is 2). Hence, the smaller the frame size is the lower is the delay. However, granularity of bandwidth allocation is inversely proportional to the frame size, resulting in an undesirable coupling between delay and bandwidth allocation granularity.

In the following, we present a multiplexing mechanism designed to integrate the flexibility and fairness of the fair queueing strategies with the simplicity of frame-based mechanisms. The starting point of our algorithm, which we call carry-over round robin (CORR), is a simple variation of round robin scheduling. Like round robin, CORR divides the time-line into allocation cycles, and each connection is allocated a fraction of the available bandwidth in each cycle. However, unlike slotted implementations of round robin schemes where bandwidth is allocated as a multiple of a fixed quantum, in our scheme bandwidth allocation granularity can be arbitrarily small. Also, unlike the framing strategies like SG and HRR, ours is a work-conserving discipline, and hence unused bandwidth is not wasted but is fairly shared among the active connections. The following is an algorithmic description of CORR scheduling discipline.

5.2 Scheduling Algorithm

Like simple round robin scheduling, CORR divides the time line into allocation cycles. The maximum length of an allocation cycle is T. Let us assume that the cell transmission time is the basic unit of time. Hence, the maximum number of cells (or slots) transmitted during one cycle is T. At the time of admission, each connection C_i is allocated a rate R_i expressed in cells per cycle. Unlike simple round robin schemes, where R_is have to be integers, CORR allows R_is to be real. Since R_is can take real values, the granularity of bandwidth allocation can be arbitrarily small, irrespective of the length of the allocation cycle. The goal of the scheduling algorithm is to allocate each connection C_i close to R_i slots in each cycle and exactly R_i slots per cycle over a longer time frame. It also distributes the excess bandwidth among the active connections C_is in the proportion of their respective R_is.

The CORR scheduler (see figure 5.1) consists of three asynchronous events —- *Initialize*, *Enqueue*, and *Dispatch*. The event *Initialize* is invoked when a new connection is admitted. If a connection is admissible [1], it simply adds the connection to the connection-list $\{C\}$. The connection-list is ordered in the decreasing order of $R_i - \lfloor R_i \rfloor$, that is, the fractional part of R_i. The event *Enqueue* is activated at the arrival of a packet. It puts the packet in the appropriate connection queue and updates the cell count of the connection. The most important event in the scheduler is *Dispatch*. The event *Dispatch* is invoked at the beginning of a busy period. Before explaining the task performed by *Dispatch*, let us introduce the variables and constants used in the algorithm and the basic intuition behind it.

The scheduler maintains separate queues for each connection. For each connection C_i, n_i keeps the count of the waiting cells, and r_i holds the number of slots currently credited to it. Note that r_is can be real as well as negative fractions. A negative value of r_i signifies that the connection has been allocated more slots than it deserves. A positive value of r_i reflects the current legitimate requirements of the connection. In order to allocate slots to meet the requirements of the connection

[1] We discuss admission control later.

```
Constants
    T: Cycle length.
    R_i: Slots allocated to C_i.
Variables
    {C}: Set of all connections.
    t: Slots left in current cycle.
    n_i: Number of cells in C_i.
    r_i: Current slot allocation of C_i.
Events
    Initialize(C_i) /* Invoked at connection setup time. */
        add C_i to {C};    /* {C} is ordered in decreasing order of R_i − ⌊R_i⌋. */
        n_i ← 0;   r_i ← 0;

    Enqueue() /* Invoked at cell arrival time. */
        n_i = n_i + 1
        add cell to connection queue;

    Dispatch() /* Invoked at the beginning of a busy period. */
        ∀C_i:: r_i ← 0;
        while not end-of-busy-period do
            t ← T;
            1. Major Cycle:
                    for all C_i ∈ {C} do    /* From head to tail. */
                            r_i ← min(n_i, r_i + R_i);   x_i ← min(t, ⌊r_i⌋);
                            t ← t − x_i;   r_i ← r_i − x_i;   n_i ← n_i − x_i;
                            dispatch x_i cells from connection queue C_i;
                    end for
            2. Minor Cycle:
                    for all C_i ∈ {C} do    /* From head to tail. */
                            x_i ← min(t, ⌈r_i⌉);
                            t ← t − x_i;   r_i ← r_i − x_i;   n_i ← n_i − x_i;
                            dispatch x_i cells from connection queue C_i;
                    end for
        end while
```

Figure 5.1: Carry-Over Round Robin Scheduling.

as closely as possible, CORR divides each allocation cycle into two sub-cycles —- a *major cycle* and a *minor cycle*. In the major cycle, integral requirement of each connection is satisfied first. Slots left over from major cycle are allocated in minor cycle to connections with still unfulfilled fractional requirements. Obviously, a fraction of a slot can not be allocated. Hence, eligible connections are allocated a full slot each in the minor cycle whenever slots are available. However, all the connections with fractional requirements may not be allocated a slot in the minor cycle. The connections that get a slot in the minor cycle over satisfy their requirements and carry a debit to the next cycle. The eligible connections that do not get a slot in the minor cycle carry a credit to the next cycle. The allocations for the next cycle are adjusted to reflect this debit and credit carried over from the last cycle. Following is a detailed description of the steps taken in the *Dispatch* event.

At the beginning of a busy period, all r_is are set to 0 and a new cycle is initiated. The cycles continue until the end of the busy period. At the beginning of each cycle, the current number of unallocated slots t is initialized to T, and the major cycle is initiated. In the major cycle, the *dispatcher* cycles through connection-list and, for each connection C_i, updates r_i to $r_i + R_i$. If the number of cells queued in the connection queue, n_i, is less than the updated value of r_i, r_i is set to n_i. This is to make sure that a connection cannot accumulate credits. The minimum of t and $\lfloor r_i \rfloor$ cells are dispatched from the connection queue of C_i. The variables are appropriately adjusted after dispatching the cells. A minor cycle starts with the slots left over from preceding major cycle. Again, the *dispatcher* walks through the connection-list. As long as there are slots left, a connection is deemed eligible for dispatching iff 1) it has queued packets, and 2) its r_i is greater than zero. If there is no eligible connection or if t reaches zero, the cycle ends. Note that the length of the major and minor cycles may be different in different allocation cycles.

Example: Let us consider a CORR scheduler with cycle length $T = 4$ and serving three connections C_1, C_2, and C_3 with $R_1 = 2$, $R_2 = 1.5$, and $R_3 = 0.5$, respectively. In an ideal system where fractional slots can be allocated, slots can be allocated to the connections in a fashion shown in figure 5.2, resulting in full utilization of the system. CORR also achieves full utilization, but with a different allocation of slots.

For ease of exposition, let us assume that all three connections are backlogged starting from the beginning of the busy period. In the major cycle of the first cycle, CORR allocates C_1, C_2, and C_3, $\lfloor R_1 \rfloor = 2$, $\lfloor R_2 \rfloor = 1$, and $\lfloor R_3 \rfloor = 0$ slots, respectively. Hence, at the beginning of the first minor cycle, $t = 1$, $r_1 = 0.0$, $r_2 = 0.5$, and $r_3 = 0.5$. The only slot left over for the minor cycle goes to C_2. Consequently, at the end of the first cycle, $r_1 = 0.0$, $r_2 = -0.5$, and $r_3 = 0.5$, and the adjusted requirements for the second cycle are

$$r_1 = r_1 + R_1 = 0.0 + 2.0 = 2.0$$

$$r_2 = r_2 + R_2 = -0.5 + 1.5 = 1.0$$

$$r_3 = r_3 + R_3 = 0.5 + 0.5 = 1.0$$

Since all the r_is are integral, they are all satisfied in the major cycle.

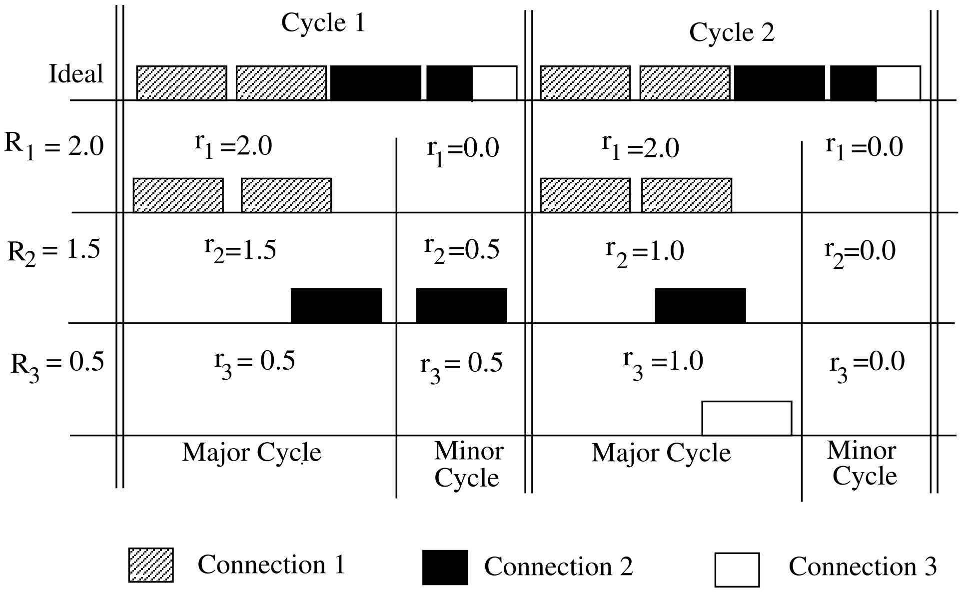

Figure 5.2: An Example Allocation.

5.2.1 Discussion

The main attraction of CORR is its simplicity. In terms of complexity, CORR is comparable to round robin and frame based mechanisms. However, CORR does not suffer from the shortcomings of round robin and frame based schedulers. By allowing the number of slots allocated to a connection in an allocation cycle to be a real number instead of an integer, we break the coupling between the service delay and bandwidth allocation granularity. Also, unlike frame based mechanisms, such as SG and HRR, CORR is a work conserving discipline capable of exploiting the multiplexing gains of packet switching. In the following, we briefly outline a hardware implementation of CORR scheduling. Later in the chapter we discuss some of its basic properties.

5.3 Implementation in a Switch

In this section, we present a design for implementation of CORR scheduling in an output buffered switch. More specifically, we outline the design of a buffer manager (figure 5.3), the key component used to implement the scheduling mechanism. The buffer manager consists of a cell pool, a table-look-up memory, an idle-address-FIFO, and a processor. The cells are stored in the cell pool. The table-look-up-memory, referred to as *connection table*, stores necessary control information. The idle-address-FIFO contains addresses of the current empty cell locations in the cell pool.

The buffer manager maintains a virtual queue for each connection by linking the cells through

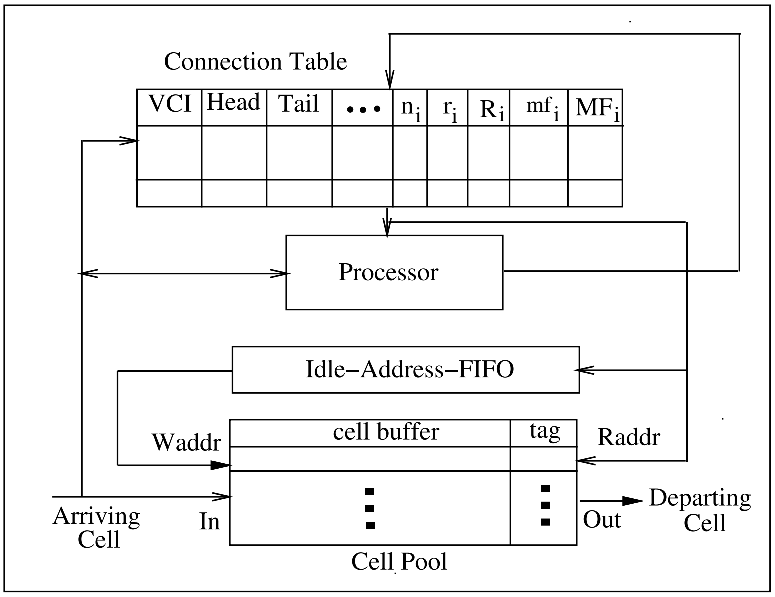

Figure 5.3: Architecture of the buffer manager.

pointers. The cells stored in the cell buffer are tagged with the pointer to the next cell from the same connection, if any. When a cell arrives, it is stored in the cell pool at the address given by the idle-address-FIFO. While the cell is being written into the cell pool, its connection identifier is extracted by the processor and the corresponding entry in the connection table is accessed to retrieve connection specific state information. The connection state is updated to reflect the new arrival, and the pointers are suitably adjusted to add the arriving cell at the end of the connection queue.

The connection table stores the state information of all connections. The state of a connection consists of Head and Tail pointers to the corresponding connection queue and a few connection specific constants and variables. The constants and variables important to the scheduling algorithm are R_i, r_i, and n_i. A cell from a connection is considered eligible for dispatching in a major cycle if n_i is greater than zero and r_i is greater than one. Similarly, a connection is considered eligible for dispatching in a minor cycle if n_i is greater than zero and r_i is greater than zero. We keep two one-bit flags, *MF* and *mf*, to indicate whether a connection is eligible for dispatching in the major and minor cycles, respectively. These flags are set/reset by local logic to save processor cycles. These flags are used for fast indexing to the connection states of the eligible connections during the major and minor cycles.

Once an eligible connection is selected, the processor extracts the buffer address of the cell at the head of the connection queue from the connection and dispatch it on the output link. The buffer address of the next cell in the queue, if any, is written into the Head field of the corresponding entry in the connection table. The processor also makes necessary changes to the state information stored in the connection table.

5.4 Basic Properties

In this section, we discuss some of the basic properties of the scheduling algorithm. Lemma 5.1 defines an upper bound on the aggregate requirements of all streams inherited from the last cycle. This result is used in lemma 5.2 to determine the upper and lower bounds on the individual requirements carried over from the last cycle by each connection.

Definition 5.1 *A connection is said to be in the busy period if the connection queue is non-empty. The system is said to be in the busy period if at least one of the connections is in its busy period.*

Note that a particular connection can switch between busy and idle periods even when the system is in the same busy period. The following theorem determines the departure time of a specific cell belonging to a particular connection.

Lemma 5.1 *If $\sum_{\forall C_i \in \{C\}} R_i \leq T$ then at the beginning of each cycle $\sum_{\forall C_i \in \{C\}} r_i \leq 0$.*

Proof: We prove this by induction. We first show that it holds at the beginning of a busy period. Then we show that if it holds in the k^{th} cycle, it also holds in the $(k+1)^{th}$ cycle.

Base Case: From the allocation algorithm, we observe that $r_i = 0$, for each connection C_i at the beginning of a busy period. Hence,

$$\sum_{\forall C_i \in \{C\}} r_i = 0$$

Thus, the assertion holds in the base case.

Inductive Hypothesis: Assume that the premise holds in the k^{th} cycle. We use superscripts for cycles in the following derivation.

$$\sum_{\forall C_i \in \{C\}} r_i^{k+1} = \sum_{\forall C_i \in \{C\}} r_i^k + \sum_{\forall C_i \in \{C\}} R_i - T \leq 0 + T - T \leq 0.$$

This completes the proof. $\square$

Remark: Henceforth we assume that the admission control mechanism makes sure that $\sum_{\forall C_i \in \{C\}} R_i \leq T$ at all nodes. This simple admission control test is one of the attractions of the CORR scheduling.

Lemma 5.2 *If $\sum_{\forall C_i \in \{C\}} R_i \leq T$ then at the beginning of each cycle*

$$-1 < -\delta_i \leq r_i \leq \delta_i < 1,$$

$$where \quad \delta_i = max_k\{kR_i - \lfloor kR_i \rfloor\}, k = 1, 2, \ldots$$

Proof: To derive the lower bound on r_i, observe that in each cycle no more than $\lceil r_i \rceil$ slots are allocated to connection C_i. Also note that r_i is incremented in steps of R_i. Hence, the lowest value r_i can have is

$$\begin{aligned} -\delta_i &= max_k\{kR_i - \lceil kR_i \rceil\}, k = 1, 2, \ldots \\ &= -max_k\{kR_i - \lfloor kR_i \rfloor\}, k = 1, 2, \ldots \end{aligned}$$

Derivation of the upper bound is a little more complex. Let us assume that there are n connections C_k, $k = 1, 2, \ldots, n$. Without loss generality we renumber them such that $R_i \geq R_j$, when $i < j$. For the sake of simplicity, let us also assume that all the R_i's are fractional. We show later that this assumption is not restrictive. To prove the upper bound, we first prove that r_i never exceeds 1 for all connections C_i. Now, since R_n is the lowest of all R_i's, C_n is the last connection in the connection list. Consequently, C_n is the last connection considered for a possible cell dispatch in both major and minor cycles. Hence, if we can prove that r_n never exceeds 1, then this is true for all other r_is. We will prove this by contradiction.

Let us assume that C_n enters a busy period in allocation cycle 1. Observe that C_n experiences the worst-case allocation when all other connections also enter their busy periods in the same cycle. Let us assume that $r_n > 1$. This would happen in the allocation cycle $\lceil 1/R_n \rceil$. Since $r_n > 1$, C_n is considered for a possible dispatch in the major cycle. Now, C_n is not scheduled during the major cycle of the allocation cycle $\lceil 1/R_n \rceil$ if and only if the following is true at the beginning of the allocation cycle:

$$\sum_{i=1}^{n-1} \lfloor r_i + R_i \rfloor \geq T.$$

From lemma 5.1, we know that $\sum_{i=1}^{n} r_i \leq 0$ at the beginning of each cycle. Since $r_n > 0$, $\sum_{i=1}^{n-1} r_i < 0$ at the beginning of the allocation cycle $\lceil 1/R_n \rceil$. However,

$$\sum_{i=1}^{n-1} \lfloor r_i + R_i \rfloor \leq \sum_{i=1}^{n-1} (r_i + R_i) < 0 + \sum_{i=1}^{n-1} R_i < T.$$

This is in contradiction with the assumption that C_n could not be scheduled in the major cycle. Hence, r_n cannot exceed 1. Since r_n is incremented in steps of R_n, the maximum value of r_n at the beginning of a cycle is the largest fractional part of kR_n, for any integer k. The same is true for other r_i's. Hence, the bound follows.

We have proved the bounds under the assumption that all R_i's are fractional. If we relax this assumption, the result still holds. This is due to the fact that the integral part of R_i is guaranteed to be allocated in each allocation cycle. Hence, even when R_i's are not all fractional, we can reduce the problem to an equivalent one with fractional R_i's using the transformation:

$$\hat{R}_i = R_i - \lfloor R_i \rfloor \quad and \quad \hat{T} = T - \sum_{i=1}^{n} \lfloor R_i \rfloor.$$

This completes the proof. □

The following theorem characterizes the departure function associated with a connection. For the purpose of simplicity, we have dropped the subscript identifying a connection in the following description.

Theorem 5.1 *Assume that a connection enters a busy period at time* 0*. Let $d(i)$ be the latest time by which the i^{th} cell, starting from the beginning of the current busy period, departs the system. Then $d(i)$ can be expressed as*

$$d(i) = \left\lceil \frac{i+1+\delta}{R} \right\rceil T, \quad i = 0, 1, \ldots, \infty,$$

where R is the rate allocated to the connection, T is the maximum length of the allocation cycle, and $\delta = max_k\{kR - \lfloor kR \rfloor\}, \quad k = 1, 2, \ldots$.

Proof: Since a cell may leave the system any time during an allocation cycle, we capture the worst-case situation by assuming that all the cells served during an allocation cycle leave the system at the end of the cycle. Now, when a connection enters a busy period, the lowest value of r is $-\delta$. If cell i departs at the end of the L^{th} cycle from the beginning of the connection busy period, the number of slots allocated by the scheduler is $L \times R - \delta$, and the number of slots consumed is $i+1$ (assuming packet number starts from 0). In the worst-case,

$$1 > L \times R - \delta - (i+1) \geq 0.$$

This implies that

$$\frac{i+1+\delta+1}{R} > L \geq \frac{i+1+\delta}{R}.$$

From the above inequality and noting that L is an integer and $d(i) = L \times T$, we get

$$d(i) = \left\lceil \frac{i+1+\delta}{R} \right\rceil T.$$

□

5.5 Summary

In this chapter, we have presented a cell scheduling discipline designed to provide deterministic performance guarantees on a per-connection basis. We have presented a detailed description of the algorithm and discussed some of its basic properties. These results are used in the next chapter to derive delay bounds and to analyze fairness properties of the algorithm.

It is clear form the discussion above that CORR is a simple extension of round robin discipline. It can be implemented using a two phase round robin scheduler and is much simpler compared to the priority queueing mechanisms. In terms of complexity, CORR is comparable to frame based mechanisms, such as SG and HRR. However, CORR does not suffer form the typical shortcomings of frame based scheduling. It is a work conserving discipline and hence can exploit the multiplexing gains of ATM. Also, unlike most frame based schemes, CORR does not suffer from the undesirable coupling between delay and bandwidth allocation granularity. In the next chapter, we show that albeit its simplicity, CORR is quite competitive with other more complex schemes in terms of performance.

Chapter 6

Quality of Service Envelope

In the last two chapters, we have discussed two basic components of a quality of service architecture, namely shaping and scheduling of traffic. The objective of this chapter is to analyze the end-to-end performance of the system and to enumerate a quality of service envelope. More precisely, our goal is to find the end-to-end delay in the worst case when CORR scheduling is used in conjunction with a specific shaping mechanism. Other end-to-end measures of performance such as delay jitter and local measures such as buffer sizes at each node can also be found from the results derived in this chapter. In order to find the end-to-end delay, we first derive delay bounds for a single-node system. We then show that the end-to-end delay for a multi-node system can be reduced to delay encountered in an equivalent single-node system. Hence, the delay bounds derived for a single-node system can be directly used to find the end-to-end delay. We also present a comprehensive fairness analysis of the CORR scheduling discipline.

The rest of the chapter is organized as follows. In section 6.1 we analyze delay characteristics of CORR scheduling. Section 6.2 is devoted to fairness properties of CORR. We summarize the contributions in section 6.3

6.1 Delay Analysis

In this section, we derive the worst-case delay bounds of a connection spanning single or multiple nodes, each employing CORR scheduling to multiplex traffic from different connections. We assume that each connection has an associated shaping envelope that describes the characteristics of the traffic it is carrying, and a minimum guaranteed rate of service at each multiplexing node. Our goal is to determine the maximum delay suffered by any cell belonging to the connection. We start with a simple system consisting of a single multiplexing node and find the worst-case delay for different traffic envelopes.

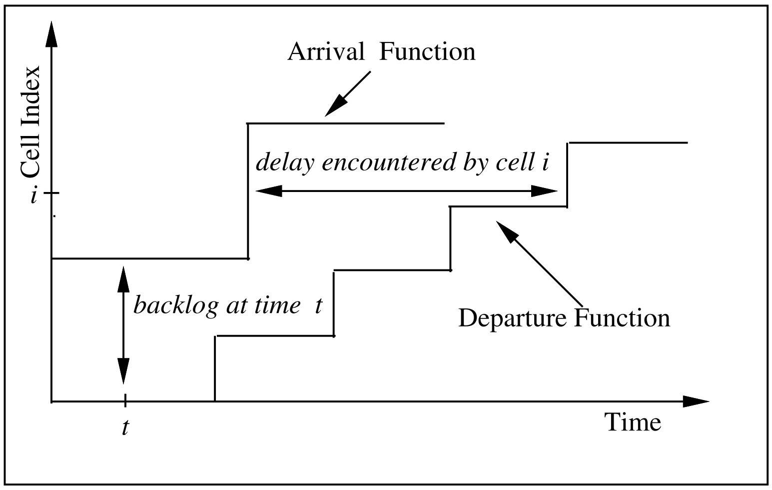

Figure 6.1: Computing delay and backlog from the arrival and departure functions.

6.1.1 Single-node Case

Let us consider a switching node employing CORR scheduling to multiplex traffic from different connections. Since we are interested in the worst-case delay behavior, and since the scheduler guarantees a minimum rate of service to all connections, we can consider each connection in isolation. The delay encountered by any cell belonging to a connection is the difference between its arrival and departure times. The arrival time of a cell can be obtained from the traffic envelope associated with a connection (see chapter 4). Theorem 5.1 expresses the worst-case departure time of a cell in terms of the service rate allocated to the connection and the length of the allocation cycle. Knowing both the arrival and the departure functions, we can compute the worst-case delay bound. In the rest of the section we derive delay bounds for arrival functions characterized by composite leaky bucket, moving window, and jumping window shapers.

The delay encountered by cell i is really the horizontal distance between the arrival and departure functions at i (see figure 6.1). Hence, the maximum delay encountered by any cell is the maximum horizontal distance between the arrival and the departure functions. Similarly, the vertical distance between these functions represents the backlog in the system. Unfortunately, finding the maximum delay, that is, the maximum horizontal difference between the arrival and the departure functions, is a difficult task. Hence, instead of finding the maximum delay directly by measuring the horizontal distance between these functions, we first determine the point at which the maximum backlog occurs and the index i of the cell which is at the end of the queue at that point. The worst-case delay is then computed by evaluating $d(i) - a(i)$, where $a(i)$ and $d(i)$ are the arrival and departure times of the ith cell, respectively. In the following, we carry out this procedure for arrival functions defined by composite leaky bucket, moving window, and jumping window shapers.

Lemma 6.1 *Consider a connection shaped using an n-component moving window shaper and passing through a single multiplexing node employing CORR scheduling with an allocation cycle of length T* [1]*. If the connection is allocated a service rate of R, then the worst case delay encountered by any cell belonging to the connection is upper bounded by*

$$D^{CORR/MW} \leq \begin{cases} \left\lceil \frac{m_j+\delta}{R} \right\rceil T - \sum_{l=j+1}^{n} \left(\frac{m_{l-1}}{m_l} - 1 \right) w_l, & \left\lceil \frac{R+\delta}{w_j\left(R/T - m_j/w_j\right)} \right\rceil = 1 \\ \left\lceil \frac{2m_j+\delta}{R} \right\rceil T - w_j - \sum_{l=j+1}^{n} \left(\frac{m_{l-1}}{m_l} - 1 \right) w_l, & \left\lceil \frac{R+\delta}{w_j\left(R/T - m_j/w_j\right)} \right\rceil > 1 \end{cases}$$

$$\textit{when} \quad \frac{m_j}{w_j} < \frac{R}{T} < \frac{m_{j+1}}{w_{j+1}}, \qquad j = 1, 2, \ldots, n-1.$$

Proof: First we will show that under the conditions stated above the system is stable. That is, the length of the busy period is finite. To prove that, it is sufficient to show that there exists a positive integer k such that the number of cells serviced in kw_j time is greater than or equal to km_j. In other words, we have to show that there exists a k such that

$$\begin{aligned} kw_j &\geq d(km_j - 1) \\ kw_j &\geq \left\lceil \frac{(km_j-1)+1+\delta}{R} \right\rceil T \\ kw_j &\geq \left[\frac{(km_j-1)+1+\delta}{R} + 1 \right] T \\ k &\geq \frac{R+\delta}{w_j\left(R/T - m_j/w_j\right)} \\ k &\geq \left\lceil \frac{R+\delta}{w_j\left(R/T - m_j/w_j\right)} \right\rceil. \end{aligned}$$

Clearly, for there to exist a positive integer k so that the above equality is satisfied, the following condition needs to hold.

$$R/T - m_j/w_j > 0 \quad or \quad R/T > m_j/w_j$$

By our assumption, $R/T > m_j/w_j$. Hence, the system is stable. Now, we need to determine the point at which the maximum backlog occurs. Depending on the value of k, the maximum backlog can occur at one of the two places.

[1] We assume that the cycle length is smaller than the smallest window period. Since R can take any real value we can choose an allocation with arbitrarily small cycle length. Hence, this assumption is not restrictive.

Case k=1: If $k = 1$, that is, when the traffic coming in during a time window of length w_j departs the system in the same window, the maximum backlog occurs at the arrival instant of the $(m_j-1)^{th}$ cell [2]. Clearly, the index of the cell at the end of the queue at that instant is $m_j - 1$. Hence, the maximum delay encountered by any cell under this scenario is the same as the delay suffered by the $(m_j - 1)^{th}$ cell and can be enumerated by computing $d(m_j - 1) - a(m_j - 1)$. We can evaluate $a(m_j - 1)$ as follows:

$$
\begin{aligned}
a(m_j - 1) &= \sum_{l=1}^{n} \left(\left\lfloor \frac{m_j - 1}{m_l} \right\rfloor - \left\lfloor \frac{m_j - 1}{m_{l-1}} \right\rfloor \frac{m_{l-1}}{m_l} \right) w_l \\
&= \sum_{l=1}^{j} \left(\left\lfloor \frac{m_j - 1}{m_l} \right\rfloor - \left\lfloor \frac{m_j - 1}{m_{l-1}} \right\rfloor \frac{m_{l-1}}{m_l} \right) w_l \\
&+ \sum_{l=j+1}^{n} \left(\left\lfloor \frac{m_j - 1}{m_l} \right\rfloor - \left\lfloor \frac{m_j - 1}{m_{l-1}} \right\rfloor \frac{m_{l-1}}{m_l} \right) w_l \\
&= 0 + \sum_{l=j+1}^{n} \left(\frac{m_j}{m_l} - 1 - \left(\frac{m_j}{m_{l-1}} - 1 \right) \frac{m_{l-1}}{m_l} \right) w_l \qquad \text{(since } m_{l-1} > m_l \text{ and } m_l \text{ divides } m_{l+1}\text{)} \\
&= \sum_{l=j+1}^{n} \left(\frac{m_{l-1}}{m_l} - 1 \right) w_l .
\end{aligned}
$$

Therefore, the worst case delay is bounded by

$$
\begin{aligned}
D^{CORR/MW} &\leq d(m_j - 1) - a(m_j - 1) \\
&\leq \left\lceil \frac{m_j + \delta}{R} \right\rceil T - \sum_{l=j+1}^{n} \left(\frac{m_{l-1}}{m_l} - 1 \right) w_l .
\end{aligned}
$$

Case k>1: When k is greater than 1, the connection busy period continues beyond the first window of length w_j. Since $R/T > m_j/w_j$, the rate of traffic arrival is lower than the rate of departure. Still, in this case not all cells that arrive during the first window of length w_j are served during that period and the left over cells are carried over into the next window. This is due to the fact that unlike the arrival function, the departure function starts at time $\lceil (1+\delta)/R \rceil$ instead of time 0. This is the case when $k = 1$ as well. However, in that case the rate of service is high enough to serve all the cells before the end of the first window. When $k > 1$ the backlog carried over from the first window is cleared in portions over the next $k - 1$ windows. Clearly, the backlogs carried over into subsequent windows diminish in size and are cleared completely by the end of the k^{th} window. Hence, the second window is the one where the backlog inherited from the last window is the maximum. Consequently, absolute backlog in the system reaches its maximum at the

[2] Note that cells are numbered from 0.

arrival of the $2m_j - 1$ cell. Hence, the maximum delay encountered by any cell under this scenario is the same as the delay suffered by the $(2m_j - 1)^{th}$ cell and can be enumerated by computing $d(2m_j - 1) - a(2m_j - 1)$. We can evaluate $a(2m_j - 1)$ as follows:

$$
\begin{aligned}
a(2m_j - 1) &= \sum_{l=1}^{n} \left(\left\lfloor \frac{2m_j - 1}{m_l} \right\rfloor - \left\lfloor \frac{2m_j - 1}{m_{l-1}} \right\rfloor \frac{m_{l-1}}{m_l} \right) w_l \\
&= \sum_{l=1}^{j-1} \left(\left\lfloor \frac{2m_j - 1}{m_l} \right\rfloor - \left\lfloor \frac{2m_j - 1}{m_{l-1}} \right\rfloor \frac{m_{l-1}}{m_l} \right) w_l \\
&\quad + \left(\left\lfloor \frac{2m_j - 1}{m_j} \right\rfloor - \left\lfloor \frac{2m_j - 1}{m_{j-1}} \right\rfloor \frac{m_{j-1}}{m_j} \right) w_j \\
&\quad + \sum_{l=j+1}^{n} \left(\left\lfloor \frac{2m_j - 1}{m_l} \right\rfloor - \left\lfloor \frac{2m_j - 1}{m_{l-1}} \right\rfloor \frac{m_{l-1}}{m_l} \right) w_l \\
&= 0 + w_j + \sum_{l=j+1}^{n} \left(\frac{2m_j}{m_l} - 1 - \left(\frac{2m_j}{m_{l-1}} - 1 \right) \frac{m_{l-1}}{m_l} \right) w_l \qquad (\text{since } m_{l-1} \geq 2m_l) \\
&= w_j + \sum_{l=j+1}^{n} \left(\frac{m_{l-1}}{m_l} - 1 \right) w_l.
\end{aligned}
$$

Therefore the worst case delay is bounded by

$$
\begin{aligned}
D^{CORR/MW} &\leq d(2m_j - 1) - a(2m_j - 1) \\
&\leq \left\lceil \frac{2m_j + \delta}{R} \right\rceil T - w_j - \sum_{l=j+1}^{n} \left(\frac{m_{l-1}}{m_l} - 1 \right) w_l.
\end{aligned}
$$

□

Lemma 6.2 *Consider a connection that is shaped using an n-component jumping window and passing through a single multiplexing node employing CORR scheduling with an allocation cycle of maximum length T. If the connection is allocated a service rate of R, then the worst-case delay suffered by any cell belonging to the connection is upper bounded by*

$$
D^{CORR/JW} \leq \begin{cases}
\left\lceil \frac{2m_1 + \delta}{R} \right\rceil T - 2\sum_{l=2}^{n} \left(\frac{m_{l-1}}{m_l} - 1 \right) w_l, & \frac{m_1}{w_1} < \frac{R}{T} < \frac{m_2}{w_2} \\
\left\lceil \frac{2m_j + \delta}{R} \right\rceil T - w_j - \sum_{l=1}^{j} \left(1 - \frac{m_{l-1}}{m_l} \right) w_l & \frac{m_j}{w_j} < \frac{R}{T} < \frac{m_{j+1}}{w_{j+1}}, \\
+w_1 - \sum_{l=2}^{n} \left(\frac{m_{l-1}}{m_l} - 1 \right) w_l, & j = 2, 3, \ldots, n-1
\end{cases}
$$

Proof: The proof is very similar to the last proof. The condition $m_j/w_j < R/T < m_{j+1}/w_{j+1}$, where $i = 1, 2, \ldots, n-1$, guarantees that the system is stable. The difference is that in the case of a jumping window, maximum backlog can occur at only one point, which is the arrival instant of cell $2m_j - 1$. This is because of the fact that unlike in a moving window, in a jumping window two bursts can come almost next to each other (see figure 4.8 for explanation). Hence, the worst case delay encountered by any cell is at most as large as the delay encountered by cell $2m_j - 1$ and can be computed as $d(2m_j - 1) + a(0) - a(2m_j - 1)$. Note that we have an additional $a(0)$ term here in the expression of delay since the arrival curve does not start at time 0 but at time $a(0)$. To compensate for that we have shifted the departure curve also by $a(0)$ in the expression for delay. The final expression is in two parts since depending on the value of j, $a(2m_j - 1)$ can have one of the two forms (refer to theorem 4.3).

When $j = 1$, $a(2m_j - 1)$ is computed as follows:

$$\begin{aligned}
a(2m_1 - 1) &= \sum_{l=1}^{n} \left(\left\lfloor \frac{2m_1 - 1}{m_l} \right\rfloor - \left\lfloor \frac{2m_1 - 1}{m_{l-1}} \right\rfloor \frac{m_{l-1}}{m_l} \right) w_l \\
&= \left(\left\lfloor \frac{2m_1 - 1}{m_1} \right\rfloor - \left\lfloor \frac{2m_1 - 1}{m_0} \right\rfloor \frac{m_0}{m_1} \right) w_1 \\
&\quad + \sum_{l=2}^{n} \left(\left\lfloor \frac{2m_1 - 1}{m_l} \right\rfloor - \left\lfloor \frac{2m_1 - 1}{m_{l-1}} \right\rfloor \frac{m_{l-1}}{m_l} \right) w_l \\
&= w_1 + \sum_{l=2}^{n} \left(\frac{2m_1}{m_l} - 1 - \left(\frac{2m_1}{m_{l-1}} - 1 \right) \frac{m_{l-1}}{m_l} \right) w_l \\
&= w_1 + \sum_{l=2}^{n} \left(\frac{m_{l-1}}{m_l} - 1 \right) w_l .
\end{aligned}$$

When $j = 2, 3, \ldots, n-1$, $a(2m_j - 1)$ is computed as follows,

$$\begin{aligned}
a(2m_j - 1) &= \sum_{l=1}^{n} \left\{ \left(\left\lfloor \frac{2m_j - 1}{m_l} \right\rfloor + 1 \right) - \left(\left\lfloor \frac{2m_j - 1}{m_{l-1}} \right\rfloor + 1 \right) \frac{m_{l-1}}{m_l} \right\} w_l \\
&= \sum_{l=1}^{j-1} \left\{ \left(\left\lfloor \frac{2m_j - 1}{m_l} \right\rfloor + 1 \right) - \left(\left\lfloor \frac{2m_j - 1}{m_{l-1}} \right\rfloor + 1 \right) \frac{m_{l-1}}{m_l} \right\} w_l \\
&\quad + \left\{ \left(\left\lfloor \frac{2m_j - 1}{m_j} \right\rfloor + 1 \right) - \left(\left\lfloor \frac{2m_j - 1}{m_{j-1}} \right\rfloor + 1 \right) \frac{m_{j-1}}{m_j} \right\} w_j \\
&\quad + \sum_{l=j+1}^{n} \left\{ \left(\left\lfloor \frac{2m_j - 1}{m_l} \right\rfloor + 1 \right) - \left(\left\lfloor \frac{2m_j - 1}{m_{l-1}} \right\rfloor + 1 \right) \frac{m_{l-1}}{m_l} \right\} w_l \\
&= \sum_{l=1}^{j-1} \left(1 - \frac{m_{l-1}}{m_l} \right) w_l + \left(2 - \frac{m_{j-1}}{m_j} \right) w_j
\end{aligned}$$

$$
\begin{aligned}
& + \sum_{l=j+1}^{n} \left\{ \left(\frac{2m_j}{m_l} - 1 + 1 \right) - \left(\frac{2m_j}{m_{l-1}} - 1 + 1 \right) \frac{m_{l-1}}{m_l} \right\} w_l \\
= \; & w_j + \sum_{l=1}^{j} \left(1 - \frac{m_{l-1}}{m_l} \right) w_l .
\end{aligned}
$$

We can compute $a(0)$ in a similar fashion. Now $d(2m_j - 1) + a(0) - a(2m_j - 1)$ yields the result. □

Lemma 6.3 *Consider a connection shaped by an n-component leaky bucket shaper and passing through a single multiplexing node employing CORR scheduling with an allocation cycle of maximum length T. If the connection is allocated a service rate of R, then the worst case delay suffered by any cell belonging to the connection is upper bounded by*

$$
D_{max}^{CORR/LB} \leq \left\lceil \frac{B_j + 1 + \delta}{R} \right\rceil T - (B_j - b_j + 1)\, t_j ,
$$

$$
\textit{when} \qquad \frac{1}{t_j} < \frac{R}{T} < \frac{1}{t_{j+1}}, \qquad j = 1, 2, \ldots, n.
$$

Proof: In order to identify the point where the maximum the backlog occurs, observe that the rate of arrivals is more than the rate of service until the slope of the traffic envelope changes from $\frac{1}{t_{j+1}}$ to $\frac{1}{t_j}$. This change in the slope occurs at the arrival of the ${B_j}^{th}$ cell in the worst case. Hence, the maximum delay encountered by any cell is at most as large as the delay suffered by cell B_j. We can compute $a(B_j)$ as follows:

$$
\begin{aligned}
a(B_j) \; = \; & \sum_{l=1}^{n+1} (B_j - b_l + 1)\, t_l \,[U(B_j - B_l) - U(B_j - B_{l-1})] \\
= \; & \sum_{l=1}^{j-1} (B_j - b_l + 1)\, t_l \,[U(B_j - B_l) - U(B_j - B_{l-1})] \\
& + (B_j - b_j + 1)\, t_j \,[U(B_j - B_j) - U(B_j - B_{j-1})] \\
& + \sum_{l=j+1}^{n+1} (B_j - b_l + 1)\, t_l \,[U(B_j - B_l) - U(B_j - B_{l-1})] \\
= \; & (B_j - b_j + 1)\, t_j .
\end{aligned}
$$

Now $d(B_j) - a(B_j)$ yields the result. □

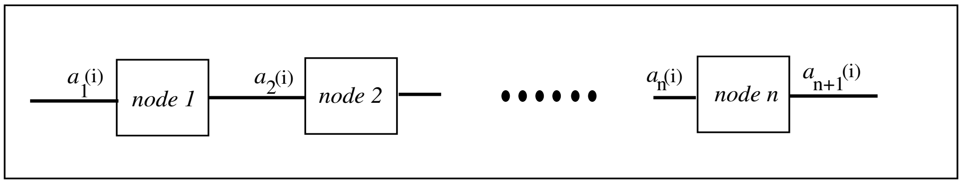

Figure 6.2: Nodes in tandem.

The results derived in this section define upper bounds for delay encountered in a CORR server under different traffic envelopes. The compact closed form expressions make the task of computing the numerical bounds for a specific set of parameters simple. We would also like to mention that compared to other published works, we consider a much larger and general set of traffic envelopes in our analysis. Although simple closed form bounds under very general arrival patterns are one of the major attractions of the CORR scheduling and an important contribution of this study, bounds for a single-node system are not very useful in a real-life scenario. In most systems, a connection spans multiple nodes, and the end-to-end delay bound is of real interest. In the next section, we derive bounds on end-to-end delay.

6.1.2 Multiple-node Case

In the last section, we derived worst-case bounds on delay for different traffic envelopes for a single-node system. In this section, we derive similar bounds for a multi-node system. We assume that there are n multiplexing nodes between the source and the destination, and at each node a minimum available rate of service is guaranteed.

We denote the arrival time of cell i at node k by $a_k(i)$. The service time at node k for cell i is denoted by $s_k(i)$. Without loss of generality, we assume that the propagation delay between nodes is zero[3]. Hence, the departure time of cell i from node k is $a_{k+1}(i)$. Note that $a_1(i)$ is the arrival time of the i^{th} cell in the system and $a_{n+1}(i)$ is the departure time of the i^{th} cell from the system (see figure 6.2).

Let us denote $\sum_{i=p}^{q} s_k(i)$ by $S_k(p,q)$. This is nothing other than the aggregate service times of cells p through q at node k. In other words, $S_k(p,q)$ is the service time of the burst of cells p through q at node k.

The following theorem expresses the arrival time of a particular cell at a specific node in terms of the arrival times of the preceding cells at the system and their service times at different nodes.

[3]This assumption does not affect the generality of the results, since the propagation delay at each stage is constant and can be included in $s_k(i)$.

This is a very general result and is independent of the particular scheduling discipline used at the multiplexing node and traffic envelope associated with the connection. We will use this result to derive the worst-case bound on end-to-end delay.

Theorem 6.1 *For any node k and for any cell i, the following holds:*

$$a_k(i) = \max_{1 \le j \le i} \left\{ a_1(j) + \max_{j=l_1 \le l_2 \le \cdots \le l_k = i} \left(\sum_{h=1}^{k-1} S_h(l_h, l_{h+1}) \right) \right\}. \tag{6.1}$$

Proof: We will prove this theorem by induction on k and i.

Induction on k:

Base Case: When $k = 1$,

$$\begin{aligned}
a_1(i) &= \max_{1 \le j \le i} \left\{ a_1(j) + \max_{j=l_1 \le l_2 \le \cdots \le l_k = i} \left(\sum_{h=1}^{0} S_h(l_h, l_{h+1}) \right) \right\} \\
&= a_1(i).
\end{aligned}$$

Clearly, the assertion holds.

Inductive Hypothesis: Let us assume that the premise holds for all $m \le k$. In order to prove that the hypothesis is correct, we need to show that it holds for $m = k+1$.

$$\begin{aligned}
a_{k+1}(i) &= \max \{ a_{k+1}(i-1) + s_k(i), a_k(i) + s_k(i) \} \\
&= \max \left\{ \max_{1 \le j \le i-1} \left[a_1(j) + \max_{j=l_1 \le l_2 \le \cdots \le l_{k+1} = i-1} \left(\sum_{h=1}^{k} S_h(l_h, l_{h+1}) \right) \right] + s_k(i), \right. \\
&\qquad \left. \max_{1 \le j \le i} \left[a_1(j) + \max_{j=l_1 \le l_2 \le \cdots \le l_k = i} \left(\sum_{h=1}^{k-1} S_h(l_h, l_{h+1}) \right) \right] + s_k(i) \right\} \\
&= \max \left\{ \max_{1 \le j \le i-1} \left[a_1(j) + \max_{j=l_1 \le l_2 \le \cdots \le l_{k+1} = i-1} \left(\sum_{h=1}^{k} S_h(l_h, l_{h+1}) \right) + s_k(i) \right], \right. \\
&\qquad \max_{1 \le j \le i-1} \left[a_1(j) + \max_{j=l_1 \le l_2 \le \cdots \le l_k = i} \left(\sum_{h=1}^{k-1} S_h(l_h, l_{h+1}) \right) + S_k(i,i) \right], \\
&\qquad \left. a_1(i) + \sum_{h=1}^{i} S_h(i,i) \right\} \\
&= \max \left\{ \max_{1 \le j \le i-1} \left[a_1(j) + \max_{j=l_1 \le l_2 \le \cdots \le l_k < l_{k+1} = i} \left(\sum_{h=1}^{k} S_h(l_h, l_{h+1}) \right) \right], \right.
\end{aligned}$$

$$
\begin{aligned}
& \max_{1\le j\le i-1}\left[a_1(j)+\max_{j=l_1\le l_2\le\cdots\le l_k=l_{k+1}=i}\left(\sum_{h=1}^{k} S_h(l_h,l_{h+1})\right)\right], \\
& a_1(i)+\sum_{h=1}^{k} S_h(i,i)\Bigg\} \\
= \;& \max\Bigg\{\max_{1\le j\le i-1}\left[a_1(j)+\max_{j=l_1\le l_2\le\cdots\le l_k\le l_{k+1}=i}\left(\sum_{h=1}^{k} S_h(l_h,l_{h+1})\right)\right], \\
& a_1(i)+\sum_{h=1}^{i} S_h(i,i)\Bigg\} \\
= \;& \max_{1\le j\le i}\left\{a_k(j)+\max_{j=l_1\le l_2\le\cdots\le l_k\le l_{k+1}=i}\left(\sum_{h=1}^{k} S_h(l_h,l_{h+1})\right)\right\}.
\end{aligned}
$$

Induction on i:

Base Case: When $i=1$,

$$
\begin{aligned}
a_k(1) &= \max_{1\le j\le 1}\left\{a_1(j)+\max_{j=l_1\le l_2\le\cdots\le l_k=1}\left(\sum_{h=1}^{k-1} S_h(l_h,l_{h+1})\right)\right\} \\
&= a_1(1)+\sum_{h=1}^{k-1} S_h(1,1) \\
&= a_1(1)+\sum_{h=1}^{k-1} s_h(1).
\end{aligned}
$$

Hence, the assertion holds in the base case.

Inductive Hypothesis: Let us assume that the premise holds for all $n\le i$. In order to prove that the hypothesis is correct, we need to show that it holds for $n=i+1$.

$$
\begin{aligned}
a_k(i+1) &= \max\{a_k(i)+s_{k-1}(i+1), a_{k-1}(i+1)+s_{k-1}(i+1)\} \\
&= \max\Bigg\{\max_{1\le j\le i}\left[a_1(j)+\max_{j=l_1\le l_2\le\cdots\le l_k=i}\left(\sum_{h=1}^{k-1} S_h(l_h,l_{h+1})\right)\right]+s_{k-1}(i+1), \\
&\qquad \max_{1\le j\le i+1}\left[a_1(j)+\max_{j=l_1\le l_2\le\cdots\le l_{k-1}=i+1}\left(\sum_{h=1}^{k-2} S_h(l_h,l_{h+1})\right)\right]+s_{k-1}(i+1)\Bigg\} \\
&= \max\Bigg\{\max_{1\le j\le i}\left[a_1(j)+\max_{j=l_1\le l_2\le\cdots\le l_k=i}\left(\sum_{h=1}^{k-1} S_h(l_h,l_{h+1})\right)+s_{k-1}(i+1)\right],
\end{aligned}
$$

$$
\begin{aligned}
& \max_{1\le j\le i}\left[a_1(j) + \max_{j=l_1\le l_2\le\cdots\le l_{k-1}=i+1}\left(\sum_{h=1}^{k-2} S_h(l_h,l_{h+1})\right) + S_{k-1}(i+1,i+1)\right], \\
& a_1(i+1) + \sum_{h=1}^{k-1} S_h(i+1,i+1)\Bigg\} \\
= \; & \max\Bigg\{\max_{1\le j\le i}\left[a_1(j) + \max_{j=l_1\le l_2\le\cdots\le l_{k-1}<l_k=i+1}\left(\sum_{h=1}^{k-1} S_h(l_h,l_{h+1})\right)\right], \\
& \max_{1\le j\le i}\left[a_1(j) + \max_{j=l_1\le l_2\le\cdots\le l_{k-1}=l_k=i+1}\left(\sum_{h=1}^{k-1} S_h(l_h,l_{h+1})\right)\right], \\
& a_1(i+1) + \sum_{h=1}^{i} S_h(i+1,i+1)\Bigg\} \\
= \; & \max\Bigg\{\max_{1\le j\le i}\left[a_1(j) + \max_{j=l_1\le l_2\le\cdots\le l_{k-1}\le l_k=i+1}\left(\sum_{h=1}^{k-1} S_h(l_h,l_{h+1})\right)\right], \\
& a_1(i+1) + \sum_{h=1}^{k} S_h(i+1,i+1)\Bigg\} \\
= \; & \max_{1\le j\le i+1}\left\{a_k(j) + \max_{j=l_1\le l_2\le\cdots\le l_{k-1}\le l_k=i+1}\left(\sum_{h=1}^{k-1} S_h(l_h,l_{h+1})\right)\right\}.
\end{aligned}
$$

□

The result stated in the above theorem determines the departure time of any cell from any node in the system in terms of the arrival times of the preceding cells and the service times of the cells at different nodes. This is the most general result known to us on end-to-end delay in terms of service times of cells at intermediate nodes. We believe that this result is a powerful tool in enumerating end-to-end delay for any rate based scheduling discipline and is an effective alternative for the ad hoc techniques commonly used for end-to-end analysis.

Although the result stated in theorem 6.1 is very general, it is difficult to make use of it in its current form. In order to find the exact departure time of any cell from any node, we need to know both the arrival times of the cells and their service times at different nodes. Arrival times of different cells can be obtained from the arrival function, but computing service times for different cells at each node is a daunting task. Hence, computing the exact departure time of a cell from any node in the system is often quite difficult. However, the accurate departure time of a specific cell is rarely of critical interest. More often we are interested in other metrics, such as the worst-case delay encountered by a cell. Fortunately, computing the worst-case bound on the departure time, and then the worst-case delay is not as difficult. The following corollary expresses the worst-case delay suffered by a cell in terms of the worst-case service times at each node.

Corollary 6.1 *Consider a connection passing through n multiplexing nodes. Assume that there exists a $S_w(\cdot)$ such that $S_w(p,q) \geq S_h(p,q)$ for all $q \geq p$ and $h = 1, 2, \ldots n$. Then, in the worst case, delay $D(i)$ suffered by the cell i belonging to the connection can be upper bounded by*

$$D(i) \leq \max_{1 \leq j \leq i} \left\{ a_1(j) + \max_{j = l_1 \leq l_2 \leq \cdots \leq l_{n+1} = i} \left\{ \sum_{h=1}^{n} S_w(l_h, l_{h+1}) \right\} \right\} - a_1(i)$$

Proof: The proof follows trivially from theorem 6.1 by substituting S_h, $h = 1, 2, \ldots n$ by S_w. □

Corollary 6.1 expresses the worst case delay encountered by any cell under the assumption that for any p and q there exists a function S_w such that $S_w(p,q) \geq S_h(p,q)$, for $h = 1, 2, \ldots, n$. The closer S_w is to S_h, the tighter is the bound. The choice of S_w depends on the particular scheduling discipline used at the multiplexing nodes. In the case of carry-over round robin, it is simply the service time at the minimum guaranteed rate of service. The following corollary instantiates the delay bound for CORR service discipline.

Corollary 6.2 *Consider a connection traversing n nodes, each of which employs carry-over round robin scheduling discipline. Let R_w be the minimum rate of service offered to a connection at the bottleneck node, and let T be the maximum length of the allocation cycle. Then the worst case delay suffered by the i^{th} cell belonging to the connection is bounded by*

$$D^{CORR}(i) \leq \left\lceil n + (n-1)\frac{2 + \delta_w}{R_w} \right\rceil T + \max_{1 \leq j \leq i} \{a_1(j) + S_w(j,i)\} - a_1(i)$$

Proof: This follows from corollary 6.1 by replacing

$$\max_{j = l_1 \leq l_2 \leq \cdots \leq l_{n+1} = i} \left\{ \sum_{h=1}^{n} S_w(l_h, l_{h+1}) \right\} \quad \textit{with} \quad \left\lceil n + (n-1)\frac{2 + \delta_w}{R_w} \right\rceil T + S_w(j,i)$$

The following steps explain the details.

$$\begin{aligned}
\sum_{h=1}^{n} S_w(l_h, l_{h+1}) &= \sum_{h=1}^{n} \left\lceil \frac{l_{(h+1} - l_h + 1) + 1 + \delta_w}{R_w} \right\rceil T \qquad (\textit{from theorem } 5.1) \\
&\leq \sum_{h=1}^{n} \left\lceil \frac{l_{h+1} - l_h + 2 + \delta_w}{R_w} + 1 \right\rceil T \\
&\leq \left\lceil n + (n-1)\frac{2 + \delta_w}{R_w} \right\rceil T + \left\lceil \frac{(l_{n+1} - l_1 + 1) + 1 + \delta_w}{R_w} \right\rceil T \\
&\leq \left\lceil n + (n-1)\frac{2 + \delta_w}{R_w} \right\rceil T + S_w(l_1, l_{n+1}) \qquad (\textit{from theorem } 5.1) \\
&\leq \left\lceil n + (n-1)\frac{2 + \delta_w}{R_w} \right\rceil T + S_w(j,i) \qquad (\textit{putting } l_1 = j \textit{ and } l_{n+1} = i)
\end{aligned}$$

The final result follows immediately. □

The expression for D^{CORR} derived above consists of two main terms. The first term is a constant independent of the cell index. If we observe the second term carefully, we realize that it is nothing other than the delay encountered by the i^{th} cell at CORR server with a cycle time T and a minimum rate of service R_w. Hence, the end-to-end delay reduces to the sum of the delay encountered in a single-node system and a constant. By substituting the delay bounds for the single-node system derived in the last section, we can enumerate the end-to-end delay in a multi-node system for different traffic envelopes.

6.1.3 Comparison with Other Schemes

As discussed in chapter 5, a plethora of multiplexing disciplines providing rate guarantees have been proposed in the last a few years. Among these, the scheduling mechanisms closest to ours are Stop-and-Go (SG) [23, 24] and Packet-by-Packet Generalized Processor Sharing (PGPS) [35]. Both SG and PGPS assume a fluid model of data flow. In order to compare them with CORR discipline, we transformed the results derived for SG and PGPS into equivalent results in the discrete time model used in our study.

In SG, each link divides the timeline into frames of size F. The admission policy under which delay guarantees can be made is that no more than m cells are admitted into the system in any frame time, where $r = m/F$ is the transmission rate assigned to a particular connection. A traffic stream that obeys this restriction is said to be (r, F) smooth. The concept of adjacent frames is central to defining the stop-and-go queueing. For each frame on an incoming link, there is an adjacent frame on the outgoing link. The cells coming in during a frame time on the incoming link are dispatched on the corresponding adjacent frame on the outgoing link. In some sense, SG tries to emulate circuit switching on top of packet switching. It has been shown that when the sum of the transmission rates assigned to all active connections at each node is less than the link capacity, the end-to-end delay for a connection spanning n nodes is bounded by

$$nF \leq D^{SG} \leq 2nF.$$

Clearly, the traffic model used in SG is far more restricted than ours and requires allocation of bandwidth at the peak rate. Also, the granularity of bandwidth allocation depends on the size of the frame. The larger the frame size is the finer the granularity of bandwidth allocation gets. However, larger frame size leads to higher delays. To alleviate this problem, SG supports different frame sizes to satisfy different rate and delay requirements. Another problem with SG is that it requires networkwide standardization of frame sizes.

Comparing the delay bounds achieved by SG and CORR is difficult because of the difference in the traffic model. While SG assumes arrivals and service at the peak rate, we can exploit the variation in traffic arrival rate using a composite shaping envelope and choosing a service rate just above the average rate of arrivals. Nevertheless, to get a rough idea of how the end-to-end delay bounds for these schemes compare, let us consider a connection shaped using a 2-component moving window shaper (m_1, w_1) and (m_2, w_2), where $m_1 > m_2$, $w_1 > w_2$ and $m_1/w_1 < m_2/w_2$. Let T be the cycle length for CORR, and let R be the number of slots allocated to this connection in each cycle. Since SG admits traffic at the peak rate, $r = m_2/w_2$. If the connection traverses n nodes, we get the following bounds on the end-to-end delay (after some simplifications)

$$\begin{aligned} D^{SG} &\leq 2nF. \\ D^{CORR} &\leq n\left(1 + \frac{2+\delta}{R}\right)T + \left\lceil \frac{2m_1}{R} \right\rceil T - w_1 - \left(\frac{m_1}{m_2} - 1\right) w_2. \end{aligned}$$

If we consider n to be very large, the first term in the bound on D^{CORR} is the dominant term, and we can neglect the second term. Thus, $D^{CORR} < D^{SG}$ when $R > (2+\delta)T/(2F - T)$. When is n not large and the second term in bound on D^{CORR} is not negligible, the bound depends on the exact values of the parameters. Clearly, delay bounds are competitive. However, because of the flexibility in choosing the appropriate service rate exploiting the variation in traffic arrivals, CORR can potentially admit more connections than SG. SG assumes the peak rate arrival of traffic. In this particular example, bandwidth allocated to the connection is m_2/w_2. In contrast, in CORR bandwidth allocated to the connection can vary from m_1/w_1 to m_2/w_2, depending on the delay requirements. This gives us a lot of room in terms of allocating the right amount of bandwidth satisfying the end-to-end delay constraints, thereby increasing the number of admissible connections.

PGPS is a packet-by-packet implementation of the fair queueing mechanism [13, 35]. In PGPS, incoming cells from different connections are buffered in a sorted priority queue and is served in the order in which they would leave the server in an ideal fair queueing system. The departure times of cells in the fair queueing system is enumerated by simulating the reference ideal system. Simulation of the reference system and maintenance of the priority queue are both quite expensive operations. Hence, the implementation of the PGPS scheme in a real system is difficult, to say the least. Nevertheless, we compare CORR with PGPS to show how the delay performance of CORR compares with that of a near ideal scheme.

It has been shown in [34] that with a (b,t) leaky bucket controlled source and a guaranteed rate of service $R > 1/t$ at each node, the worst case end-to-end delay of a connection traversing n nodes is bounded by

$$D^{PGPS} \leq bt + (n-1)t + \sum_{k=1}^{n} \tau_k,$$

where τ_k is the service time of a cell at the k^{th} server operating at the maximum rate. For small cell sizes and high transmission rates, the third term is negligible and is not considered henceforth. The assumption that the rate of service $R > 1/t$ at each node means that there is no queueing in the system. Under that assumption, the delay bound derived above does not come as a big surprise. As in the case of SG, the traffic model used in PGPS is also very restricted. In fact the author in [34] suggests a generalization of the traffic model as an important extension of his work. Non-availability of delay bounds of PGPS for more general traffic models makes the task of comparing it with CORR difficult. In any case, to get a rough idea of how these bounds compare, let us consider a connection shaped using two leaky buckets (b_1, t_1) and (b_2, t_2), where $b_1 > b_2$ and $t_1 > t_2$. Let T be the length of the allocation cycle in CORR, and R be the slots allocated to the connection in each cycle. Since PGPS uses only single leaky bucket characterization of the source, the traffic envelope for PGPS is determined by (b_2, t_2). If we consider a connection spanning n nodes, we get the following bounds on end-to-end delay

$$\begin{aligned} D^{PGPS} &\leq (n-1)t_2 + b_2 t_2. \\ D^{CORR} &\leq (n-1)\left(1 + \frac{2+\delta}{R}\right)T + \left(1 + \left\lceil \frac{b_1+2}{R} \right\rceil\right)T - \left(\left\lfloor \frac{b_1 t_1 - b_2 t_2}{t_1 - t_2} \right\rfloor - b_1 + 1\right)t_1. \end{aligned}$$

If we assume n to be large, the first terms in the bounds on D^{PGPS} and D^{CORR} are the dominant terms. If we also assume that $t_2 = T/R$, then D^{CORR} is close to $(2+\delta) \times D^{PGPS}$. Hence, as expected, PGPS performs better than CORR in terms of worst case delay bound. However, the delay bounds are quite competitive. When n is not negligible, the bounds depend on the exact values of the parameters. Although for large n D^{PGPS} is better than D^{CORR} in terms of worst case end-to-end delay performance, the number of connections admitted under CORR may still be more than that under PGPS. This is because of the fact that in PGPS the bandwidth allocated to a connection depends on the parameters of the leaky bucket used to regulate its traffic flow, not so much on the delay requirement. For example, in this particular case, bandwidth allocated to the the connection is $1/t_2$. In contrast, CORR provides us with the flexibility to choose the right amount of bandwidth, just enough to satisfy the end-to-end delay requirement. In this case, under CORR scheme, we can allocate any amount of bandwidth between $1/t_2$ to $1/t_1$ satisfying the end-to-end delay constraints. Consequently, more connections may be admitted.

6.2 Fairness Analysis

In the last section we analyzed some of the worst-case behavior of the system. In the worst-case analysis it is assumed that the system is fully loaded, and each connection is served at the minimum guaranteed rate. However, that is often not the case. In a work conserving server, when the system is not fully loaded the spare capacity can be used by the busy connections to achieve better performance. One of the important performance metrics of a work conserving server is the fairness. That is, how fair is the server in distributing the excess capacity among the active connections. In the rest of the section we define a measure of fairness, analyze the fairness properties of CORR scheduling, and compare CORR with other fair queueing mechanisms.

Let us denote the number of cells of connection C_p transmitted during [0,t) by $N_p(t)$. We define the normalized work received by a connection p as $w_p(t) = N_p(t)/R_p$, where R_p is the service rate of connection C_p. Accordingly, $w_p(t_1, t_2) = w_p(t_2) - w_p(t_1)$, where $t_1 \leq t_2$ is the normalized service received by connection C_p during $[t_1, t_2)$.

In an ideally fair system, the normalized service received by different connections in their busy state increases at the same rate. That is, when both connections C_p and C_q are in their busy period,

$$\frac{d}{dt} w_p(t) = \frac{d}{dt} w_q(t).$$

For connections that are not busy at t, normalized service stays constant, that is,

$$\frac{d}{dt} w_p(t) = 0.$$

If two connections C_p and C_q are both in their busy period during $[t_1, t_2)$, we can easily show that $w_p(t_1, t_2) = w_q(t_1, t_2)$.

Unfortunately, the notion of ideal fairness is only applicable to hypothetical fluid flow models. In a real packet network, a complete packet from one connection has to be transmitted before service is shifted to another connection. Therefore, it is not possible to satisfy the equality of normalized rate of services for all busy connections at all times. However, it is possible to keep the normalized services received by different connections close to each other. The *Packet-by-Packet Generalized Processor Sharing* (PGPS) and the *Self-Clocked-Fair-Queueing* (SFQ) are close approximations to *ideal-fair-queueing* in the sense that they try to keep the normalized services received by busy sessions close to that of an ideal system. Unfortunately, the realization of PGPS and SFQ are quite complex. Following, we derive the fairness properties of CORR scheduling and compare them with those of PGPS and SFQ.

6.2.1 Fairness of CORR

For the sake of simplicity, we assume that our sampling points coincide with the beginning of the allocation cycles only. If frame sizes are small, this approximation is quite reasonable.

Lemma 6.4 *If a connection p is in a busy period during cycles k_1 through k_2, where $k_2 \geq k_1$, the amount of service received by the connection during $[k_1, k_2]$ is bounded by*

$$max\{0, \lfloor (k_2 - k_1)R_p - \delta_p \rfloor\} \leq N_p(k_1, k_2) \leq \lceil (k_2 - k_1)R_p + \delta_p \rceil.$$

Proof: The proof follows directly from lemma 5.2. □

Corollary 6.3 *If a connection C_p is in a busy period during the cycles k_1 through k_2, where $k_2 \geq k_1$, the amount of normalized service received by the connection during $[k_1, k_2]$ is bounded by*

$$max\left\{0, \frac{\lfloor (k_2 - k_1)R_p - \delta_p \rfloor}{R_p}\right\} \leq w_p\,(k_1, k_2) \leq \frac{\lceil (k_2 - k_1)R_p + \delta_p \rceil}{R_p}.$$

Proof: The proof follows directly from lemma 6.4 and the definition of normalized service. □

Theorem 6.2 *If two connections p and q are in their busy periods during cycles k_1 through k_2, where $k_2 \geq k_1$, then*

$$\Phi(p, q) = |w_p(k_1, k_2) - w_q(k_1, k_2)| \leq \frac{1 + \delta_p}{R_p} + \frac{1 + \delta_q}{R_q}$$

Proof: From corollary 6.3 we get

$$\begin{aligned}
\Phi(p, q) &= |w_p(k_1, k_2) - w_q(k_1, k_2)| \\
&\leq max\left\{\left|\frac{\lceil (k_2 - k_1)R_p + \delta_p \rceil}{R_p} - \frac{\lfloor (k_2 - k_1)R_q - \delta_q \rfloor}{R_q}\right|,\right. \\
&\qquad \left.\left|\frac{\lceil (k_2 - k_1)R_q + \delta_q \rceil}{R_q} - \frac{\lfloor (k_2 - k_1)R_p - \delta_p \rfloor}{R_p}\right|\right\} \\
&\leq max\left\{\left|\frac{\left\lceil \left([k_2 - k_1] + \frac{\delta_p}{R_p}\right) R_p \right\rceil}{R_p} - \frac{\left\lfloor \left([k_2 - k_1] - \frac{\delta_q}{R_q}\right) R_q \right\rfloor}{R_q}\right|,\right. \\
&\qquad \left.\left|\frac{\left\lceil \left([k_2 - k_1] + \frac{\delta_q}{R_q}\right) R_q \right\rceil}{R_q} - \frac{\left\lfloor \left([k_2 - k_1] - \frac{\delta_p}{R_p}\right) R_p \right\rfloor}{R_p}\right|\right\}
\end{aligned}$$

$$
\begin{aligned}
&\leq \; max\left\{\left|\left([k_2-k_1]+\frac{1+\delta_p}{R_p}\right)-\left([k_2-k_1]-\frac{1+\delta_q}{R_q}\right)\right|\right. \\
&\qquad\qquad \left.\left|\left([k_2-k_1]+\frac{1+\delta_q}{R_q}\right)-\left([k_2-k_1]-\frac{1+\delta_p}{R_p}\right)\right|\right\} \\
&\leq \; \frac{1+\delta_p}{R_p}+\frac{1+\delta_q}{R_q}.
\end{aligned}
$$

This completes the proof. □

6.2.2 Comparison with Other Schemes

The notion of fainess was first applied to network systems in [13] using an idealized fluid flow model. In a fluid model of traffic, multiple connections can receive service in parallel, and hence it is possible to divide service capacity among active connections exactly in proportion of their service share at all times. As mentioned earlier, this ideal form of fair queueing cannot be implemented in practice. In [13, 35] an extension to the ideal fair queueing mechanism is proposed for packet switched network. In the scheme (PGPS) proposed in [35] the service order of packets are determined using the ideal fair queueing system as the reference, and by simulating the corresponding fluid flow model. The simulation of the hypothetical fluid model is computationally expensive and may be prohibitive, particularly at high transmission speed. The complexity of the of the PGPS scheme is somewhat alleviated in the SFQ mechanism proposed in [25]. In the SFQ scheme the hypothetical fluid flow simulation is eliminated by using an internal virtual time reference as an index of work. Although SFQ is far less complex than PGPS, it is still not simple enough for implementation at gigabit speed. The SFQ mechanism requires the waiting cells to be arranged in a priority queue sorted in an increasing order of their virtual departure time. Insertion in a hardware priority queue is an $O(n)$ operation [9], where n is the length of the queue. Hence, for a reasonably large value of n and at a high transmission speed, a priority queue of cells may be difficult to maintain. Compared to schemes described here, CORR scheduling is much simpler. Of course, simplicity comes at the cost of loss in fairness. In the following we show that although CORR is not as fair as PGPS and SFQ, it is quite close to them. But along with fairness, if we consider the complexity of implementation as one of the criteria for evaluation, we believe that CORR is an attractive choice.

To compare SFQ, PGPS, and CORR in terms of fairness, we use $\Phi(p,q)$ as the performance metric. As discussed earlier, $\Phi(p,q)$ is the absolute difference in normalized work received by two sessions over a time period where both of them were busy. We proved earlier that if our sample points are at the beginning of the allocation cycles, then

$$\Phi^{CORR}(p,q) \leq \frac{1+\delta_p}{R_p}+\frac{1+\delta_q}{R_q}.$$

Under the same scenario, it can be proved that in the SFQ scheme the following holds at all times,

$$\Phi^{SFQ}(p,q) \leq \frac{1}{R_p} + \frac{1}{R_q}$$

Due to difference in the definition of busy periods in PGPS, a similar result is difficult to derive. However, Golestani [25] has shown that the maximum permissible service disparity between a pair of busy connections in the SFQ scheme is never more than two times the corresponding figure for any real queueing scheme. This proves that

$$\Phi^{PGPS}(p,q) \geq \frac{1}{2}\Phi^{SFQ}(p,q).$$

Note that, $0 \leq \delta_i \leq 1$ for all connection i. Hence, the fairness index of CORR is within two times that of SFQ and at most four times that of any other queueing discipline, including PGPS. Clearly, it is a very competitive bound.

6.3 Summary

In this chapter, we have analyzed delay performance and fairness properties of CORR scheduling. We have derived closed form bounds on the worst-case delay when CORR is used in conjunction with composite leaky bucket, moving window, and jumping window regulators. We have shown that when the scheduling discipline guarantees a minimum rate of service at all nodes, the worst case end-to-end delay in a multi-node system can be reduced to the delay in an equivalent single-node system. We have used this result to derive corresponding end-to-end bounds on delay. We have shown that albeit its simplicity, CORR is very competitive with other more complex scheduling disciplines, such as PGPS and SFQ, in terms of both delay performance and fairness.

Chapter 7

Conclusions

In this dissertation, we have addressed two important problems hindering the ubiquitous deployment of distributed multimedia applications: 1) preserving network throughput at the end-hosts, and 2) developing traffic control mechanisms for providing service guarantees in ATM networks.

7.1 Contributions

We have addressed the first problem in chapters 2 and 3. In chapter 2, we proposed an I/O architecture which goes beyond preserving network throughput in end-system, and addresses chronic I/O bottleneck of the current generation of operating systems. The proposed architecture limits the operating system involvement in I/O transfers by migrating some of the operating system's I/O functions to the I/O devices. The presence of on-board micro-processors on most of the I/O adapters makes it a feasible task without a major overhaul of the device hardware. We believe that autonomy of devices, coupled with a connection oriented I/O architecture, is a fundamentally different and very promising approach to solving the I/O bottleneck in the host systems.

In chapter 3, we have experimentally demonstrated the performance impact of the proposed I/O architecture on networked multimedia applications. On a video conferencing system built around IBM RS/6000s equipped with high-performance video CODECs and connected via 100 Mb/s ATM links, we have shown that a connection oriented autonomous I/O architecture can improve end-to-end network throughput by as much as three times that achievable using the existing architecture.

Traffic control mechanisms for providing end-to-end service guarantees have been addressed in chapters 4, 5 and 6. The contribution of chapter 4 is in the precise characterization of traffic envelopes defined by composite shapers. These shaping envelopes are used to model input traffic in the delay analysis of the scheduling policies. With the recent standardization of multiple leaky-bucket-shaping of virtual connections by the ATM forum [21], our work on characterization of traffic generated by composite shapers is extremely important in developing end-to-end quality of service architecture.

In chapter 5, we have presented the algorithmic description of Carry-Over Round Robin Scheduling. The main attraction of CORR is its simplicity. In terms of complexity, CORR is comparable to round robin and frame based mechanisms. However, CORR does not suffer from the shortcomings of round robin and frame based schedulers. By allowing the number of slots allocated to a connection in an allocation cycle to be a real number instead of an integer, we break the coupling between the service delay and bandwidth allocation granularity. Also, unlike frame based mechanisms, such as Stop-and-Go and Hierarchical-Round-Robin, CORR is a work conserving discipline capable of exploiting the multiplexing gains of ATM.

The performance of the traffic control mechanism and the service policy has been analyzed in chapter 6. We have shown that, when used in conjunction with composite shapers, the CORR scheduling discipline is very effective in providing end-to-end delay guarantees. Besides providing guarantees on delay, CORR is also fair in distributing the excess bandwidth. Our results show that albeit its simplicity, CORR is very competitive with much more complex scheduling disciplines, such as Packet-by-Packet Generalized Processor Sharing and Self-Clocked Fair Queueing, both in terms of delay performance and fairness.

7.2 Future Directions

The work presented in this dissertation can be extended in several ways. In chapter 3, we demonstrated the effectiveness of the autonomous I/O architecture on a video conferencing application. A use of similar principles and mechanisms to other multi-media applications would be an interesting endeavor. Although video conferencing is representative of typical multi-media applications, a high-performance video server is much more demanding, both in terms of I/O throughput and network performance. We believe that the I/O architecture proposed in this dissertation is the only way to meet the I/O and network demands of a large video server. An experimental verification of this conjecture would be a worthwhile undertaking.

We have proposed multi-rate shaping as an effective mechanism to capture variability in traffic flows. We have analyzed end-to-end delay performance when CORR scheduling is used in conjunction with multi-rate shapers. It would be interesting to perform similar analyses for other scheduling disciplines proposed in the literature. Most of the analysis of packet scheduling disciplines assume very simple traffic models, such as peak rate or simple leaky bucket controlled sources. Since these models are inadequate in capturing the variability in traffic arrivals, the performance of the scheduling algorithms in exploiting the multiplexing gains of ATM has not been properly tested in these analysis. Re-evaluating them under the generalized traffic model will expose their strengths and weaknesses in a true packet switched environment.

Bibliography

[1] MMTplus Functional Description. *IBM Internal Document*, 1993.

[2] ATM Turboways 100 Adapter. *IBM Internal Document*, 1994.

[3] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanin, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the USENIX*, 1986.

[4] J. Adam, H. Houh, M. Ismert, and D. Tennenhuse. A Network Architecture for Distributed Multimedia Systems. In *Proceedings of the Multimedia Computing and Systems*, 1994.

[5] C. M. Aras, J. F. Kurose, D. S. Reeves, and H. Schulzrinne. Real-Time Communication in Packet-Switched Networks. *IEEE Transsactions on Information Theory*, 82(1), 1994.

[6] B. Bershad, C. Chanmers, S. Eggers C. Maeda, D. McNamee, P. Pyzemyslaw, S. Savage, and S. Emin Gon. SPIN - An Extensible Microkernel for Application-Specific Operating System Services. In *Department of Computer Science and Engineering FR-35, TR-94-03-03, University of Washington*, 1994.

[7] S. Keshav C. R. Kalmanek, H. Kanakia. Rate Controlled Servers for Very High Speed Networks. In *Proceedings, GLOBECOM*, 1990.

[8] J.B. Carter, A. Davis, R. Kuramkote C.-C. Kuo, L.B. Stoller, and M. Swanson. Avalanche: A Communication and Memory Architecture for Scalable Parallel Computing. *Draft Report, Computer Systems Laboratory, University of Utah*, 1995.

[9] H. J. Chao. Architecture Design for Regulating and Scheduling User's Traffic in ATM Networks. In *Proceedings, SIGCOMM*, 1992.

[10] David Cheriton. The V Distributed System. *Communications of the ACM*, 33(3), March 1988.

[11] D. Clark, B. Davie, and I. Gopal et al. The AURORA Gigabit Testbed. *Computer Networks and ISDN Systems*, 25(6), 1993.

[12] D. Clark, and D. Tennenhouse. Architectural Cosideration for a New Generation of Protocols. In *Proceedings, ACM SIGCOMM*, 1990.

[13] A. Demers, S. Keshav, and S. Shenkar. Anslysis and Simulation of Fair Queuing Algorithm. In *Proceedings, SIGCOMM*, 1989.

[14] P. Druschel, M. Abbott, M. Pagels, and L. Peterson. Analysis of I/O Subsystem Design for Multimedia Workstations. In *Proceedings of the Workshop on Network and Operating System Support for Digital Audio and Video*, 1992.

[15] P. Druschel, and L. Peterson. Fbufs: A High-Bandwidth Cross-Domain Transfer Facility. In *Proceedings of the SOSP*, 1993.

[16] P. Druschel, L. Peterson, and B.S. Davie. Experiences with a High-Speed Network Adapter: A Software Perspective. In *Proceedings, ACM SIGCOMM*, 1994.

[17] D.R. Engler, M.F. Kaashoek, and J. O'Toole. The Operating System Kernel as a Secure Programmable Machine. In *Proceedings of the sixth SIGOPS European Workshop*, 1994.

[18] K. Fall, and J. Pasquale. Exploiting In-Kernel Data Paths to Improve I/O throughput and CPU Availability. In *Proceedings of the USENIX Winter Conference*, 1993.

[19] G. Finn. An integration of network communication with workstation architecture. *Computer Communication Review*, 21(5), October 1991.

[20] R. Fitzgerald, and R. Rashid. The Integration of Virtual Memory Management and Interprocess Communication in Accent. volume 4, 1986.

[21] ATM Forum. ATM user-network interface specification, version 3.1, 1994.

[22] L. Georgiadis, R. Guerin, and A. Parekh. Optimal Multiplexing on Single Link: Delay and Buffer Requirements. In *Proceedings, INFOCOM*, 1994.

[23] S. J. Golestani. A Framing Strategy for Congestion Management. *IEEE Journal on Selected Areas of Comminication*, 9(7), 1991.

[24] S. J. Golestani. Congestion Free Communication in High-Speed Packet Networks. *IEEE Transaction on Comminication*, 32(12), 1991.

[25] S.J. Golestani. A Self-Clocked Fair Queuing Scheme for Broadband Applications. In *Proceedings, INFOCOM*, 1993.

[26] D. Ferrari H. Zhang. Rate Controlled Static Priority Queuing. In *Proceedings, INFOCOM*, 1993.

[27] M. Hayter, and M. Derek. The Desk Area Network. *Operating Systems Review*, 25(4), October 1991.

[28] D. Heyman, A. Tabatabai, and T.V. Lakshman. Statistical Analysis and Simulation Study of Video Teleconferencing Traffic in ATM Networks. *IEEE Journal on Selected Areas in Communications*, 2(1), 1992.

[29] M. Laubach. Classical IP and ARP over ATM. Internet RFC–1577, January 1994.

[30] D. Legall. MPEG - A Video Compression Standard for Multimedia Applications. *Communications of the ACM*, 34(4), 1991.

[31] H. Massalin. Synthesis: An Efficient Implementation of Fundamental Operating System Services. *PhD thesis, Columbia University*, 1992.

[32] J. Mogul, and A. Borg. The Effect of Context Switches on Cache Performance. In *Proceedings of the ASPLOS-IV*, 1991.

[33] John Ousterhout. Why Aren't Operating Systems Getting Faster As Fast As Hardware? In *Proceedings of the USENIX Summer Conference*, 1990.

[34] A. Parekh. A Generalized Processor Sharing Approach to Flow Control In Integrated Services Network. *Ph.D. Thesis, Massachussetts Institute of Technology*, 1992.

[35] A. K. Parekh, and R. G. Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Network: The Single Node Case. *IEEE/ACM Transactions on Networking*, 1(3), 1993.

[36] M. Pasieka, P. Crumley, A. Marks, and A. Infortuna. Distibuted Multimedia: How Can the Necessary Data Rates be Supported. In *Proceedings of the USENIX Summer Conference*, 1991.

[37] J. Pasquale, E. Anderson, and P.K. Muller. Container-Shipping: Operating System Support for I/O Intensive Applications. *IEEE Computer*, 27(3), 1994.

[38] E. Rathgeb. Modeling and Performance Comparison of Policing Mechanisms for ATM Networks. *IEEE Journal on Selected Areas of Communication*, 9(3), 1991.

[39] D. Saha, D. Kandlur, T. Barzilai, Z. Shae, and M. Willebeek-LeMair. A videoconferencing testbed on ATM: Design, implementation, and optimizations. submitted for publication, November 1994.

[40] D. Saha, S. Mukherjee, and S. K. Tripathi. Multi-rate Traffic Shaping and End-to-end Performance Guarantees in ATM Networks. In *Proceedings, International Conference on Network Protocols*, 1994.

[41] M. D. Schroeder, and M. Burrows. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 8(1), 1989.

[42] Z-Y. Shae, and M-S. Chen. Mixing and playback of JPEG compressed packet videos. In *Proceedings GLOBECOM 92*. IEEE, December 1992.

[43] J. S. Turner. New Directions in Communications. *IEEE Communications*, 24(10), 1986.

[44] S-Y Tzou, and D. P. Anderson. The Performance of Message Passing Using Restricted Virtual Memory Remapping. *Software-Practice and Experience*, 21, 1991.

[45] The USENIX Association and O'Reilly & Associates, Inc., 103 Morris Street, Suite A, Sebastopol, CA 94572. *4.4BSD Programmer's Reference Manual*, April 1994.

[46] G. K. Wallace. The JPEG Still Picture Compression Standard. *Communications of the ACM*, 34, 1991.

[47] S. Wray, T. Glauert, and A. Hopper. Networked Multimedia: The Medusa Environment. *IEEE Multimedia*, 1(4), 1994.

[48] L. Zhang. Virtual Clock: A New Traffic Control Algorith for Packet Switching Networks. In *Proceedings, SIGCOMM*, 1990.