STRUCTURING HOST COMMUNICATION SOFTWARE FOR QUALITY OF SERVICE GUARANTEES

by

Ashish Mehra

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1997

Doctoral Committee:
Professor Kang G. Shin, Chair
Associate Professor Farnam Jahanian
Assistant Professor Sugih Jamin
Dr. Dilip Kandlur
Professor Toby Teorey
Assistant Professor Kimberly Wasserman

To my mother
and
the precious memories of my father.

For me you were the guilding light
The radiance of a humble soul
The glitter of a golden heart
My precious jewel that fate stole.

Nay, the jewel's there I realize
Studded on the canvas of time
The life you sketched now emits a glow
Illuminating the past we left behind.

ACKNOWLEDGEMENTS

I write this dissertation with a heavy heart, for it marks the culmination of a crucial phase in my career and life. Much has happened during the years that I have been in graduate school. Besides experiencing the struggle faced by many doctoral students, I had to endure the tragic and untimely loss of my father, Late Dr. Satish Narain Mehra, who departed for his heavenly abode on May 30, 1994. A humble and honest man, he was and remains my deepest inspiration. It was my heartfelt desire to have him by my side when I graduate; now that desire must remain a dream.

This dissertation is my tribute to him, fond memories of our togetherness, his unflinching faith in me, and his never-ending pursuit of excellence. Wherever you are, Papa, may your soul rest in eternal peace. This dissertation is also a tribute to my mother, Mrs. Uma Mehra, who has endured this loss with great strength, and to my sister Bharti Kakkar and my wife Neelu Mehra, for their exemplary courage in weathering the shock of losing Papa when he suddenly breathed his last.

It was my advisor, Prof. Kang Shin, who seven years ago gave me the opportunity to attend the University of Michigan as a member of RTCL, and I will always be indebted to him for that. I owe my deepest gratitude to him for his enormous patience towards me while I selected a dissertation topic, and for the complete freedom he gave me to pursue research ideas and take them to fruition. His constant encouragement, support and guidance has played a big role in this dissertation.

I would like to express my sincere thanks to Dr. Dilip Kandlur for serving on my committee and for giving me the opportunity to spend two summers with his group at the IBM T. J. Watson Research Center. I learnt a great deal during those summers and truly enjoyed my interaction with researchers at Watson. I would like to thank Prof. Toby Teorey, Prof. Farnam Jahanian, Prof. Sugih Jamin, and Prof. Kimberly Wasserman for agreeing to serve on my committee despite a tight schedule. A special note of thanks to Prof. Farnam Jahanian for being a friend and colleague all these years. I will always cherish your friendship, collaboration, and sense of humor. I gratefully acknowledge the support provided for my study by Office of Naval Research, National Science Foundation, and Defense Advanced Research Projects Agency.

A number of individuals have contributed towards this dissertation in one way or the other. My collaboration with Atri Indiresan laid the foundation for the ideas explored in this dissertation. Since then I have collaborated with Anees Shaikh, Tarek Abdelzaher, and Zhiqun Wang and truly benefitted from this interaction. It would be difficult to find a more caring and conscientious group of individuals who are always so willing to help, learn, and accommodate. Special thanks to Tarek and Zhiqun for their help during the final stages of my dissertation work. I have also had the pleasure of working closely with Tsipora Barzilai and Debanjan Saha at the IBM T. J. Watson Research Center, and I look forward to continuing that collaboration.

Throughout graduate school I have been fortunate to have had very considerate and supportive officemates who often played a crucial role as friends and peer advisors. These include Tom Tsukada, Nigel Hinds, Chao-ju Hou, Jim Dolter, Stuart Daniel, Frank Lei Zhou, Jennifer Rexford, Scott Dawson, Todd Mitton, Wu-chang Feng, Tarek Abdelzaher, and Anees Shaikh. Special thanks to Jim Dolter for sharing with me his expertise and experience in software and hardware design, and to Jennifer Rexford for being a wonderful colleague and close friend all these years. Nothing is more important in graduate school than a hot steaming cup of gourmet coffee first thing in the morning. Accordingly, my most caffeinated gratitude to Scott Dawson for playing the role of "ultimate provider" to perfection. I will surely miss his friendship, technical expertise, and our frequent discussions on anything that came to mind after a sip of gourmet coffee. I will also miss my dear "dost" (friend) Anees Shaikh for the good times spent cooking delicious Indian dishes and for patiently listening to my distorted renditions of popular Indian and Western songs. Graduate work was made that much more enjoyable and hassle-free by the friendship and affection of Beverly J. Monaghan, the RTCL administrative assistant. I will miss her sense of humor and affectionate laughter.

My long stay in Ann Arbor was enriched greatly by wonderful friends, some who have since moved on and some whom I am leaving behind. I have numerous fond memories of time spent with Santanu Paul, Raghu Mani, Raja Sengupta, Amit Misra, Aditi Dubey, Bishnu Gogoi, Arindam Chatterjee and Krishnendu Chakraborty. For the past three years or so, I have been able to actively pursue my interest in music in the company of several friends. Mrs. Shubhangi Deshpande taught me Hindustani classical vocal and participated in some memorable musical evenings. I will always be indebted to her for exposing me to a whole world of creative beauty and graceful precision. My heartfelt thanks to Sushila Subramanian and Kavita Goverdhanam for participating in the weekly music sessions when we could forget everything else and indulge in music, and for the few musical performances we gave together. A special note of thanks to Maninder Singh for always finding time for these performances; for me his amazing skills with the Tabla were both a source of musical pleasure as well as inspiration. My indulgence in Indian music was further facilitated by Pramila and Sushil Birla, who invited me to numerous musical gatherings and dinners.

In recent times my circle of friends has grown to include some very special and caring couples: Sushila Subramanian and Dan Kiskis, Kavita Goverdhanam and Uday Nandan, Janani Janakiraman and Karthik Ramamoorthy, Rohini and Atri Indiresan, Zakia and Anees Shaikh, Seemeen Naushaba and Mubashir Mohiuddin, Deepika and Chetan Ahuja, and Neeta and Saryu Goel. A special note of appreciation for Seemeen and Mubashir, and Neeta and Saryu, for their genuine affection and considerable help during the time that I was busy writing the dissertation. Needless to say, leaving behind such nice friends is perhaps the saddest part of moving on to life after graduate school.

Finally, no measure of gratitude and love is enough towards my wife, Neelu, for her friendship and patience while I struggled to wrap up graduate school. Her support, care and encouragement gave me much-needed strength and hope during a demanding phase. Without her this little achievement of mine would not be as meaningful.

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGEMENTS iii

LIST OF TABLES ix

LIST OF FIGURES x

CHAPTERS

1 INTRODUCTION 1
  1.1 QoS Issues in End-to-End Communication 3
    1.1.1 General Scenario 3
    1.1.2 Integrated Services on the Internet 6
  1.2 Dissertation Focus and Problem Statement 7
  1.3 Primary Contributions 11
  1.4 Dissertation Overview 14

2 HOST SUPPORT FOR QUALITY OF SERVICE 16
  2.1 Application QoS Requirements 17
  2.2 Communication Subsystem Overview 19
  2.3 Factors Affecting Performance 21
    2.3.1 API Semantics 22
    2.3.2 Protocol Stack Execution 23
    2.3.3 Multiprogramming and Network Load 25
  2.4 Efficient Protocol Architectures and Implementations 27
    2.4.1 Performance optimizations for efficient data transfer 28
    2.4.2 Network Interface Design 31
    2.4.3 Parallel Protocol Implementations 33
  2.5 QoS-Sensitive Communication and Computation 35
    2.5.1 Dynamic QoS Negotiation and Adaptation 36
    2.5.2 Communication Architectures for QoS 37
    2.5.3 Multimedia/Real-Time Operating Systems 40

3 A QOS-SENSITIVE COMMUNICATION SUBSYSTEM 43
  3.1 Introduction 43
  3.2 Architectural Requirements for Guaranteed-QoS Communication . 46
    3.2.1 Software Structure for QoS-Sensitive Data Transport 48

3.2.2 Real-Time Channels: A Model for Guaranteed-QoS Communication . . . 50
3.2.3 Performance Related Considerations . . . 52
3.3 QoS-Sensitive Communication Subsystem Architecture . . . 53
3.3.1 Salient Features . . . 55
3.3.2 Accounting for CPU Preemption Delays and Overheads . . 59
3.4 Prototype Implementation . . . 62
3.4.1 Architectural Configuration . . . 62
3.4.2 Realizing a QoS-Sensitive Architecture . . . 63
3.4.3 System Parameterization . . . 66
3.5 Experimental Evaluation . . . 69
3.5.1 Methodology and Metrics . . . 69
3.5.2 Efficacy of the Proposed Architecture . . . 70
3.5.3 Need for Cooperative Preemption . . . 72
3.5.4 Discussion . . . 73
3.6 Summary and Future Work . . . 74

4 ADMISSION CONTROL EXTENSIONS FOR END HOSTS . . . 77
4.1 Introduction . . . 77
4.2 Managing CPU and Link Bandwidth . . . 79
4.2.1 Implementation Issues . . . 80
4.2.2 Performance Implications . . . 84
4.3 Worst-Case Service and Wait Times . . . 86
4.3.1 Estimating Service Time . . . 87
4.3.2 Estimating Wait Time . . . 90
4.3.3 Experimental Validation . . . 91
4.4 Channel Admissibility . . . 92
4.4.1 Channel Admissibility in O1 . . . 93
4.4.2 Channel Admissibility in O2 . . . 94
4.5 Admission Control Extensions for Receiving Hosts . . . 95
4.5.1 Reception Issues and Assumptions . . . 95
4.5.2 Pure Reception . . . 98
4.5.3 Simultaneous Send and Receive . . . 104
4.6 Summary . . . 105

5 GUARANTEED-QoS COMMUNICATION SERVICES . . . 107
5.1 Introduction . . . 107
5.1.1 Goals, Approach and Assumptions . . . 108
5.1.2 Outline . . . 110
5.2 Real-Time Communication Service . . . 111
5.2.1 Goals and Paradigm . . . 111
5.2.2 Service Architecture . . . 111
5.3 Service Components and Their Interaction . . . 113
5.3.1 Service Invocation via RTC API . . . 113
5.3.2 RTCOP-based Signalling and Resource Reservation . . . 117
5.3.3 CLIPS-based Resource Scheduling for Data Transfer . . . 120
5.4 Prototype Implementation: Environment and Configuration . . . 124
5.4.1 Testbed and Implementation Environment . . . 124

5.4.2 OSF Path Framework: Implications and Extensions 125
5.4.3 Server Configuration: Pros and Cons 127
5.4.4 CORDS-based Service Protocol Stack 129
5.5 Realization of the Service Architecture 130
5.5.1 Service Library: librtc 131
5.5.2 RTC API Anchor Implementation 132
5.5.3 RTCOP and RTROUTER Implementation 135
5.5.4 CLIPS Implementation 138
5.6 System Profiling and Parameterization 139
5.6.1 Profiling the RTC API Anchor 140
5.6.2 Profiling the Protocol Stack 142
5.6.3 Profiling Link Input/Output 144
5.7 Accounting for API Overheads and Threads 145
5.7.1 Admission Control Enhancements 145
5.7.2 Architectural Enhancements 147
5.8 Experimental Evaluation 149
5.8.1 Traffic enforcement 149
5.8.2 Traffic isolation 151
5.9 Summary 155

6 SELF-PARAMETERIZING PROTOCOL STACKS 158
6.1 Introduction 158
6.2 Motivation and Problem Statement 160
6.2.1 Nature of System Parameters, Costs and Overheads 160
6.2.2 Infeasibility of Detailed Manual Profiling 161
6.2.3 Automated Approach to Performance Profiling 162
6.3 Related Work 163
6.3.1 Protocol Stack Performance 163
6.3.2 Protocol Benchmarking 164
6.3.3 Operating System Performance and Resource Monitoring . . 165
6.4 Design Approach for Self-Parameterization 166
6.4.1 Overall Architecture 166
6.4.2 Structuring the Admission Control Module 169
6.5 Minimizing Perturbation 170
6.5.1 Placement and Control of Profile Points 170
6.5.2 Deferred Sample Processing and Parameter Tuning 173
6.6 Implementation and Evaluation 174
6.6.1 Modules, Parameters and Sample Collection 174
6.6.2 Message Generation 175
6.6.3 Parameter Tuning and Update 176
6.6.4 Experimental Results 177
6.7 Summary and Future Work 180

7 QOS SUPPORT IN TCP/IP PROTOCOL STACKS 182
7.1 Introduction 182
7.2 RSVP and Integrated Services: An Overview 185
7.2.1 RSVP: An End-to-End View 185
7.2.2 Service Classes 186

7.3 Architectural Overview and QoS Components 188
7.3.1 Control Functions 189
7.3.2 Data Transfer 190
7.4 Efficacy of the QoS Architecture 193
7.4.1 User-Level Performance Measurements 193
7.5 QoS Component Overheads 197
7.5.1 Kernel Instrumentation and Measurement Methodology . . 199
7.5.2 Component Overheads 200
7.5.3 Effective Overhead 201
7.6 Performance Implications 202
7.6.1 Accommodating variations in the shaping latency 203
7.6.2 QoS-Sensitive CPU Scheduling 205
7.7 Summary and Future Work 206

8 INTEGRATION WITH HOST OPERATING SYSTEM 208
8.1 Protocol Processing Architectures 208
8.2 Realizing Application-Level QoS Guarantees 212
8.3 Summary 215

9 CONCLUSIONS AND FUTURE WORK 216
9.1 Primary Contributions 216
9.2 Future Research Avenues 218

BIBLIOGRAPHY 220

LIST OF TABLES

Table

3.1 Routines constituting the real-time channel API. 62
3.2 Available policies in the prototype implementation. 67
3.3 Workload used for the evaluation. 70
4.1 Important system parameters. 87
5.1 Routines comprising `RTC API`. 114
5.2 `RTCOP` functional requirements. 117
5.3 `CLIPS` functional requirements. 120
5.4 Data transfer overheads in `RTC API` anchor (in μs). 140
5.5 Application-level send and receive latencies (in μs). 141
5.6 Protocol stack latencies for send and receive paths (in μs). 143
5.7 Link scheduler packet transmission latencies (in μs). 144
5.8 `CORDS` device input thread overhead (in μs). 145
6.1 Communication resources and their cost components. 160
6.2 Modules and parameters in the prototype implementation. 174
6.3 API routines exported by the self parameterization (SP) core. 176
6.4 Anchor: comparison of manual and self parameterization (in μs). 177
6.5 Protocol stack: manual and self parameterization (in μs). 178
6.6 Link driver: comparison of manual and self parameterization (in μs). 179
7.1 Control path latencies. 195
7.2 Data path performance. 196
7.3 Overheads of different QoS components (in μs). 201
7.4 Data path latencies on tapti (in μs). 202
8.1 Protocol processing architectures. 209

LIST OF FIGURES

Figure

1.1 End-to-end communication across multiple nodes. 4
1.2 Focus of dissertation research. 7
1.3 Communication subsystem at end hosts. 8
2.1 Communication subsystem as a pipeline. 20
2.2 Interaction between protocols and resource management functions 21
3.1 Desired overall software architecture. 48
3.2 Software structure for QoS-sensitive data transmission. 49
3.3 Proposed communication subsystem architecture. 54
3.4 Channel state maintained at host. 55
3.5 Execution profile of a channel handler. 56
3.6 Important system parameters in the proposed architecture. 60
3.7 Real-time communication protocol stack (x-kernel). 63
3.8 EDF CPU scheduler layered above native x-kernel scheduler. 65
3.9 Processing done by the link scheduler. 66
3.10 Implementation environment. 68
3.11 Maintenance of QoS guarantees when traffic specifications are honored. . . 71
3.12 Maintenance of QoS guarantees under violation of R_{max}. 72
3.13 Violation of QoS guarantees with cooperative preemption disabled. 73
4.1 Throughput as a function of packets processed between preemption points (packet size 4 KB). 83
4.2 Throughput as a function of packet size and link speed. 85
4.3 Protocol processing and link transmission overlap in O1. 88
4.4 Protocol processing and link transmission overlap in O2. 89
4.5 Comparison of measured (m) and predicted (p) throughput ($\mathcal{P}$ = 4). 91
4.6 Effect of $\mathcal{P}$ and $\mathcal{S}$ on channel admissibility in O1. 93
4.7 Effect of $\mathcal{P}$ and $\mathcal{S}$ on channel admissibility in O2. 94
4.8 Link reception and protocol processing overlap in interrupt mode input. . . 100
4.9 Link reception and protocol processing overlap in polled mode input. 103
5.1 Real-time communication service architecture. 112
5.2 `RTCOP` internal structure and interfaces. 119
5.3 `CLIPS` internal structure and interfaces. 121
5.4 Experimental testbed. 125
5.5 Service implementation as `CORDS` server. 127
5.6 `CORDS`-based service protocol stack. 129

5.7 Traffic enforcement on a single real-time channel. 150
5.8 Traffic isolation between two real-time channels. 152
5.9 Traffic isolation between real-time and best-effort traffic. 153
5.10 Traffic isolation and unused capacity utilization. 154
5.11 ARMADA middleware and real-time communication services. 156
6.1 Architecture for self-parameterizing protocol stacks. 167
6.2 On-line construction of system parameter database. 168
6.3 Internal structure of admission control procedure. 169
6.4 Prologue and epilogue processing for profile sample generation. 171
7.1 PATH and RESV messages in RSVP. 187
7.2 Protocol stack architecture and QoS extensions. 189
7.3 Best effort and QoS data paths. 191
7.4 Data path through QOSMGR. 192
7.5 Experimental testbed for prototype implementation. 194
7.6 Timeline of events and associated overheads during shaping. 198
8.1 Integrated QoS-sensitive computation and communication subsystems. . . . 213

ABSTRACT

STRUCTURING HOST COMMUNICATION SOFTWARE
FOR QUALITY OF SERVICE GUARANTEES

by
Ashish Mehra

Chair: Kang G. Shin

This dissertation addresses several issues involved in structuring communication software at end hosts to provide per-connection quality of service (QoS) guarantees. Our primary thrust is on realizing *deterministic* QoS guarantees in an *experimental* setting.

We design a novel *QoS-sensitive communication subsystem architecture* that provides components and mechanisms to manage communication resources in a QoS-sensitive fashion. This architecture is based on three key design principles: maintenance of QoS guarantees, overload protection via per-connection traffic enforcement, and fairness to best-effort traffic. We demonstrate the efficacy of the architecture via experiments using our x-kernel-based prototype implementation.

We develop *admission control extensions* to capture important implementation-related aspects not considered by theoretical resource management policies proposed in the literature. Communication performance is significantly affected by implementation-related aspects such as preemption overheads and constraints, simultaneous management of CPU and link bandwidth, link scheduling paradigms at sending hosts, and packet input mechanisms at receiving hosts.

We build a full-fledged *guaranteed-QoS communication service* on a contemporary microkernel operating system using a new service architecture integrating a QoS-aware application programming interface, a reliable signalling and resource reservation protocol, and QoS-sensitive data transmission and reception. We develop architectural and admission control enhancements that capture new overheads and constraints imposed by a microkernel server configuration. The service is implemented and evaluated on Pentium-based PCs running OSF MK 7.2 and communicating across switched Ethernet.

To enhance the portability of guaranteed-QoS communication services, we design *self-parameterizing protocol stacks* that profile and parameterize themselves appropriately during data transfer. We experimentally evaluate a self-parameterizing guaranteed-QoS communication service to demonstrate the feasibility of our approach. Self-parameterizing protocol stacks are a natural way to design portable QoS-sensitive communication software.

To provide "better than best-effort" Internet connectivity, the *performance impact of supporting integrated services in TCP/IP protocol stacks* must be assessed. We have designed, implemented, and evaluated an RSVP-based QoS architecture for TCP/IP protocol stacks supporting an integrated services Internet. Using detailed kernel-based profiling of RS/6000-based servers running AIX 4.2, we explore the performance impact of QoS support in TCP/IP protocol stacks.

We conclude by outlining strategies for integrating QoS-sensitive communication subsystems with QoS-sensitive application scheduling policies within host operating systems.

CHAPTER 1

INTRODUCTION

The advent of high-speed networks and the WWW has generated an increasing demand for a new class of distributed applications that require certain quality-of-service (QoS) guarantees on end-to-end communication across the underlying network. In general, these QoS guarantees may be specified in terms of parameters such as the end-to-end delay, delay jitter, and bandwidth delivered on each connection; additional requirements regarding packet loss and in-order delivery may also be specified. Examples of such applications include distributed multimedia applications (e.g., video conferencing, video-on-demand, digital libraries) and distributed real-time command/control systems. The need for QoS guarantees on communication is also evidenced by the rapid proliferation of on-line multimedia content on the WWW, the shortcomings of the "best-effort" service provided by today's Internet, and the efforts by the Internet Engineering Task Force (IETF) to address these shortcomings by developing protocols and standards for Integrated Services on the Internet [20, 22, 65].

To support such applications, all hardware and software components involved in transferring application data from one host to another across the network must be designed to be *QoS-sensitive*, i.e., to provide QoS guarantees. Such components include the end host operating system that controls application execution and manages host resources, the network adapter that interfaces the host to the network, and network elements such as routers that manage network resources to forward traffic through the network. A QoS-sensitive operating system, for example, would ensure that applications exchanging QoS-sensitive data are scheduled for execution so as to satisfy the desired end-to-end QoS guarantees on communication. There have been numerous proposals for QoS-sensitive provisioning of network resources such as transmission bandwidth and buffers [8, 188], and host computation resources such as CPU cycles [71, 165, 179]. However, not much attention has been paid to the *interface* between the applications and the network, i.e., the communication subsystem at end hosts, for provision of QoS guarantees. While communication subsystems have been extensively studied to improve the performance of traditional best-effort traffic, several significant design and implementation issues must be considered when handling QoS-sensitive traffic.

This dissertation focuses on the host communication subsystem and examines various issues involved in structuring communication software at *end hosts* to provide per-connection QoS guarantees. For the most part it is assumed that the network provides appropriate support to establish and maintain guaranteed-QoS connections. That is, we focus on QoS-sensitive allocation and management of communication resources at end hosts. The primary thrust of the dissertation is on identifying and addressing key issues involved in realizing *deterministic* QoS guarantees on real computer systems. These issues include bridging the gap between theory and practice in resource management for guaranteed-QoS communication to account for implementation overheads and constraints, exploring mechanisms that facilitate development of portable QoS-sensitive communication software, and assessing the nature and performance impact of architectural enhancements needed in traditional TCP/IP protocol stacks to support an Integrated Services Internet. Of paramount concern is the integration of a QoS-sensitive host communication subsystem with the host operating system to provide application-level QoS guarantees.

The contributions made by this dissertation are applicable to a variety of host and network architectures for provision of QoS. Further, the issues identified and addressed, as well as the insights gained, are of significance to other QoS models, approaches to QoS provisioning, and resource reservation strategies. While the dissertation focuses on provision of QoS in unicast communication, the results and approach adopted are relevant to provision of QoS in multicast communication as well. Our research methodology is primarily geared towards experimental realization of QoS-sensitive communication subsystems. Thus, we have implemented and evaluated all the architectural solutions outlined in this dissertation. As such, our contributions are also of significance to designers and practitioners of operating systems and networks.

The rest of this chapter is organized as follows. The next section gives a brief overview of QoS in end-to-end communication. Subsequently, we identify the primary research focus of this dissertation and relate it to the overall goal of providing QoS in end-to-end communication. The following section summarizes the primary contributions made by this dissertation. Finally, we conclude the chapter with a brief organizational description of the dissertation.

1.1 QoS Issues in End-to-End Communication

Provision of QoS in end-to-end communication is complex, requiring significant new functionality to be provided at end hosts and the network elements forming the communication fabric. It is important, therefore, to develop the necessary taxonomy and put the primary focus of this dissertation in context. We first outline the general issues involved in provision of QoS in end-to-end communication, followed by a brief description of related efforts towards realization of an Integrated Services Internet.

1.1.1 General Scenario

Figure 1.1 depicts a scenario in which an application on node S (source or sender) sends data to an application at node D (destination or receiver). S sends data to D through the network via one or more intermediate nodes denoted by I; S and D are referred to as *end hosts* while all the intermediate nodes are referred to as *routers*. As the data generated at S traverses the network, it may get buffered and queued at host S, at each of the network routers, and at host D before being delivered to the receiving application. This buffering and queueing occurs due to contention for communication resources such as CPU and link bandwidth. For example, at intermediate nodes traffic from several other nodes may need to be simultaneously forwarded to one or more nodes.

The end-to-end QoS delivered to the application is a function of the extent of resource contention at S, D, and the intermediate nodes I, which is in turn determined by the nature of buffering, queueing, and processing of the application's data packets. The delivered QoS can be measured in terms of one or more *QoS parameters* such as the end-to-end packet delay, delay jitter, and throughput. Uncontrolled resource contention may result in unbounded packet delays, high delay jitter (i.e., significant variability in packet delay), insufficient availability of bandwidth, and even packet loss due to buffer overflows, especially under high load. An example of uncontrolled resource contention is traffic handling on a first-

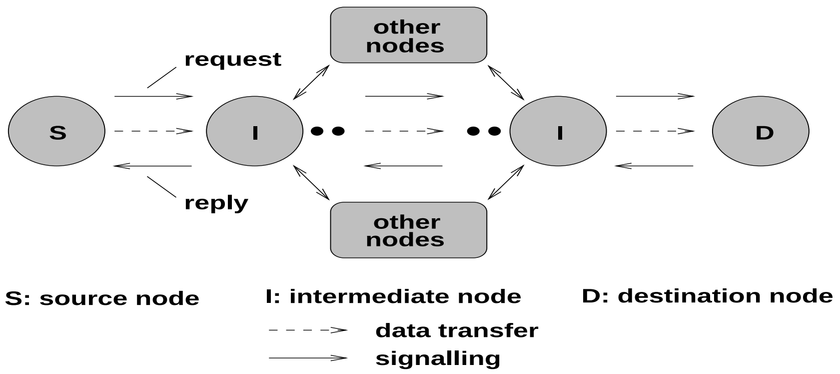


Figure 1.1: End-to-end communication across multiple nodes.

in-first-out basis, in which traffic from all nodes is multiplexed together in an uncontrolled fashion.

In general, provision of QoS guarantees on end-to-end communication requires appropriate support in the network routers and end hosts to prevent uncontrolled resource contention and maintain per-connection QoS guarantees [8]. The application provides a per-connection *QoS specification*, which defines its QoS requirements for each QoS parameter, and also provides a *traffic specification*, which defines its traffic generation characteristics on that connection against which the QoS guarantees are to be given. QoS guarantees can be *deterministic* such that no QoS violations are permitted even in the worst case, or *statistical* (i.e., probabilistic) such that QoS violations may be permitted to a limited extent in the worst case, as long as the application conforms to its traffic specification. In the following, we assume a deterministic QoS specification, although much of the discussion is also applicable to statistical QoS guarantees.

The application's QoS and traffic specifications must be communicated to each node involved in end-to-end data transfer so that each node can allocate sufficient resources for the connection. This communication is performed via end-to-end reliable *signalling*, which can be either sender-initiated or receiver-initiated. Before signalling can commence, however, the communication subsystem must select the best route from source to destination via *QoS routing*. As shown in Figure 1.1 (depicting sender-initiated signalling), signalling requests are sent from S to D via the intermediate nodes; replies are delivered in the reverse direction from D to S.

Each node examines the signalling request and performs *authentication* and policy control to verify the request and *admission control* to check availability of communication resources (such as CPU cycles, connection state and buffers, and link bandwidth) for the new connection. If sufficient resources are available at a node, it performs *resource reservation* to set aside the necessary communication resources for this connection, and forwards the request to the next node along the chosen route. If D accepts the request, it returns a reply with a success indication; this reply is propagated back to S through the intermediate nodes after committing the reserved resources for this connection. If sufficient resources are unavailable at a node, the node returns a reply with a failure indication, which is also propagated back to S after performing *resource reclamation* on any resources reserved at intermediate nodes.

If the connection is successfully established, data sent by S to D on this connection will be serviced according to the corresponding QoS guarantees. Consider the handling of traffic arriving at an intermediate node. To determine how this traffic should be serviced, the node performs *traffic classification* using information contained in packet headers to determine the associated connection. Since the QoS guarantees are given relative to the connection's traffic specification, the node performs *traffic enforcement* via *policing* and/or *shaping* to enforce the traffic specification. While policing ascertains whether the traffic is *conformant* or not, shaping buffers non-conformant traffic until conformance as per the traffic specification. Each node also implements a QoS-sensitive *queueing policy* which determines the nature of queueing at the node, and a *service discipline* [8, 188] which determines the order in which packets are scheduled for service, e.g., processing and transmission to the next node along the chosen route. Example service disciplines include Weighted Fair Queueing [50], also known as Packet-by-Packet Generalized Processor Sharing [144], and Rate-Controlled Static-Priority Queueing [187]. In addition to servicing (QoS-sensitive) traffic according to the associated QoS guarantees, the service discipline also determines the treatment of best-effort traffic at the node. Note that the service discipline may also determine the admission control procedure employed at the node. When done, the sender closes the connection so that resources allocated to this connection can be reclaimed for reuse.

The above description outlines an example scenario for QoS in unicast end-to-end communication using sender-initiated signalling for resource reservation. A similar scenario can be constructed for receiver-initiated signalling of unicast connections. Provision of QoS in end-to-end multicast communication adds significantly higher complexity during QoS routing, signalling, as well as data transfer. While we haven't mentioned it explicitly, nodes may also have to provide support for fault-tolerance, i.e., reclaim resources and/or re-route traffic in the presence of processor and link failures. Finally, instead of deterministic or probabilistic guarantees, even looser forms of QoS may be provisioned for applications that can adapt to variations in the delivered QoS. Proposals for predictive (or best-effort) real-time communication, such as FIFO+ [39] and Hop-Laxity [156], fall in this category.

1.1.2 Integrated Services on the Internet

Significant efforts are being made by the IETF to enhance the service model of the Internet to support integrated services for voice, video, and data transport [22, 39]. Similar to the general scenario described above, to support integrated services on the Internet, the network routers and end hosts need to be enhanced to perform traffic classification on a per-flow basis, create and maintain flow-specific soft reservation states, and handle data packets from different flows in accordance with their QoS requirements [22]. Towards that end, the IETF is developing a set of protocols and standards for integrated services [20, 65, 149, 189].

In the IETF's vision, applications request and reserve resources, both in the network and at the attached hosts (clients or servers) using an end-to-end receiver-initiated Resource ReSerVation Protocol (RSVP) [23, 189]. Resource management is performed via per-flow traffic shaping and scheduling for various classes of service [22], such as guaranteed service [159] that provides guaranteed delay, and controlled load service [183] that has more relaxed QoS requirements. The guaranteed service is intended for applications requiring firm guarantees on loss-less on-time datagram delivery. The controlled load service, on the other hand, is designed for the broad class of adaptive real-time applications (such as vic, vat, nevot, etc.), in active use on the Internet today, that are sensitive to network overload conditions.

In addition to the above services, looser forms of QoS may also be provided via services such as predictive service [85], which performs measurement-based admission control. In the context of integrated services, the expected QoS requirements of applications and issues involved in sharing link bandwidth across multiple classes of traffic are explored in [65, 158]. Much support being provided on the Internet is geared towards realizing efficient multicast communication and accommodating the heterogeneity of receivers. RSVP, in particular, has been designed to support multicast communication between heterogeneous receivers. In contrast to RSVP, which initiates reservation setup at the receiver, an alternative approach

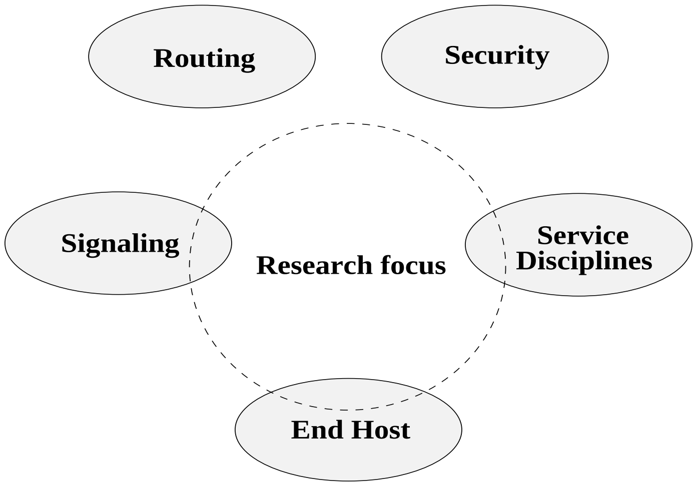


Figure 1.2: Focus of dissertation research.

to signalling is adopted by the ST-II protocol [49], which initiates reservation setup at the sender. The issues involved in providing QoS support in IP-over-ATM networks [149] are also being explored [20].

1.2 Dissertation Focus and Problem Statement

As mentioned, the primary focus of this dissertation is host communication subsystem design to request and obtain per-connection QoS, assuming availability of appropriate network support. As Figure 1.2 illustrates, we are concerned with signaling and resource management services at end hosts, for a *given* service discipline. In particular, we are interested in the issues and problems encountered *in practice* when managing host communication resources according to the relative importance of the connections requesting service. These problems arise in the context of unicast as well as multicast communication, for both sender-initiated and receiver-initiated signalling.

Thus, provision of QoS in the network, including issues in QoS routing, and authentication are beyond the scope of this dissertation. We do not propose a new service discipline either; instead, we focus on providing new architectural components and mechanisms to realize an existing service discipline on actual hardware and software systems. Towards that end, besides run-time resource management, we consider the issues involved in supporting signaling, admission control, and resource reservation services within host communication subsystems.

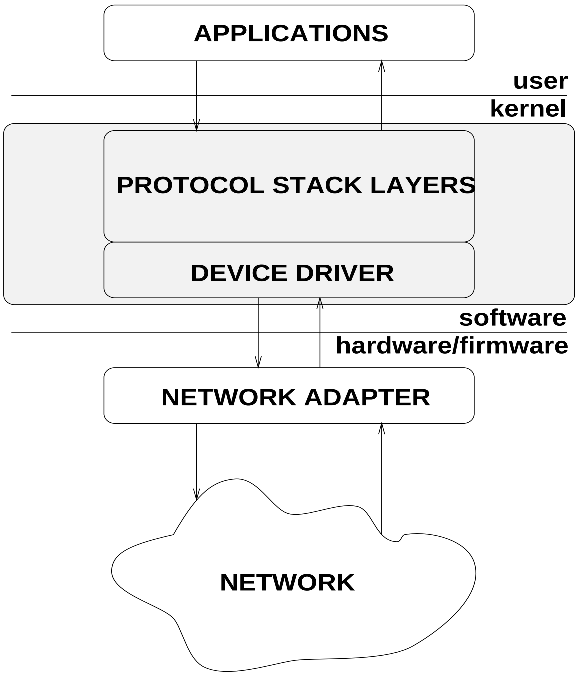


Figure 1.3: Communication subsystem at end hosts.

Figure 1.3 depicts the typical communication subsystem at end hosts. Applications running in user space interact with the protocol stack layers residing in the kernel, or at user level, to transmit and receive data. The protocol layers interact with each other to perform protocol processing (such as header encapsulation and removal, fragmentation and reassembly, and various other services) on the application's data. The device driver, which is the bottom layer in the protocol stack, interacts with the external network through the network adapter residing in the host. While Figure 1.3 illustrates a kernel-level stack, similar interactions occur even when the protocol stack executes in user space. For a sending host, communication resources include CPU bandwidth for protocol processing (e.g., header encapsulation and fragmentation), link bandwidth for packet transmissions, and buffer space for outgoing traffic. Similarly, for a receiving host communication resources include protocol processing bandwidth (e.g., header removal and reassembly), reception bandwidth of the network adapter, and buffer space for incoming traffic.

Several key problems must be addressed in order to enable QoS-sensitive management of communication resources in practice. The first of these is the determination of the appropriate architectural components required within the communication subsystem to provide and maintain QoS guarantees. In addition to these architectural components, the gap between theory and practice of communication resource management must be bridged by capturing important implementation-related aspects that affect communication performance. The architectural tradeoffs involved in building a full-fledged guaranteed-QoS communication service for use by applications must be explored and resolved.

A significant concern in realizing such services in practice is the development of mechanisms that enhance the portability of the constituent QoS-sensitive communication subsystem software. The ubiquity of hosts attached to the Internet makes it important to identify the nature and performance impact of designing host communication subsystems to support integrated services on the Internet. Finally, realization of application-level QoS guarantees necessitates that approaches to integrate a QoS-sensitive communication subsystem within the host operating system be explored. We expand on each of the above problems below.

◦ *Architectural components for QoS Guarantees:* Most existing communication subsystems are designed with resource management policies geared towards statistical fairness and/or time-sharing. Such policies can introduce excessive interference between different connections, thus degrading the delivered QoS on individual connections. The unpredictability introduced and excessive delays due to interference between different connections may even result in QoS violations. Accordingly, new architectural components are needed within the communication subsystem to eliminate such performance degradation via: (i) conformance of QoS guarantees, (ii) overload protection via per-connection traffic enforcement, and (iii) fairness to best-effort traffic. These requirements together ensure that per-connection QoS guarantees are maintained even as the number of active connections grows or per-connection traffic load increases.

◦ *Gap between theory and practice in communication resource management:* Satisfying the above requirements requires QoS-sensitive management of communication resources. This in turn necessitates admission control and resource scheduling policies to ensure that each connection receives, at least, its required QoS. These policies are typically formulated using idealized models of the resources being managed. For example, it may be assumed that a given resource is immediately preemptible or the cost of preemption is negligible. More importantly, it may be assumed that a required set of resources can be accessed, and hence allocated, independent of one another. However, the above assumptions can be violated when implementing resource management policies, since the performance characteristics of the hardware and software components employed can deviate significantly from those of the idealized resource models. Thus, these policies must be extended to make them useful in practice.

◦ *Guaranteed-QoS communication services on microkernel operating systems:* Microkernel operating systems play an increasing important role in today's PC, workstation, and server markets, as evidenced by the growing popularity of Windows NT [45]. Realization of guaranteed-QoS communication services on contemporary microkernel operating systems presents additional challenges that affect the structuring and performance of communication software. Further, the architectural components and mechanisms developed for QoS guarantees must be appropriately mapped to resource management support already available within the communication subsystem and/or the operating system. Moreover, resource management policies may need to be suitably enhanced to accommodate additional constraints imposed by the operating system or implementation environment available. Insights into such issues can only be developed through actual realization and deployment of guaranteed-QoS services on a contemporary microkernel operating system.

◦ *Portability of QoS-sensitive communication software:* Provision of QoS guarantees is highly platform-specific (i.e., depends on the CPU and network capacities, as well as the operating system, of a platform), especially for deterministic guarantees. Realizing QoS-sensitive communication software requires that the host communication subsystem be parameterized accurately to capture processing costs and overheads that comprise the abstraction of the underlying communication subsystem. The architectural framework and methodology adopted for designing QoS-sensitive communication software should be applicable to a variety of host platforms and networking technologies. This is necessary in order to retarget the admission control procedure and run-time management support to a given host platform and/or networking technology. Accordingly, mechanisms must be developed to facilitate and enhance portability of QoS-sensitive communication software, which is crucial for large-scale cost-effective deployment.

◦ *Performance impact of supporting QoS in TCP/IP stacks:* For wide spread deployment of the service model envisioned by the IETF, it is imperative that the TCP/IP protocol stacks running at Internet hosts be enhanced with appropriate architectural mechanisms to support integrated services. These enhancements must not only preserve the structure of existing sockets-based Unix-like communication subsystems, but also be compatible with the semantics of the sockets API. Even with availability of these architectural enhancements, such a service model will be widely deployed and utilized only if they do not impose significant overheads relative to the traditional best-effort data path. Therefore, it is important to ascertain the potential performance impact of supporting integrated services within the traditional TCP/IP protocol stacks running at Internet hosts.

◦ *Delivering QoS to applications:* For application-level QoS guarantees, resource management within the communication subsystem must be integrated with that for applications. The architectural components, mechanisms, and extensions developed to realize a QoS-sensitive communication subsystem should neither preclude nor be rendered infeasible by such an integration. Accordingly, the issues and tradeoffs involved in realizing such an integration must be explored.

1.3 Primary Contributions

This dissertation makes several key contributions towards the practical realization of QoS-sensitive communication subsystems. The following contributions address each of the problems identified in the preceding section, thus advancing the state of the art in the provisioning of host resources to enable end-to-end QoS.

• ***QoS-sensitive communication subsystem architecture:*** We have designed, implemented, and evaluated a QoS-sensitive communication subsystem architecture for end hosts that satisfies the requirements of providing per-connection QoS guarantees, isolating traffic between different connections, and ensuring fairness to best-effort traffic. While this architecture is applicable to a wide variety of service disciplines for providing QoS guarantees, it is validated by implementing *real-time channels* [63, 92], a paradigm for guaranteed-QoS communication in packet-switched networks proposed by other researchers.

Key features of the architecture include a *process-per-channel* model for protocol processing, QoS-sensitive CPU scheduling of per-channel protocol processing, QoS-sensitive link scheduling of packet transmissions, multiplexing of the CPU amongst channels via *cooperative preemption*, overload protection on each channel via traffic enforcement (policing and shaping), and per-channel buffer management. Our approach in essence decouples protocol processing priority from that of the application.

- ***Admission control extensions for end hosts:*** We develop admission control extensions to account for various implementation-related aspects not previously considered in the literature. These include the overlap between CPU protocol processing and link transmission or reception, the relative speed (capacity) of the CPU and the link, and the incorporation of various implementation overheads (e.g., context switching, cache misses, interrupt handling, packet classification). We also consider two distinct mechanisms for implementing link scheduling of packet transmissions, namely, thread-based and interrupt-based, at sending hosts. Similarly, for receiving hosts we consider two distinct mechanisms for packet input, namely, polled mode and interrupt mode. These extensions are then refined for hosts simultaneously engaged in transmission as well as reception of QoS-sensitive traffic.

The issues of simultaneous management of CPU and link bandwidth for real-time communication are of wide-ranging interest. The above-mentioned extensions are applicable to other proposals for real-time communication and QoS guarantees [8, 188], and to other host and network architectures. In particular, Internet servers running TCP/IP protocol stacks supporting integrated services, especially the guaranteed service class [159], can also benefit from these extensions. Similarly, Internet routers can apply these extensions when incoming packets must be fragmented before forwarding in order to reconcile the different MTUs of the attached networks.

- ***Guaranteed-QoS communication services:*** Based on the above-mentioned architecture and extensions, we have realized a full-fledged guaranteed-QoS communication service on the Mach MK 7.2 operating system from Open Software Foundation (OSF). This has been done in the context of the `ARMADA` (A Real-Time Middleware Architecture for Distributed Applications) project, a collaborative effort between the Real-Time Computing Laboratory at the University of Michigan and Honeywell Technology Center.

The communication subsystem constituting this service features a new service architecture that makes extensive use of OSF's `CORDS` framework [172]. This service architecture integrates three components: an API for specifying and obtaining QoS guarantees, a sender-based reliable end-to-end signalling protocol, and enhanced mechanisms and policies for admission control, resource reservation, and run-time resource management. We develop architectural and admission control enhancements that capture additional constraints imposed by our implementation configuration. The efficacy of these enhancements are evaluated via experiments performed between Pentium-based PCs running OSF Mach MK 7.2 and `CORDS`-based servers implementing our service architecture.

• ***Self-parameterizing protocol stacks:*** As mentioned earlier, it is highly desirable to enhance the portability of QoS-sensitive communication software, which is inherently dependent on the performance characteristics of the underlying hardware and software components. To address this concern, we have designed *self-parameterizing protocol stacks* that are designed with the ability to parameterize themselves appropriately during data transfer.

We retarget our guaranteed-QoS communication service to utilize a self-parameterizing protocol stack, and evaluate our `CORDS`-based implementation to demonstrate the feasibility of our approach. Our design and implementation methodology strives to minimize the overheads and perturbation induced in the data transfer path, while supporting relatively fine-grain performance profiling and system parameterization. Constructing communication subsystems using self-parameterizing protocol stacks is a promising way to design portable QoS-sensitive communication software.

• ***QoS support in TCP/IP protocol stacks:*** In collaboration with researchers at the IBM T. J. Watson Research Center, we have developed an RSVP-based QoS architecture for TCP/IP protocol stacks supporting an integrated services Internet [13]. This architecture represents a major functional enhancement to the traditional sockets-based communication subsystem [108]. One of the key features of this architecture is the transparent accommodation of network interfaces with differing QoS capabilities, ranging from traditional LANs, such as Token Ring, to ATM networks with a high degree of QoS support.

Using our prototype implementation on RS/6000 based servers running AIX 4.2, we demonstrate the efficacy of our QoS architecture in providing the desired QoS to individual connections exchanging data across an ATM network. Via detailed kernel profiling we measure the overheads imposed by various QoS support components provided in form of traffic policing, traffic shaping, and buffer management. Based on these overhead measurements, we derive key implications for the performance impact of adding QoS support components to TCP/IP protocol stacks.

• ***Integration with the host operating system:*** We highlight the issues involved in, and outline strategies for, integration of a QoS-sensitive communication subsystem with operating system resource management policies for application-level QoS. Such an integration must ensure that the protocol processing priority of a connection is derived from the QoS requirements, traffic characteristics, and run-time communication behavior of the application on that connection. We argue that the features and mechanisms provided in our architecture neither preclude nor are rendered invalid with such an integration, and are also applicable to other protocol processing architectures.

1.4 Dissertation Overview

The rest of the dissertation is organized as follows. In the next chapter we present a survey of related work in host support for QoS in computation and communication, and compare and contrast these efforts with our contributions. The related work surveyed includes the nature of application QoS requirements, factors that affect communication subsystem performance, communication subsystem design for high performance networking, and other proposals for host support for QoS-sensitive communication and computation.

Chapter 3 presents the proposed QoS-sensitive communication subsystem architecture for end hosts, motivating and justifying the choice of the various architectural components provided. An x-kernel-based prototype implementation of this architecture in a standalone configuration is described and evaluated to demonstrate the effectiveness with which QoS guarantees are maintained. This architecture and prototype implementation realizes a subset of the admission control extensions described in Chapter 4.

Chapter 4 motivates and develops admission control extensions to bridge the gap between the theory and practice in resource management for real-time communication. We first develop extensions for hosts engaged only in transmission of QoS-sensitive data (e.g., servers). We then develop similar extensions for hosts engaged purely in reception of QoS-sensitive data (e.g., clients). Finally, we develop suitable extensions for hosts simultaneously engaged in QoS-sensitive data transmission and reception.

Chapter 5 explores realization of a guaranteed-QoS communication service on the OSF Mach MK 7.2 microkernel operating system utilizing the `CORDS` framework. We first motivate and describe the service architecture and its components, followed by details of the prototype implementation. We then present results from a detailed profiling of the communication subsystem. Subsequently we present significant modifications to the resource management policies and service architecture to address the performance implications of our implementation configuration. Finally we present results from an experimental evaluation of our prototype implementation.

Chapter 6 presents the design and implementation of self-parameterizing protocol stacks for guaranteed-QoS communication. Drawing upon our experiences with the guaranteed-QoS communication service, we argue that for large-scale cost-effective deployment, it is crucial to develop mechanisms to enhance the portability of QoS-sensitive communication software. We then propose and describe the design of self-parameterizing protocol stacks, and compare and contrast them with related work. The guaranteed-QoS communication service described in Chapter 5 is then retargeted to work with a self-parameterizing protocol stack. Subsequently, we present experimental results demonstrating the feasibility of self-parameterizing protocol stacks.

Chapter 7 explores the performance impact of QoS support in TCP/IP protocol stacks. We begin with a brief overview of RSVP and integrated services on the Internet, followed by a description of our QoS architecture and enhancements to the sockets based communication subsystem. The efficacy of these enhancements are illustrated via user-level experiments on RS/6000-based hosts running AIX 4.2 and communicating across an ATM network. We then report measurements of QoS component overheads obtained via detailed kernel-level profiling, and discuss their performance implications.

Chapter 8 discusses the issues involved in, and possible strategies for, integrating a QoS-sensitive communication subsystem with the rest of the host operating system.

Chapter 9 concludes the dissertation by summarizing its contributions and outlining avenues for further research in designing QoS-sensitive communication subsystems and operating systems, and realizing application-level QoS guarantees.

CHAPTER 2

HOST SUPPORT FOR QUALITY OF SERVICE

In this chapter we highlight the issues involved, and survey related work, in host support for quality of service in communication. By quality of service, we refer to the performance delivered on one or more of the QoS parameters an application considers important. For example, networked interactive applications are often sensitive to data transfer latency, while typical multimedia streaming applications are sensitive to data transfer throughput (or rate). Factors that affect the delivered (application-level) QoS on communication include not only the raw performance of the communication subsystem, but also the policies adopted to schedule the processes/threads that constitute the communicating application.

From an application's perspective (Figure 1.3), the communication subsystem consists of the application programming interface (API) through which applications request and receive communication services, the communication software that provides these services through one or more communication protocols and necessary resource management, the operating system that forms the glue between the communication software and hardware, and the (hardware) network adapters that interface hosts to the communication network. Numerous research efforts have focused on optimizations and design strategies to improve the *average* performance of each of these components, mostly in the context of traditional best-effort traffic, for latency as well as throughput. Besides leveraging such performance optimizations and design approaches, significant additional efforts are necessary to realize "value-added" communication subsystems that can support guaranteed-QoS communication while providing high performance to traditional best-effort traffic.

The policies used to schedule the communicating applications, the typical sources and sinks of data, also has a significant impact on the delivered QoS on communication. Application scheduling determines the degree and nature of resource (i.e., CPU) sharing within the host computation subsystem, and hence the extent of resource contention and the resulting resource acquisition (queueing) delays. This directly affects the generation of network data at a sending host, and consumption of data at a receiving host, which in turn affects the dynamics of the communication subsystem. QoS-sensitive scheduling policies provide greater control over processor allocation, thus minimizing the above-mentioned effects. However, realization of application-level QoS guarantees necessitates that such policies be consistent with the policies and mechanisms employed for QoS-sensitive data handling within the communication subsystem.

In the next section we discuss the nature of QoS requirements imposed by emerging networked multimedia applications. We then present a brief overview of the structure and functionality of a typical host communication subsystem. Next we discuss the factors and tradeoffs that have a significant effect on communication subsystem performance, and hence the QoS delivered to applications. Following this we survey related work in efficient protocol architectures and implementations to improve the average performance of host communication subsystems. The general implications of, and related work in, QoS-sensitive data handling within communication subsystems are outlined next. Finally, we present related work in QoS-sensitive data handling within communication subsystems and operating system support for multimedia and real-time systems.

2.1 Application QoS Requirements

The quality of service required by an application corresponds to that application's sensitivity or tolerance to certain QoS *parameters*. QoS parameters include, for example, the end-to-end communication delay, the delay jitter or variance, the communication throughput, and packet loss. A given application may exhibit varying degrees of tolerance to only a single QoS parameter or a combination of multiple QoS parameters. Networked multimedia applications integrating various media such as audio, video, and text, exhibit a wide range of QoS requirements on communication [62]. Examples of such applications include high-definition television, medical imaging, scientific visualization, full-motion video, multiparty video conferencing, and collaborative workspace [35, 145, 148].

The QoS requirements of multimedia applications differ significantly from those of traditional data transfer applications such as remote login, electronic mail, and file transfer. The latter are primarily loss-sensitive and hence need reliable communication, would perform better with low latency or high throughput, and are largely tolerant to communication delays and jitter. QoS requirements of multimedia applications, on the other hand, reveal a broad spectrum of sensitivity to multiple parameters. For example, full-motion video requires guaranteed high bandwidth which may be highly variable, but may tolerate limited amounts of packet loss; audio, on the other hand, requires low delay jitter and is sensitive to packet loss [145].

Similarly, multiparty video-conferencing and collaborative workspace applications require multicast capabilities with temporal synchronization between the audio and video streams; this necessitates additional *inter-stream* QoS dependencies in addition to the above-mentioned *intra-stream* QoS requirements. In general, one can identify several grades (or levels) of QoS that the network may provide to support emerging applications. Note that since the QoS requirements are specified in terms of end-to-end performance, the sending and receiving hosts must also be cognizant of, and provide support for, these requirements.

The sensitivity of distributed applications to QoS parameters determines the support needed in the communication subsystem for various grades of QoS. Strict *deterministic* QoS guarantees on bandwidth, delay, delay jitter, and packet loss may be necessary for *hard real-time* applications and multimedia applications such as medical imaging. This necessitates new policies and approaches for network design [8] in which messages or packets have deadlines associated with them. A message arriving at its destination after its deadline has expired is practically useless to the application; such messages are considered lost and are discarded. Prior resource reservation with specification of the desired QoS guarantee and traffic characteristics is required to *isolate* the needed resources for these applications. The guaranteed service [159] being standardized by the IETF for an integrated services Internet supports deterministic guarantees on loss-less on-time data delivery.

There are a large class of emerging applications, however, that can adapt to fluctuations in the delivered QoS through additional buffering and/or adjusting their rate of consumption and generation of network data. Such *adaptive* applications, also called *soft real-time*, may be sensitive to QoS parameters such as end-to-end communication delay, but can be designed to be accommodate delay jitter and packet loss. By providing additional buffers and adaptively controlling the rate at which packets are consumed at the receiver, these applications can adapt to existing network conditions [8, 39, 145]. The destination buffers serve to restore the spacing between successive data samples and buffer out variations in network delay. A study on packet video transport [91] illustrated the effectiveness of using network feedback about load conditions to modulate the frame transmission rate of video sources. The feasibility of adapting frame transmission rates to the available network bandwidth was also demonstrated in [86].

For such applications it suffices to provide *statistical* performance bounds on end-to-end communication delay and packet loss, which could also be computed using the measured performance of the network [85]. In general, adaptive applications allow the network to trade off strict QoS guarantees for simpler network design, higher network utilization, and looser QoS guarantees supplemented with additional intelligence and resources at the senders and receivers. The QoS delivered by the communication subsystem directly influences the type and design of applications that can be supported effectively. For example, for adaptive applications that can adjust to existing network load conditions, the delivered QoS may determine the amount of buffering needed for acceptable performance. Recent studies using voice conferencing indicate that the delivered QoS determines the degree and exact nature of adaptivity required in the application [145]. The controlled load service [183] being standardized by the IETF for an integrated services Internet is intended for such adaptive real-time applications.

2.2 Communication Subsystem Overview

Before discussing the factors affecting communication subsystem performance, we present a brief overview of typical host communication subsystems. The communication subsystem can be viewed as a pipeline, comprising several stages formed by the different modules that participate in network communication. Application data is shepherded between the application task and network through this pipeline. Figure 2.1 shows the generic structure of such a communication pipeline.

Application *tasks* generate communication requests by invoking the appropriate procedures in the *application programming interface* (API). On transmission, the API forwards the requests to the *communication software*, possibly after performing some protection checks and/or buffering. The communication software, which includes the communication

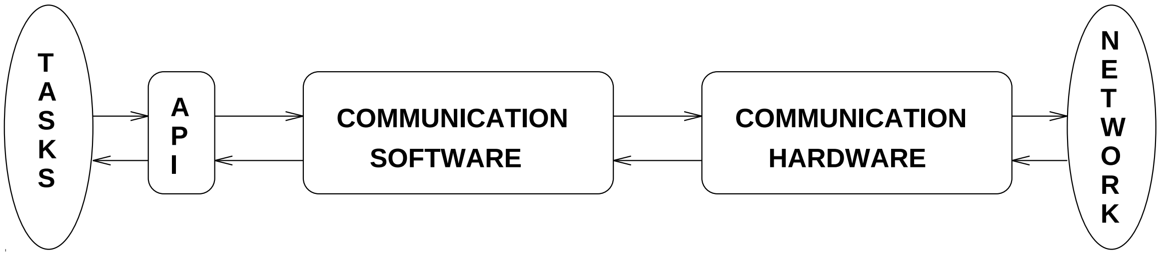


Figure 2.1: Communication subsystem as a pipeline.

protocols that constitute the protocol stack, processes these requests by invoking the specified protocol(s) that constitute the protocol stack. After creating appropriate protocol data units, the communication software issues appropriate commands to the *communication hardware* to transmit the data on the network. Communication hardware is comprised by one or more network adapters.

For reception, the API may block the requesting task if the requested data has not arrived. On reception of a datum, the communication hardware "interrupts" the communication software, thus initiating processing of the received datum. After processing the datum is passed to the API, which passes it on to the waiting (destination) task, possibly after some buffering (if the destination task is not waiting) and/or protection checks. In a multi-stage pipeline such as this, the slowest stage determines the throughput and latency of communication, and hence the performance delivered to the application. Similarly, variable per-packet delay (e.g., due to uncontrolled first-in-first-out queueing) incurred at a stage may introduce significant amounts of jitter in the observed end-to-end delay.

During protocol processing and data transfer, the protocols may invoke various operating system resource management functions such as timer services, buffer management, and device management (i.e., interrupt servicing and low-level device operations), as shown in Figure 2.2. Depending upon the protocol architecture, kernel architecture, and process architecture [152], other resource management functions such as process management and scheduling may also occur during data transfer. On multiprocessor hosts, processor allocation may also occur in addition to the above. An extensive survey of transport system architecture services and implementation strategies in a variety of operating systems such as System V UNIX, BSD UNIX, x-kernel, and Choices is presented in [152].

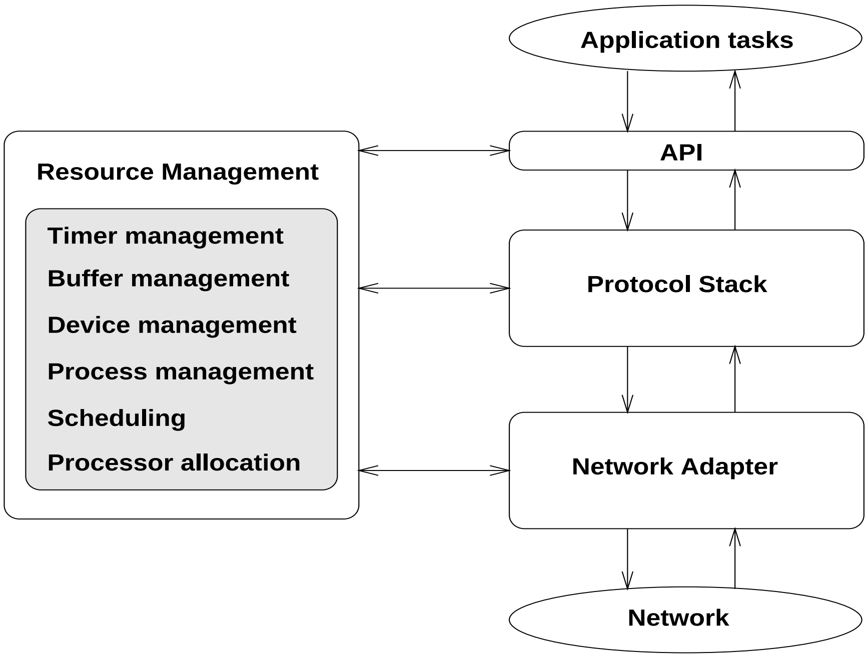


Figure 2.2: Interaction between protocols and resource management functions

2.3 Factors Affecting Performance

As mentioned, communication subsystems provide applications with a range of communication services through a variety of communication protocols, management of the necessary communication resources, and interaction with network devices for data transmission and reception. With slower networks, packets (or messages) were generated and consumed at a slower rate, and therefore the host was generally not the bottleneck. Gigabit networks have shifted the bottleneck from the transmission media to the end hosts [145, 161]. For the designers of the communication subsystem, therefore, the primary challenge has been realization of high-throughput and low-latency communication.

The performance of communication subsystems is primarily determined by three factors:

- the semantics of the API, which determines the cost of moving data to/from the communication subsystem,
- the overhead of protocol stack execution, including the protocol processing performed by individual protocols and the numerous OS resource management services invoked during execution, and
- the multiprogramming and network input load on the host, that determines the extent

of resource contention experienced by communicating tasks and within the communication subsystem.

We do not consider factors that affect end-to-end communication performance, such as the traffic characteristics and contention in the network, and the network topology. Contention can occur for resources such as processing capacity, network buffers, and network links, and may occur inside the sending as well receiving hosts. We discuss each of the above-mentioned factors in detail next.

2.3.1 API Semantics

The location and data passing semantics [25] of the API affects communication performance in a variety of ways. Typically the API is implemented by the kernel (e.g. BSD Sockets [108]), or by a trusted server at user level, and the overhead of crossing protection domains (e.g., the user-kernel boundary) during data transfer degrades communication throughput. Splitting the API into data transfer and control parts, with the data transfer part implemented as a library at user level, was shown to improve throughput substantially [112]. The amount of buffering available at the API also affects the protocol dynamics of end-to-end protocols. An application transferring large amounts of data may be blocked if enough socket buffers are not available. Similarly, the amount of buffer space available is used by protocols such as TCP to invoke flow control and congestion control mechanisms [84].

Existing APIs such as BSD Sockets have traditionally provide applications no control over resource management for communication-related activities. Thus, for the application the communication subsystem is a "black box," accepting requests for data transmission or reception and informing the application upon completion. There can be wide variation in the application-generated traffic patterns presented to the communication subsystem. These patterns include bulk data transfers, short, interactive request-response style communication, and the sustained access patterns of applications supporting continuous media. Together with the size of the transfers, the time between successive communication requests (the "think time") reflects the grain of the application and determines how often the application performs network data transfer. For an application performing uni-directional video transfer, for example, frames of varying sizes may be generated at periodic (33 ms) intervals.

The access pattern might be significantly different for interactive multimedia applications.

Knowledge of such application patterns may benefit communication subsystems, which could employ resource management policies significantly different from those used for computation. The scheduling schemes for messages and packets, the priorities assigned to them, and the management of protocol threads can then be tuned to the characteristics of network communication and QoS requirements of applications. In addition, this allows protocol processing to be scheduled via thread scheduling schemes distinct from those developed for application threads, especially for multimedia applications [6, 70].

In recent years, resource management for computation has moved towards giving more control to the applications, e.g., user-level threads [113], scheduler activations [7], and application-kernel coordination to dynamically vary the degree of active parallelism in scientific computing applications [174]. Providing applications with similar control over thread management and scheduling within the communication subsystem can potentially improve performance by exploiting locality, improving resource utilization, and reducing context switching and scheduling overheads.

In general, the API can be extended to allow *application-specific* resource management. This flexibility allows the application to manage its own computation and communication resources. However, leaving resource management entirely up to the application may not suffice for QoS-sensitive management of resources. Alternately, applications and operating systems can be designed to cooperate with one another to perform QoS negotiation and adaptation to maximize service accessibility and continuity [2, 33, 104]. A novel way to realize application-specific policies in networking is via installing of application code directly in the kernel [64]. We note that requiring applications to specify their traffic characteristics and desired QoS levels also realizes application-specific resource management, in that the communication subsystem and the operating system are made aware of, and undertake to allocate and manage resources in accordance with, application QoS requirements.

2.3.2 Protocol Stack Execution

Once application data is handed to the communication subsystem, it is processed by all the protocols constituting the protocol stack. For outgoing data, this may involve fragmentation and header encapsulation, while for incoming data this may involve packet classification, header removal, and reassembly. Additional computationally-intensive functions such as checksum computation, encryption, coding and compression may also be performed. For example, growing concerns of security and privacy on the Internet have led to the development of a number of security standards such as the Secure Socket Layer (SSL) and Secure HTTP.

The execution performance of protocol stacks depends on a number of factors. The first and foremost is the processing cost imposed by each layer in the protocol stack. This cost depends on the particular function performed by the corresponding protocol, which in turn may depend on the type of the data to be processed. The aggregate processing cost of the protocol stack depends on the depth of the protocol stack, i.e., the number of protocols comprising the stack. Recent trends in the development of WWW point to the increasing prevalence of deeper protocol stacks which enhance the functionality provided by the communication subsystem.

One of the primary factors determining communication subsystem performance is the number of times data is copied during processing. Excessive data copying has been identified as one of the main reasons behind the relatively smaller improvements in the performance of networking software compared to the advances made in networking hardware [142]. While data copying primarily impacts data transfer throughput, protocol processing latency is significantly affected by the complex, protocol-specific cache behavior of network protocols [16,133].

Finally, protocol stack execution performance is significantly affected by operating system overheads such as context switching and device interrupts [163]. The number of context switches incurred during protocol processing are a function of the protocol architecture employed [152], while the number of interrupts is a function of the design of the network adapter and the packet input mechanism adopted [151,173]. Further, since the communication subsystem typically shares processing resources with application threads, additional scheduling and context switching overheads, and associated cache misses [124], may be incurred. Additional sources of overhead include buffer management, timer management, and error-recovery mechanisms such as retransmissions. As highlighted in the remaining chapters of this dissertation, support for QoS-sensitive handling of data imposes new overheads and demands on communication subsystem performance.

With faster networks, acceptable application-level throughputs and latencies cannot be achieved without a substantial investment in managing the communication subsystem resources efficiently. As described in Section 2.4, this has fueled significant redesign, optimization, and exploitation of parallelism within the communication subsystem.

2.3.3 Multiprogramming and Network Load

There are two primary aspects to multiprogramming and network load as it relates to communication performance: (a) a task engaged in network I/O may compete for host resources with other purely computational tasks, and (b) it may compete with other tasks also simultaneously engaged in network I/O. In both cases the background load and operating system scheduling policy together determine the frequency and duration of execution of the task. Scenario (a) disrupts the execution profile (and hence the packet input or output rate) of the network task due to uncontrolled contention for the CPU. Scenario (b) introduces additional uncontrolled contention for communication resources such as output queues and link bandwidth.

For a sending task, scenario (a) can arbitrarily lower network utilization or produce bursty traffic on the network, thus introducing significant jitter in the packet transmission profile and even incurring substantial packet loss. For a receiving task, delays in receiving arrived messages (possibly due to contention from other higher priority tasks) may result in dropped packets due to buffer overflow. Since the task is unable to receive data from the socket buffers, flow control mechanisms in protocols like TCP may stall the sender until the received data is acknowledged. Both these effects lower throughput drastically and also increase the end-to-end communication delay. QoS degradation can occur with asynchronous sends as well; because of limited socket buffer space, the sender may be blocked if the previous network transmission requests were pending, or the excess data may get dropped inside device driver or adapter queues.

Scenario (b) is similar to scenario (a) in that the tasks still experience QoS degradation due to uncontrolled contention for the CPU. However, due to multiple tasks performing network input/output, the order of packet processing and transmission is also arbitrary. This interference manifests itself in higher, variable queuing delays for protocol processing and transmission/reception, and hence unpredictable degradation in the delivered QoS. Due to higher total aggregate traffic, the likelihood of packet loss is also significantly higher and unpredictable. Once again, the packet transmission profile is rendered extremely bursty, which may increase network load and hence further QoS degradation in downstream routers or the receiving host.

For example, the receiving host may be unable to keep up with the packet arrival rate, which is now higher because of a bursty sender. A persistent burst of packet arrivals at the receiver can result in *receive livelock* [151]. Receive livelock is a phenomenon in which, under network input overload, a host or router is swamped with processing and discarding arriving packets to the extent that the effective throughput of the system falls to zero. We highlight strategies to prevent or eliminate receive livelock in Section 2.4.

Experimental and analytical studies

Several studies have attempted to study the effects of multiprogramming on communication performance experimentally as well as analytically.

An in depth study of the effect of background load on communication performance for the 4.2 BSD TCP/IP protocol stack is reported in [31]. The delivered end-to-end delay and throughput for TCP and UDP traffic is measured against background load for a range of message sizes. The effect of background Ethernet load on TCP and UDP performance is studied in [21]. More recently, the effect of background load on the performance of SunOS inter-process communication (IPC) mechanisms and TCP/IP protocol stack is studied in [143], for an artificial CPU-intensive workload as well as a real distributed application. The study highlights the negative influence of UNIX scheduling mechanisms on IPC performance; delays in scheduling processes engaged in network I/O interferes with the flow control mechanisms of the communication protocols coordinating the data transfer, causing an increase in end-to-end delay and a substantial drop in throughput.

Useful insights into the subtle effects of multiprogramming and network input load on packet reception time are provided by Danzig [47]. He constructs a performance model of protocol processing in UNIX [108] to capture the effects of multiprogramming and limited socket buffers. The model accounts for protocol processing overhead and the variability introduced by interrupt servicing, scheduling latency, and memory contention, in the time taken by a destination process to receive arriving packets. This study provides ample evidence that both multiprogramming and network load introduce subtle degradations in the delivered QoS.

Implications for CPU scheduling

Provision of guarantees on the delivered QoS necessitates appropriate operating system support to schedule applications. While we describe possible approaches to QoS-sensitive CPU scheduling in Section 2.5, here we describe the results from an interesting study done by Mogul [126]. He argues that communication delays in a multiprogrammed system may be reduced through "dallying". Dallying, or latency-sensitive scheduling, is a scheduling strategy in which the CPU scheduler allows a process waiting for network input to busy-wait on the CPU for some additional time in the current quantum, instead of blocking it and selecting another process to run. The tradeoff is between busy-waiting (and avoiding later contention from background load) and excessive context switching.

Mogul's observations are based on the presence of locality between communicating processes resident on distinct hosts connected by a local area network. He observes high per-process locality such that three quarters of all packets arriving at a host are for the same process that received the previous packet, while one quarter to two thirds of incoming packets are for the process that most recently sent a packet. Clearly, the effectiveness of dallying largely depends on the heuristics used for predicting the delay until the next packet arrival based on the history of packet arrivals for the process. We note that provisioning QoS guarantees on end-to-end communication allows such packet arrival characteristics to be readily constructed from the traffic and QoS specifications supplied by applications, assuming bounded traffic distortions through the network.

2.4 Efficient Protocol Architectures and Implementations

To reap the benefits of the advances made in networking technology, and translate them to good application-level performance (i.e., high throughput and low latency), it is necessary to minimize transmission and reception overheads within the communication subsystem via careful implementation. The techniques employed to improve the delivered throughput and latency of communication protocols include [61]

- minimization of data-copying within the communication subsystem and across the API [52, 181],
- optimization of protocol implementations, such as hand-optimized critical paths [40],

and integrated layer processing [1, 41],

- appropriate network interface design for high performance [161], and,
- exploitation of concurrency and parallelism within the communication subsystem [152].

We discuss each of these techniques and their compatibility with, and implications for, QoS-sensitive communication subsystem design.

2.4.1 Performance optimizations for efficient data transfer

A wide variety of performance optimizations have been applied to communication subsystems. The most significant of these are briefly described below.

Buffer management for data copy elimination: Elimination of unnecessary data copying via appropriate buffer management has been the focus of numerous research efforts in recent years. Proposals to achieve high-bandwidth data transfer across protection domains (e.g., between the kernel and an application) include restricted virtual memory remapping [175], container shipping [147], and Fbufs [54]. Support for in-kernel device-to-device data transfers for multimedia applications is described in [60, 93]. An extensive taxonomy of the software and hardware tradeoffs involved in data passing and performance comparison of different buffering semantics is provided in [25], and two copy avoidance techniques evaluated in [27]. Network adapter support for checksummed, multiple-packet communication in an ATM network is explored in [26].

Protocol stack optimization: A number of research efforts have focused on the optimization of protocol stack execution latency. These include protocol-specific optimizations for TCP [40] and UDP [146], improving data touching overheads of checksumming for UDP/IP stacks [97], and exploring the non-data touching processing and related operating system overheads in TCP/IP stacks [96]. A detailed study of factors affecting end-to-end communication latency in LAN environments is described in [170]. ILP, first proposed in [41] and subsequently implemented and evaluated in [1], reduces the number of accesses to network data by effectively collapsing protocol layers and executing them in an integration fashion for each data word accessed. Several recent efforts have also focused on optimizing the protocol processing latency in TCP/IP protocol stacks [16, 129, 176].

User-level protocol processing: Several research efforts have focused on increasing communication subsystem throughput via user-level handling of network data [57, 112, 167]. In addition to data copy minimization compared to a server-based implementation, user-level protocol processing offers significant flexibility in developing and debugging communication protocols. With appropriate support in the network interface and the operating system, it is possible to grant applications full access to the network interface, thereby bypassing the operating system completely [28, 55].

Application-specific networking: Recently there has been much interest in OS extension technologies, in which trusted application code is installed for execution within the OS. This allows operating systems to be customized or "extended" to implement application-specific policies. Plexus [64] and application-specific handlers in the Aegis kernel [180] are two examples of approaches to realize application-specific networking via OS extensions.

Packet classification: Packet filters provide general and flexible classification (i.e., demultiplexing) of incoming packets to application end-points, at the lowest layer of the protocol stack. They were first proposed primarily to enable user-level network capture for developing new protocols without kernel modifications [127], but are increasingly being utilized for realizing protocols such as UDP and TCP as user-level libraries. Successive implementations of packet filters have reduced classification overhead significantly, e.g., BPF [114], MPF [186], and PATHFINDER [11]. While all of these filter designs perform classification via interpretation, more recently dynamic code generation techniques have been applied to realize very efficient packet filters, as in DPF [59]. Packet filters are necessary for packet classification in connectionless networks, such as the IPv4-based Internet. Further, they may also be needed in native ATM networks that aggregate multiple end-to-end connections over virtual circuits [11].

Receive livelock elimination: Besides optimizing the receive data path through the communication subsystem, efforts have also been made to address another key problem associated with data reception, namely, receive livelock [151]. Receive livelock has been addressed at length in [125] via a combination of techniques (such as limiting interrupt arrival rates, fair polling, processing packets to completion, and regulating CPU usage for protocol processing) to avoid receive livelock and maintain system throughput near the maximum system input capacity under high load. Lazy receiver processing (LRP) [53], while not completely eliminating it, significantly reduces the likelihood of receive livelock even under high input load. In LRP, an incoming packet is classified and enqueued, but not processed, until the application receives the data. We have studied adapter support for receive livelock elimination using the END [81].

"Paths" through the communication subsystem: The Path abstraction realized in the Open Software Foundation's CORDS framework [172] provides a rich framework for development of real-time communication services for distributed applications. Using this abstraction, unique *paths* can be defined through the communication subsystem, and path-specific allocation performed for resources such as packet buffers, input packet queues, and input shepherd threads. Similarly, the Scout operating system from the University of Arizona proposes the use of explicit paths as an important abstraction in operating system design to improve performance [128]. However, while paths in [172] are envisioned primarily as a static, relatively coarse-grain mechanism, paths in [128] are not associated with communication resources and assigned deadlines or priorities via admission control.

Implications for QoS-sensitive protocol processing

With the exception of paths, most of the optimizations and design strategies described above are geared primarily towards traditional best-effort traffic with the primary goal of increasing data transfer throughput or reducing latency. As such these efforts are complementary to our work, which focuses on provision of QoS guarantees for communication. Moreover, these optimizations continue to be applicable in our proposed QoS-sensitive architecture as well as the enhancements made to TCP/IP stacks for integrated services. In constrast, we examine the overheads imposed by *new* data-handling components in the protocol stack. We adapt and extend the notion of paths in [128, 172] to obtain QoS guarantees and communication resources via admission control, and perform fine-grain path-based resource management to maintain QoS guarantees.

While packet filters were designed primarily to classify incoming packets, they can also be used to classify outgoing traffic at network routers. However, traffic classification for provision of QoS guarantees imposes additional requirements on the classification function, and as such may require more elaborate functionality [178]. Regarding receive livelock elimination, while approaches such as LRP work well for best-effort traffic, appropriate OS support is needed to ensure the application is scheduled to run in a QoS-sensitive fashion. Moreover, architectural support similar to that presented in this dissertation is needed to multiplex resources across multiple connections originating from the same application.

2.4.2 Network Interface Design

With network bandwidth increasing faster than processor and memory bandwidth, efficient hardware support may be provided for low-level communication protocols, thus relieving the burden on the communication subsystem software. By migrating frequent and time-consuming communication functions into hardware, the available network bandwidth can be utilized effectively. Since the network adapter defines the communication primitives available to the communication software, flexible support for network I/O necessitates a careful division of functionality between the adapter and the host processor [89, 164]. Network adapters either facilitate flexibility in supporting different protocols through simple designs [46, 103, 120], or restrict flexibility, and hence improve performance, by supporting specific communication protocols and resource management strategies [37, 90].

In general, the overhead of host-adapter interaction and unnecessary data movement across the system bus can limit the amount of useful concurrency between the software and hardware portions of the protocol stack, thus affecting communication subsystem performance. The design of high-speed network adapters, their performance characteristics, and implications for protocol stacks in uniprocessor workstation environments has received significant attention recently, for FDDI [151] as well as ATM [48, 173] networks.

Uniprocessor front-ends and network adapters: With dedicated network front-ends, the network adapter operates under control of a front-end *communication processor* which handles all communication-related functions including network interrupts. Data exchange between the host and the communication processor is performed either by explicit data copying or through shared memory. Not only does this permit special tuning of the functionality desired and significantly reduce the number of interrupts delivered to the host, it also permits exploitation of available concurrency through pipelining of the transmission and reception datapaths [90].

More importantly, it permits management of the communication subsystem by a separate executive designed for communication-related operations, such as the x-kernel [76]. The VMP NAB [90] and Nectar CAB [120] are two examples of this functional partitioning. For example, the Nectar CAB [10, 120] off-loads all protocol processing functions from the host processor, freeing it from adapter handshake overheads and permitting greater overlap between useful computation and communication processing.

Multiprocessor front-ends: Multiprocessor front-ends may be designed using special-purpose or general-purpose hardware. Special-purpose designs facilitate efficient interaction with the host and the network interface unit [82, 83], while general-purpose designs must explicitly coordinate accesses to these interfaces [14, 138, 182]. More processing power in the front-end can improve the quality of service provided to the applications, by reducing queuing delays within the communication subsystem.

Network interfaces for multicomputers: Network interface design for multicomputer environments presents unique opportunities and cost-performance tradeoffs. In dedicated multicomputers, network interface designs are typically tightly coupled with the processor, and as such can exploit attributes of such a tight coupling. For example, the Hamlyn [30] network interface performs sender-controlled data transfer, in which the sender controls the memory location at the receiver where data will be deposited. Similarly, the network interface for the SHRIMP multicomputer [18] realizes direct writes to remote memory via virtual memory mapped communication [56]. This requires specialized network interface hardware that maintains virtual to physical addressing mappings.

Implications for QoS-sensitive transmission/reception

Neither of these approaches are feasible for QoS-sensitive communication between workstation-type clients and servers across WANs. For example, unlike these approaches, clients and servers do not trust one another. Moreover, the operating system cannot be bypassed; instead, it must explicitly validate remote accesses and, at the very minimum, schedule packet transmissions and receptions. Further, all application data must be processed using appropriate protocols, thus necessitating CPU involvement during data transfer.

Most of the network adapter design tradeoffs highlighted above have been explored primarily in the context of best-effort network traffic. That is, while these design optimizations improve data transfer throughput, no explicit guarantees are given; furthermore, improvements in throughput may be obtained at the expense of low, predictable latency. For example, FIFO queueing in the network adapter allows successive packet transmissions to be pipelined, thereby improving throughput; however, it exacerbates packet transmission latency while making it more unpredictable. Appropriate queueing and scheduling support is needed on network interfaces to support QoS-sensitive traffic. We have studied the implications of network adapter characteristics for real-time communication on a Fibre

Channel adapter manufactured by Ancor Communications [77]. As highlighted in Chapter 4, QoS-sensitive handling of network data also requires appropriate support on network adapters.

2.4.3 Parallel Protocol Implementations

While we do not consider multiprocessor hosts or parallelized communication subsystems in this dissertation, it is important to understand the issues and tradeoffs in exploiting and managing communication parallelism. Multiprocessor hosts are increasingly be adopted as workstations or servers. Accordingly, our design for a QoS-sensitive communication subsystem should, at the very least, permit extension to multiprocessor hosts. Moreover, we have derived the basic structure for the architecture described in Chapter 3 from the proposed architectural approaches to manage communication parallelism, as outlined below.

With the advent of high-speed networks, uniprocessor protocol implementations (on the host or the network interface) are often unable to keep up with, or fully utilize, the network [83], resulting in performance bottlenecks within the communication subsystem. This performance degradation is all the more pronounced for multiprocessor hosts due to a higher rate of message generation and consumption. The pursuit of application-level gigabits-per-second has prompted recent efforts to develop faster communication subsystems through the exploitation of parallelism.

In general, communication subsystems employ one or more processes (or threads) to implement a protocol graph [76]. Depending upon the allocation of work to these processes, communication subsystems can be classified into *horizontal* or *vertical* process architectures [152]. In horizontal architectures [182], each process implements a specific layer of a protocol graph; at most two processes can be assigned to each layer, one each for transmission and reception. In vertical process architectures [14, 76, 83], on the other hand, processes are assigned to active entities such as connections or messages and each process implements one path through the protocol graph. This approach significantly reduces context switches and message buffering that are unavoidable in horizontal process architectures.

Grains of parallelism: Several grains of parallelism can be exploited within both these approaches [152]. These grains roughly correspond to the mapping between the process architecture and the available processing resources. Finer grains of parallelism can be exploited for vertical process architectures. A relatively coarse-grain technique, *connectional* *parallelism*, associates a processor with each active connection. All messages belonging to a connection are handled by the associated processor. Relatively *fine-grain* techniques include *message parallelism* and *packet parallelism.* With message parallelism a processor is associated with each message generated or consumed at the host.

Packet parallelism, on the other hand, associates a processor with each arriving or departing packet [83]. Various combinations of these approaches are also possible. Fine-grain techniques provide more scalable parallelism and hence potentially greater performance gains. In practice, though, synchronization constraints, resource contention, and load balancing requirements limit the speedups, and hence the message throughputs, observed.

Overheads of managing parallelism: The techniques used in [14] to parallelize x-kernel [76, 141] and the protocols, including the locking mechanisms used and dedication of special x-kernel functions to specific processors, suggest that special synchronization paradigms and processor allocation mechanisms are needed to manage communication parallelism effectively. Synchronization and contention overheads may seriously limit the amount of parallelism that can be exploited effectively in practice. As shown in [14], parallel implementations of contemporary transport layer protocols like TCP are synchronization-limited because these protocols retain a significant amount of connection state that must be maintained consistently. In comparison, the results obtained for UDP show higher speedups since UDP maintains significantly less state than TCP.

This limitation is in part due to the fact that protocols such as TCP were not designed with parallel implementations in mind. Large connection state and the requirement of in-order delivery of packets to applications are two examples of features that limit performance in parallel implementations. Increased parallelism is likely to be a feature of the next generation of communication protocols [150, 191]. Recent results have shown that connectional parallelism delivers comparatively more scalable performance than message parallelism [134, 153, 154].

Protocol processing in SMMPs: Similar problems arise when implementing protocol processing in shared-memory multiprocessors (SMMPs), where the approaches adopted essentially lie on two extremes. In one approach, each processor executing a process also performs protocol processing for messages transmitted by that process [169]. In this model, protocol processing is treated as work strictly local to each processor, resulting in an implicit sharing between the computation and communication subsystems. An alternative approach treats protocol processing as global work that can be scheduled uniformly on any available processor [67, 99]; this results in explicit sharing between the two subsystems.

Implications for QoS-sensitive protocol processing

These approaches to exploiting communication parallelism may not suffice for QoS-sensitive protocol processing since they introduce unpredictability in the availability and allocation of processing resources, and complicate global coordination for network access. A process-per-message model seems unsuitable for QoS-sensitive protocol processing. Assuming that each message's shepherd process is independently scheduled on the protocol processors, simultaneous processing of multiple messages from the same connection would lead to out-of-order consumption of protocol processing and transmission bandwidth. Further, shepherd processes handling messages belonging to the same connection must now synchronize to maintain consistent connection state.

With potentially several connections mapped to the same processor, it becomes more expensive to coordinate handling of messages on a connection and between connections. Accordingly, the process-per-connection model, with appropriate extensions, seems the most suitable candidate for QoS-sensitive protocol processing. We have proposed an architecture for QoS-sensitive protocol processing on SMMPs that maps and schedules protocol processing for different guaranteed-QoS connections on a dedicated set of protocol processors [119], with each connection handler mapped to exactly one processor. Our approach of statically partitioning the processing resources is similar to multiprocessor front-ends [138], except that a set of processors within the host are dedicated for protocol processing and communication-related functions, as in [14]. Accordingly, the architecture described in Chapter 3 is based on the process-per-connection model of protocol processing.

2.5 QoS-Sensitive Communication and Computation

The preceding section focused on techniques that improve the average performance of communication subsystems, and hence the application-level throughput and latency realized. However, these techniques do not attempt to perform QoS-sensitive data handling. In this section we survey related work in QoS-sensitive communication and computation, and compare and contrast it with the contributions made by this dissertation.

A number of approaches are being explored to realize QoS-sensitive communication and computation in the context of distributed multimedia systems. An extensive survey of QoS architectures is provided in [32], which provides a comprehensive view of the state of the art in the provisioning of end-to-end QoS. In the discussion below, we highlight a subset of these approaches, focusing on enhancements and the associated implications for end hosts. We first discuss approaches to QoS negotiation and adaptation, then highlight communication architectures for QoS, followed by related work in multimedia and real-time operating systems.

2.5.1 Dynamic QoS Negotiation and Adaptation

Since a broad class of multimedia applications are soft real-time in nature, i.e., can tolerate limited fluctuations in the delivered QoS, several research efforts have explored the issues involved in supporting QoS negotiation and adaptation functions at end hosts. The AQUA system [104] is one such effort which has developed QoS negotiation and adaptation support for allocation of CPU and network resources. Similarly, a QoS adaptive transport system is described in [33] that incorporates a QoS-aware API and mechanisms to assist applications to adapt to fluctuations in the delivered network QoS. A scheme for adaptive rate-controlled scheduling is presented in [184].

A dynamic QoS control scheme using optimistic processor reservation and application feedback is described in [135]. This system, which is implemented in RT-Mach [171], reserves processing capacity based on average processing usage of applications and provides notifications to applications to adjust QoS levels. QoS negotiation and adaptation support has also been developed for real-time applications [2], which provides support for specification of QoS compromises and supports graceful QoS degradation under overload or failure conditions.

The negative effects of the scheduling variability introduced by the UNIX operating system on audio playback is highlighted in [102]. Application-level support to adapt to such QoS variations and its realization in a real-time audio tool is also described.

While we do not consider dynamic QoS negotiation and adaptation in this dissertation, most of the architectural mechanisms and enhancements can be utilized for such scenarios. For example, the architectures proposed in Chapters 3 and 7 provide mechanisms to enforce application traffic contracts and generate notifications in response to which applications can adapt. Future incarnations of the guaranteed-QoS service described in Chapter 5 will include support for dynamic QoS negotiation.

2.5.2 Communication Architectures for QoS

Architectural support for QoS in communication includes network and protocol support for signaling and data transfer, appropriate QoS architectures for host communication subsystems, run-time support to manage QoS-sensitive traffic, and appropriate design of network adapters. Related work in each of these areas is described below.

Network and protocol support for QoS: The DASH project [4, 5] explored provision of performance guarantees for unicast sessions in IP networks. It argued that bounded processing delays and overheads within the communication subsystem can be achieved through appropriate priority-based scheduling [6]. The Tenet real-time protocol suite [12] is one of the first implementations of real-time channels [63] on wide-area networks (WANs), and supports both deterministic and statistical service guarantees. In addition to providing protocol support for end-to-end signalling, support is provided for QoS-sensitive packet scheduling at hosts and routers via Rate-Controlled Static-Priority Queueing [187].

However, the Tenet project does not consider incorporation of protocol processing overheads into the network-level resource management policies. In particular, it has not addressed the problem of QoS-sensitive protocol processing inside hosts. Further, they do not consider incorporation of implementation constraints and overheads, and simultaneous management of CPU and link bandwidth for transmission and reception. While they also develop architectural enhancements for a sockets based communication subsystem, the protocol suite adopted does not conform to IETF standards for integrated services. Moreover, they do not provide support for network interfaces with widely differing capabilities.

In the context of integrated services on the Internet, much support being provided on the Internet is geared towards multicast communication, in contrast with our primary focus on unicast real-time channels. We note that the architectural approach, mechanisms, and extensions developed in this dissertation are applicable to unicast as well as multicast sessions, for both sender-initiated and receiver-initiated signalling.

QoS architectures: The OMEGA [132] end point architecture provides support for end-to-end QoS guarantees. In this architecture, application QoS requirements are translated to network QoS requirements by the QoS Broker [131], which negotiates for the necessary host and network resources. The OMEGA approach assumes appropriate support from the operating system for QoS-sensitive application execution, and the network subsystem for provision of transport-to-transport layer guarantees. The primary focus of OMEGA is development of an integrated framework for the specification and translation of application QoS requirements and allocation of the necessary resources.

QoS-A [34] is a layered architecture focusing on provision of QoS within the communication subsystem and the network. It provides features such as end-to-end admission control, resource reservation, QoS translation between layers, and QoS monitoring and maintenance. QoS-A specifies a functionally rich and general architecture supporting networked multimedia applications. Practical realization of QoS-A, however, would necessitate architectural mechanisms and extensions similar in flavor to the ones demonstrated in this dissertation.

We have designed and implemented an RSVP-based QoS architecture supporting integrated services in TCP/IP protocol stacks, running on legacy (e.g., Token Ring and Ethernet) LAN interfaces as well as high-speed ATM networks [13], as described in Chapter 7. A native-mode ATM transport layer has been designed and implemented in [3], and enhanced with QoS support for a user space implementation [75]. Similar to our RSVP-based QoS architecture, traffic policing and shaping is performed while copying application data into kernel buffers; the application is blocked if it is violating its traffic specification. However, our design is applicable to general TCP/IP protocol stacks, including legacy LAN and ATM interfaces. Further, whether shaping is performed on traffic associated with a reservation is determined from the service class that reservation belongs to.

Support for QoS-sensitive communication: The design of a QoS-controlled communications system for ATM networks is described in [43]. However, implementation overheads and constraints are not incorporated in the resource management policies. Moreover, no performance impact of supporting QoS in communication is reported.

Real-time upcalls (RTUs) [68] are a mechanism to schedule protocol processing for networked multimedia applications via event-based upcalls [38]. Protocol processing activities are scheduled via an extended version of the rate monotonic (RM) scheduling policy [111]. Similar to our approach, delayed preemption adopted to reduce the number of context switches. Our approach differs from RTUs in that we use thread-based execution model for protocol processing, schedule threads via a modified earliest-deadline-first (EDF) policy [111], and extend resource management policies within the communication subsystem to account for a number of implementation overheads and constraints.

Similar to our approach, rate-based flow control of multimedia streams via kernel-based communication threads is also proposed in [185]. In contrast to our notion of per-connection threads, however, a coarser notion of per-process kernel threads is adopted. This scheme is clearly not suitable for an application with multiple QoS connections, each with different QoS requirements and traffic characteristics. Mechanisms for scheduling multiple communication threads, and the issues involved in reception side processing, are not considered. More importantly, the architecture outlined in [185] does not consider provision of signaling and resource management services within the communication subsystem.

Communication support for efficient messaging in distributed real-time environments is provided in FLIPC [15]. FLIPC exploits programmable communication controllers to realize low latency message transfer. Similar to our approach, arriving messages are not processed in interrupt context, but processed by a thread that is in turn scheduled for execution. However, FLIPC does not provide any explicit QoS guarantees on the end-to-end message delivery. The CORDS path abstraction [172], which is similar to Scout paths [128], provides a rich framework for development of real-time communication services for distributed applications, as demonstrated with the guaranteed-QoS communication service in Chapter 5.

As mentioned earlier, while LRP [53] works well for receive-livelock elimination for best-effort traffic, the architectural approach outlined in this dissertation is needed to extend it to accommodate QoS-sensitive traffic. Similar to LRP, our approach also utilizes early demultiplexing and channel-specific queueing of incoming packets. However, packet processing and message reassembly is performed in a QoS-sensitive fashion via EDF scheduling of channel handlers, as and when communication capacity is made available. Demultiplexing incoming packets early and absorbing bursts in distinct per-connection queues is an attractive way to prevent receive livelock, an observation also made in the context of paths in Scout [128]. Our architectural approach facilitates provision of QoS guarantees while preventing receive livelock.

Network adapter design: The MNI [17] is a network interface unit designed specifically for continuous media applications. It supports functions for full end-to-end multimedia communication in addition to protocol processing, and is capable of moving data directly between I/O devices and the network via device-to-device communication. The Credit Net ATM network interface [101] provides appropriate buffer management and flow scheduling support suitable for connections engaged in guaranteed-QoS communication.

We have examined the impact of adapter characteristics on the ability to support real-time communication effectively [77], and have explored QoS support on network adapters for point-to-point and shared networks [80]. We have designed SPIDER [51], an adapter supporting real-time communication in point-to-point networks. Other adapter features that facilitate provision of QoS guarantees include allowing the communication software to exercise fine-grain control over packet transmissions, and the ability to efficiently examine packet headers and selectively discard packets, if needed, on reception.

2.5.3 Multimedia/Real-Time Operating Systems

A number of commercial and research efforts have focused on the development of multimedia and real-time operating systems, as described below.

Commercial and research operating systems: Govindan and Anderson [70] first proposed scheduling and IPC mechanisms for operating systems supporting continuous media. Key abstractions proposed and implemented were a split-level scheduling architecture featuring an EDF [111] kernel scheduler and an application-level scheduler for efficient real-time scheduling. The application-level and kernel-level schedulers communication with each other via shared memory.

Support for continuous media in the Chorus microkernel is described in [42] which utilizes the IPC and scheduling techniques proposed in [70]. Nemesis [109] is a multimedia operating system design from scratch. While many aspects of operating system design have been considered, not much attention has been given to QoS issues in communication subsystem design. The Rialto operating system [87, 88] combines minimum-laxity and fair scheduling policies to realize integrated scheduling of real-time and non real-time processes. The primary unit of execution in Rialto is an activity, which is a time-constrained section of code which can adapt to overload conditions. The focus of Rialto is primarily on QoS-sensitive application scheduling, and issues in QoS-sensitive communication subsystem design are not considered.

Real-Time Mach (RT-Mach) [171] provides processor capacity reserves [123], a mechanism to allow applications to reserve processing capacity for execution of its threads. Reserves are independent of application threads and can account for execution of a thread in multiple protection domains. While capacity reserves are useful for applications with a priori knowledge resource requirements, they are also useful in environments requiring dynamic QoS control [106]. Virtual memory management for interactive multimedia applications has also been realized in RT-Mach [136].

Modifying an existing operating system to support multimedia and real-time applications is typically a major undertaking and prone to subtle timing issues. Commercial operating systems such as Solaris have been enhanced with some support for real-time applications, such as fixed priorities and priority inheritance [99]. However, such mechanisms have been found to be insufficient in a dynamic execution environment [104]. Extensions to Windows NT to support dynamic real-time applications are described in [162]. Support for real-time processes, admission control, and detecting and managing overrun is provided. However, as stated by the authors, they have not yet accounted for implementation overheads and constraints. Moreover, the issues and tradeoffs involved in realizing QoS-sensitive communication subsystems are not considered.

Modeling of multimedia/real-time operating systems: A number of recent efforts have attempted to bridge the gap between theory and practice for real-time systems and multimedia computing [29, 69, 95, 98]. The admission control extensions developed in Chapter 4 are geared towards the real-time communication needs of distributed systems, and hence complement these efforts. The implications of priority inversion due to non-preemptible critical sections was studied in [121]; however, preemption costs such as context switches and cache misses, and the resulting degradation in useful resource capacity, were not considered. We take these costs into account when developing the admission control extensions described in Chapter 4.

OS support for QoS-sensitive communication: The need for scheduling protocol processing at priority levels consistent with those of the communicating application was highlighted in [6], and some implementation strategies demonstrated in [70]. More recently, processor capacity reserves in RT-Mach [123] have been combined with user-level protocol processing [112] for predictable protocol processing inside hosts [107]. Operating system support for multimedia communication is explored in [73], where the focus is on provision of preemption points and EDF scheduling in the kernel, and in [177], which also focuses on the scheduling architecture.

None of these approachees provide support for traffic enforcement or decoupling of protocol processing priority from application priority. Our design approach decouples the protocol processing priority from that of the application, allowing the former to be derived from the traffic characteristics and run-time behavior of the application.

QoS-sensitive CPU scheduling: Recently several QoS-sensitive CPU scheduling policies have been proposed recently [66, 71, 165, 179]. Scheduling algorithms for integrated scheduling of multimedia soft real-time computation and traditional hard real-time tasks on a multiprocessor multimedia server are proposed and evaluated in [94]. These schemes do not directly consider support for QoS guarantees on communication-related CPU processing; however, the architecture and extensions described in this dissertation can be integrated with these policies for application-level QoS, as outlined in Chapter 8.

CHAPTER 3

A QOS-SENSITIVE COMMUNICATION SUBSYSTEM

3.1 Introduction

In this chapter, we propose and evaluate a QoS-sensitive communication subsystem architecture for guaranteed-QoS connections. Our primary focus is on the architectural mechanisms for run-time traffic management within the communication subsystem to satisfy the QoS requirements of all connections, without undue degradation in the performance of best-effort traffic. While the proposed architecture is applicable to other proposals for guaranteed-QoS connections [8, 188], we focus on *real-time channels*, a paradigm for guaranteed-QoS communication services in packet-switched networks [63, 92].

Consider the problem of servicing several guaranteed-QoS and best-effort connections engaged in network input/output at a host. The data to be transmitted over each connection resides either in an input device (such as a frame-grabber) or in host memory; the computation subsystem prepares outgoing data in a QoS-sensitive fashion before handing it to the communication subsystem. Each guaranteed-QoS connection has traffic-flow semantics of unidirectional data flow, in-order message delivery, and unreliable data transfer. That is, on each channel, data is transferred from the source to the destination, successive messages on a channel are delivered in the order they were generated, and data that suffers loss of QoS guarantees within the network is unusable and hence not retransmitted. Generally speaking, these connection semantics are applicable to a large class of multimedia and real-time command/control applications. Best-effort traffic (i.e., traffic with no QoS guarantees) does not require in-order delivery, but may require retransmissions to ensure loss-free data transfer.

Protocol processing for large data transfers, common in multimedia applications, can be quite expensive. Resource management policies geared towards statistical fairness and/or time-sharing can introduce excessive interference between different connections, thus degrading the delivered QoS on individual connections. Since the local delay bound at a node may be fairly tight, the unpredictability and excessive delays due to interference between different connections may even result in QoS violations. This performance degradation can be eliminated by designing the communication subsystem to provide: (i) maintenance of QoS guarantees, (ii) overload protection via per-connection traffic enforcement, and (iii) fairness to best-effort traffic. These requirements together ensure that per-connection QoS guarantees are maintained as the number of connections or per-connection traffic load increases.

The proposed architecture features a *process-per-channel* model for protocol processing on each channel, coordinated by a unique channel handler created on successful establishment of the channel. While the service *within* a channel is FIFO, QoS guarantees on multiple channels are provided via appropriate CPU scheduling of channel handlers and link scheduling of packet transmissions. Traffic isolation between channels is facilitated via per-channel traffic enforcement and interaction between the CPU and link schedulers. Channels violating their traffic specification are prevented from consuming processing and link capacity either by blocking the execution or lowering the priority of the corresponding handlers relative to the well-behaved channels. Protocol processing can be work-conserving or non-work-conserving, with best-effort traffic given processing and transmission priority over "work ahead" real-time traffic. The architectural framework adopted utilizes an abstraction of the underlying communication subsystem in terms of various processing costs, overheads, and policies, which are used for admission control [116] and run-time resource management.

We have developed a prototype implementation of the proposed architecture using a communication executive derived from x-kernel 3.1 [76] that exercises complete control over a Motorola 68040 CPU. This configuration avoids any interference from computation or other operating system activities on the host, allowing us to focus on the communication subsystem. We evaluate the proposed architecture under varying degrees of traffic load, and demonstrate the efficacy with which it maintains QoS guarantees on real-time channels and provides fair performance for best-effort traffic, even in the presence of ill-behaved real-time channels.

While the proposed architecture is designed to handle both incoming and outgoing traffic, this paper focuses primarily on run-time communication resource management for outgoing traffic. The issues involved in QoS-sensitive handling of incoming traffic and the associated admission control extensions are presented in Chapter 4. Based on the proposed architecture and admission control extensions, we have designed, implemented, and evaluated a full-fledged guaranteed-QoS communication service on Open Software Foundation's MK 7.2/CORDS framework, as described in Chapter 5.

For end-to-end guarantees, resource management within the communication subsystem must be integrated with that for applications. The architecture proposed and analyzed in this chapter is directly applicable if a portion of the host processing capacity can be reserved for communication-related activities [107,123]. The proposed architectural extensions can be realized as a server with appropriate capacity reserves and/or execution priority. Our prototype implementation is indeed such a server executing in a standalone configuration. More importantly, our approach decouples protocol processing priority from that of the application. We believe that the protocol processing priority of a connection must be derived from the QoS requirements, traffic characteristics, and run-time communication behavior of the application on that connection.

We note that the proposed architecture and the admission control extensions described in Chapter 4 are also applicable to application-level framing [41] and user-level protocol processing architectures explored in recent efforts [55,112,167] to improve data transfer throughput in high-speed networks. In our design approach we have not made any specific assumptions about the location of the protocol stack, which could reside in the kernel or in user space. Architectural features such as CPU scheduling of channel handlers and cooperative preemption can be utilized irrespective of the address space and protection domain. With user-level protocol processing, however, allocation of communication resources is coupled directly with allocation of computation resources, and as such the admission control and run-time resource management must be more comprehensive. Regardless of the location of the protocol stack, realization of QoS guarantees in practice necessitates architectural mechanisms and admission control extensions similar in nature to the ones considered in this and the next chapter, respectively. The issues and mechanisms for integrating the proposed architecture with QoS-sensitive application scheduling and/or processor capacity reserves are discussed in Chapter 8.

The architectural framework and methodology we have adopted can be applied to other host platforms and networking technologies. Since provision of QoS guarantees is platform-specific, specific instances of this architecture depend on the CPU and network capacities of a platform. Implementing this architecture requires that the host communication subsystem be parameterized accurately to capture overheads and processing costs that comprise the abstraction of the underlying communication subsystem. The admission control extensions and run-time management support can then be re-targeted for a given host platform and/or networking technology. Chapter 6 addresses the issues of portability and accurate parameterization of QoS-sensitive communication subsystems. While we have implemented the architecture on an x-kernel platform, the architecture and the implementation do not utilize any features specific to this platform. The underlying resource management policies can be supported on any operating system platform.

The remainder of this chapter is organized as follows. We first discuss the architectural requirements for guaranteed-QoS communication and provides a brief description of real-time channels. Next we present a QoS-sensitive communication subsystem architecture realizing these requirements followed by a description of our prototype implementation of this architecture. We then present results demonstrating the efficacy of the proposed architecture via an experimental evaluation of the implementation. Finally, we conclude the chapter after summarizing the key contributions and highlighting some directions for future work.

3.2 Architectural Requirements for Guaranteed-QoS Communication

For guaranteed-QoS communication, we consider unidirectional data transfer, from source to sink via intermediate nodes, with data being delivered at the sink in the order in which it is generated at the source. Corrupted, delayed, or lost data is of little value; with a continuous flow of time-sensitive data, there is insufficient time for error recovery. Thus, we consider data transfer with unreliable-datagram semantics with no acknowledgements and retransmissions. To provide per-connection QoS guarantees, host communication resources must be managed in a QoS-sensitive fashion, i.e., according to the relative importance of the connections requesting service. Host communication resources include CPU bandwidth for protocol processing, link bandwidth for packet transmissions and reception, and buffer space.

A QoS-sensitive communication subsystem must provide services for the two related aspects of guaranteed-QoS communication [8], namely, traffic specification and resource management. Figure 3.1 illustrates a generic software architecture for guaranteed-QoS communication services at the host. The components constituting this architecture are briefly discussed below.

Application programming interface (API): The API must export routines that can be used to specify traffic and QoS requirements, set up and teardown guaranteed-QoS connections, and perform data transfer on these connections.

Signalling and admission control: A signalling protocol is required to establish/tear down guaranteed-QoS connections across the communicating hosts, possibly via multiple network nodes. The communication subsystem must keep track of communication resources, perform admission control on new connection requests, and establish *connection state* to store connection specific information.

Network data transport: Protocols are needed for unidirectional (reliable and unreliable) data transfers, including fragmentation (reassembly) of application data into (from) smaller units (packets) for network transmission (reception).

Traffic enforcement: Traffic enforcement forces an application to conform to its traffic specification and provide overload protection between established connections. This is required at the session level, and may be required at the link level depending on the nature of the traffic violation; link level traffic enforcement may be required at receiving hosts.

Link access scheduling and link abstraction: Link bandwidth must be managed such that all active connections receive their promised QoS. This necessitates abstracting the link in terms of transmission delay and bandwidth, and scheduling all outgoing packets for network access. The minimum requirement for provision of QoS guarantees is that packet transmission time on the link be bounded and predictable.

Assuming support for signalling, our primary focus in this chapter is on the components involved in data transfer, namely, traffic enforcement, protocol processing and link transmission. In particular, we study architectural mechanisms for structuring host communication software to provide QoS guarantees. Issues in admission control are dealt with

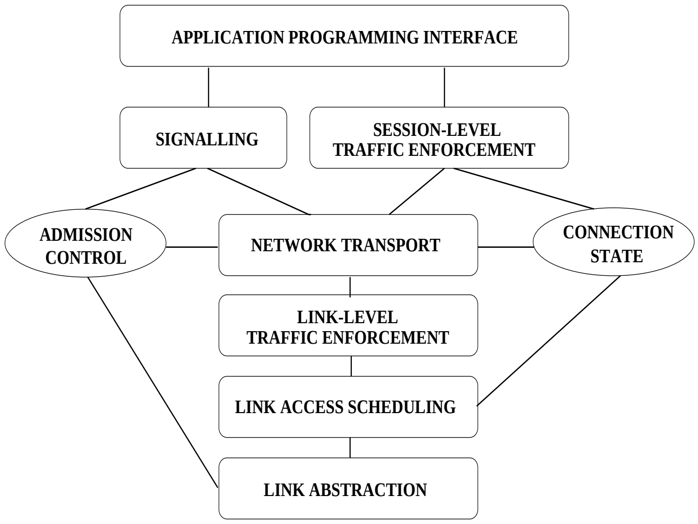


Figure 3.1: Desired overall software architecture.

in Chapter 4.

3.2.1 Software Structure for QoS-Sensitive Data Transport

In Figure 3.2, an application presents the API with data (*messages*) to be transported on a guaranteed-QoS connection. The API must allocate buffers for this data and queue it appropriately. Data that is conformant (as per the traffic specification) is forwarded for protocol processing and transmission.

Maintenance of per-connection QoS guarantees: Protocol processing involves, at the very least, fragmentation of application messages, including transport and network layer encapsulation, into *packets* with length smaller than a certain maximum (typically the MTU of the attached network). Additional computationally intensive services such as coding, compression, or checksums may also be performed during protocol processing. QoS-sensitive allocation of processing bandwidth necessitates multiplexing of the CPU amongst active connections under control of the CPU scheduler, which must provide deadline-based or priority-based policies for scheduling protocol processing on individual connections.

Non-preemptive protocol processing on a connection implies that the CPU can be reallocated to another connection only after processing an entire message, resulting in a coarser temporal grain of multiplexing and making admission control less effective. More importantly, admission control must consider the largest possible message size (maximum number of bytes presented by the application in one request) across *all* connections, including best-

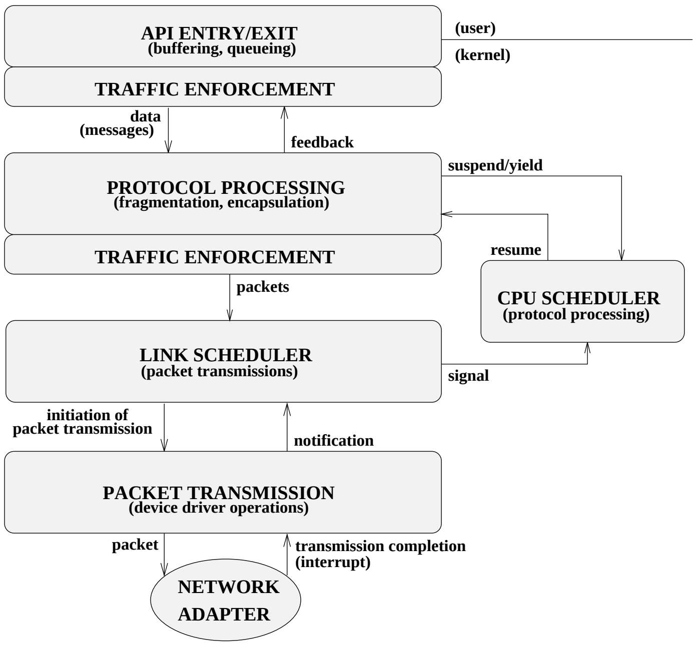


Figure 3.2: Software structure for QoS-sensitive data transmission.

effort traffic. While maximum message size for guaranteed-QoS connections can be derived from application-level attributes such as frame size for multimedia applications, the same for best-effort traffic may not be known *a priori*. Accordingly, mechanisms to suspend and resume protocol processing on a connection are needed. Protocol processing on a connection may also need to be suspended if there are no packet buffers available for that connection.

The packets generated via protocol processing cannot be directly transmitted on the link as that would result in FIFO (i.e., QoS-insensitive) consumption of link bandwidth. Instead, they are forwarded to the link scheduler, which must provide QoS-sensitive policies for scheduling packet transmissions. The link scheduler selects a packet and initiates packet transmission on the network adapter. Notification of packet transmission completion is relayed to the link scheduler so that another packet can be transmitted. The link scheduler must signal the CPU scheduler to resume protocol processing on a connection that was suspended earlier due to shortage of packet buffers.

Overload protection via per-connection traffic enforcement: As mentioned earlier, only conformant data is forwarded for protocol processing and transmission. This is necessary since QoS guarantees are based on a connection's traffic specification; a connection violating its traffic specification should not be allowed to consume communication resources over and above those reserved for it. Traffic specification violations on one connection should not affect QoS guarantees on other connections and the performance delivered to best-effort traffic. Accordingly, the communication subsystem must police per-connection traffic; in general, each parameter constituting the traffic specification (e.g., rate, burst length) must be policed individually. An important issue is the handling of non-conformant traffic, which could be buffered (shaped) until it is conformant, provided with degraded QoS, treated as best-effort traffic, or dropped altogether. Under certain situations, such as buffer overflows, it may be necessary to block the application until buffer space becomes available, although this may interfere with the timing behavior of the application. The most appropriate policy, therefore, is application-dependent.

Buffering non-conformant traffic till it becomes conformant makes protocol processing *non-work-conserving* since the CPU idles even when there is work available; the above discussion corresponds to this option. Alternately, protocol processing can be *work-conserving*, with CPU scheduling mechanisms ensuring QoS-sensitive allocation of CPU bandwidth to connections. Work-conserving protocol processing can potentially improve CPU utilization, since the CPU does not idle when there is work available. While the unused capacity can be utilized to execute other best-effort activities (such as background computations), one can also utilize this CPU bandwidth by processing non-conformant traffic, if any, assuming there is no pending best-effort traffic. This can free up CPU processing capacity for subsequent messages. In the absence of best-effort traffic, work-conserving protocol processing can also improve the average QoS delivered to individual connections, especially if link scheduling is work-conserving.

Fairness to best-effort traffic: Best-effort traffic includes data transported by conventional protocols such as TCP and UDP, and signalling for guaranteed-QoS connections. Best-effort traffic should not be unduly penalized by non-conformant real-time traffic, especially under work-conserving processing.

3.2.2 Real-Time Channels: A Model for Guaranteed-QoS Communication

Several models have been proposed for guaranteed-QoS communication in packet-switched networks [8]. While the architectural mechanisms proposed in this paper are applicable to most of the proposed models, we focus on *real-time channels* [63, 92], using the model proposed and analyzed in [92]. A real-time channel is a simplex, fixed-route, virtual connection between a source and destination host, with sequenced messages and associated performance guarantees on message delivery. It therefore conforms to the connection semantics mentioned earlier. A real-time channel can be viewed as a pipe between a data source and a data sink, with unidirectional data flow from the source to the sink.

An application requests a real-time channel by specifying its QoS requirements and traffic characteristics, as outlined in Chapter 1. During admission control, the resources required to satisfy the application's request are computed based on the specified worst-case traffic, and the request accepted if sufficient resources can be reserved for it. A local bound, which determines the worst-case transit delay seen by a packet on this channel plus a certain slack, is assigned to each node. Once the channel is successfully established, the communication subsystem and the network maintain QoS guarantees via appropriate resource management and traffic enforcement policies. When the application requests that the channel be destroyed, all resources allocated for the channel are released by the network and the communication subsystems at the source and destination hosts.

Traffic and QoS Specification: Traffic generation on real-time channels is based on a *linear bounded arrival process* [4,44] characterized by three parameters: maximum message size (M_{max} bytes), maximum message rate (R_{max} messages/second), and maximum burst size (B_{max} messages). In any interval of length δ, the number of messages generated is bounded by $B_{max}+\delta\cdot R_{max}$. Message generation rate is bounded by R_{max}, and its reciprocal, I_{min}, is the minimum inter-generation time between messages. The burst parameter B_{max} bounds the allowed short-term variation in message generation, and partially determines the buffer space requirement of the real-time channel. The notion of *logical arrival time* is used to enforce a minimum separation $I_{min} = \frac{1}{R_{max}}$ between messages on the same real-time channel. This ensures that a channel does not use more resources than it reserved at the expense of other channels. The logical arrival time, $\ell(m)$, of a message m is defined as:

$$\begin{aligned} \ell(m_0) &= t_0 \\ \ell(m_i) &= max\{(\ell(m_{i-1}) + I_{min}),\ t_i\}, \end{aligned}$$

where t_i is the actual generation time of message m_i; $\ell(m_i)$ is the time at which m_i would have arrived (generated) if the R_{max} constraint was strictly obeyed.

The QoS on a real-time channel is specified as the desired deterministic, worst-case bound on the end-to-end delay experienced by a message. ;the delay bound is, therefore, specified independent of the desired bandwidth. If d is the desired end-to-end delay bound for a channel, message m_i generated at the source is guaranteed to be delivered at the sink by time $\ell(m_i) + d$. See [92] for more details.

Resource Management: As with other proposals for guaranteed-QoS communication [8], there are two related aspects to resource management for real-time channels: admission control and (run-time) scheduling. Admission control for real-time channels is provided by Algorithm `D_order` [92], which uses fixed-priority scheduling for computing the worst-case delay experienced by a channel at a link. When a channel is to be established at a link, the worst-case response time for a message (when the message completes transmission on the link) on this channel is estimated based on non-preemptive fixed-priority scheduling of packet transmissions. The total response time, which is the sum of the response times over all the links on the route of the channel, is checked against the maximum permissible message delay and the channel established only if the latter is greater. A local delay bound is derived from the worse-case response time and the specified end-to-end delay bound. The priority assignment algorithm ensures that the new channel does not affect the QoS promised to existing channels. `D_order` assumes that the worst-case delay at each link for any channel does not exceed its message inter-arrival time. The total end–to–end delay, however, can exceed the message inter-arrival time of the channel.

Contrary to the approach for admission control, run-time link scheduling is governed by a variation of the multi-class earliest-deadline-first (EDF) policy [111]. The above approach only accounts for management of link bandwidth at the host. As shown in [116], and discussed in the next chapter, it cannot be applied directly to CPU bandwidth management.

3.2.3 Performance Related Considerations

To provide deterministic QoS guarantees on communication, all processing costs and overheads involved in managing and using resources must be accounted for. Processing costs include the time required to process and transmit a message, while the overheads include preemption costs such as context switches and cache misses, costs of accessing ordered data structures, and handling of network interrupts. It is important to keep the overheads low and predictable (low variability) so that reasonable worst-case estimates can be obtained.

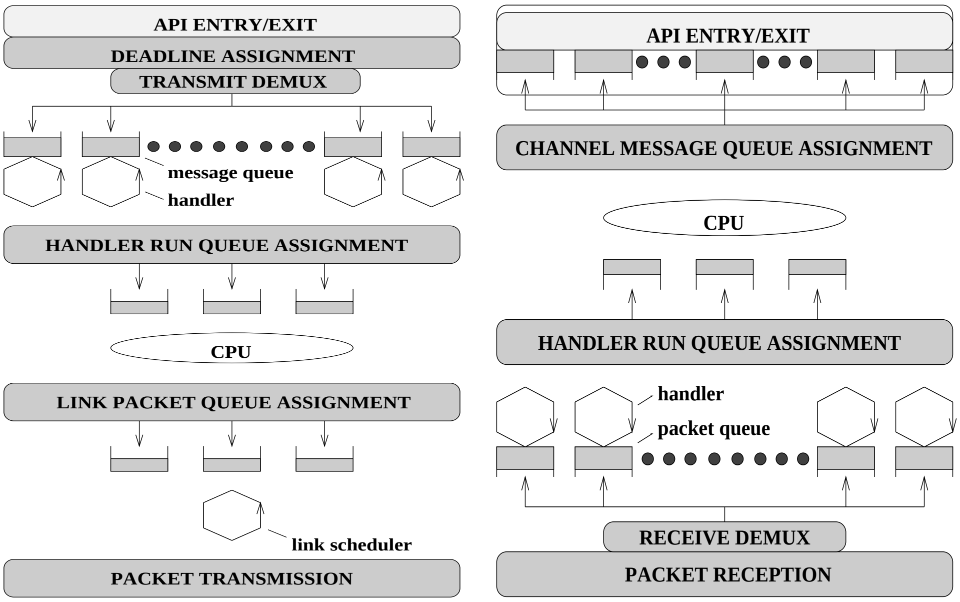
An important performance metric is scalability, i.e., the number of guaranteed-QoS connections that can be serviced at the host. Resource management policies must maximize the number of connections accepted for service. In addition to processing costs and implementation overheads, factors that affect admissibility include the relative bandwidths of the CPU and link and any coupling between CPU and link bandwidth allocation. In the next chapter we study the extent to which these factors affect admissibility in the context of real-time channels.

3.3 QoS-Sensitive Communication Subsystem Architecture

In the *process-per-message* model [152], a process or thread shepherds a message through the protocol stack. Besides eliminating extraneous context switches encountered in the *process-per-protocol* model [152], it also facilitates protocol processing to be scheduled according to a variety of policies, as opposed to the software-interrupt level processing in BSD Unix. However, the process-per-message model introduces additional complexity for supporting per-channel QoS guarantees.

Creating a distinct thread to handle each message makes the number of active threads a function of the number of messages awaiting protocol processing on each channel. Not only does this consume kernel resources (such as process control blocks and kernel stacks), but it also increases scheduling overheads which are typically a function of the number of runnable threads in dynamic scheduling environments. More importantly, with a process-per-message model, it is relatively harder to maintain channel semantics, provide QoS guarantees, and perform per-channel traffic policing. For example, bursts on a channel get translated into "bursts" of processes in the scheduling queues, making it harder to police ill-behaved channels and ensure fairness to best-effort traffic. Further, scheduling overhead becomes unpredictable, making worst-case estimates either overly conservative or impossible to provide.

Since QoS guarantees are specified on a per-channel basis, it suffices to have a single thread coordinate access to resources for all messages on a given channel. We employ a *process-per-channel* model, which is a QoS-sensitive extension of the *process-per-connection* model [152]. In the process-per-channel model, protocol processing on each channel is coordinated by a unique *channel handler*, a lightweight thread created on successful establishment of the channel. With unique per-channel handlers, CPU scheduling overhead is only a function of the number of *active* channels, those with messages waiting to be transported. Since the number of established channels, and hence the number of active channels,



(a) Source host (b) Destination host

Figure 3.3: Proposed communication subsystem architecture.

varies much more slowly compared to the number of messages outstanding on all active channels, CPU scheduling overhead is significantly more predictable. As we discuss later, a process-per-channel model also facilitates per-channel traffic enforcement. Further, since it reduces context switches and scheduling overheads, this model is likely to provide good performance to connection-oriented best-effort traffic.

Figure 3.3 depicts the key components of the proposed architecture at the source (transmitting) and destination (receiving) hosts; only the components involved in data transfer are illustrated. Associated with each channel is a *message queue*, a FIFO queue of messages to be processed by the channel handler (at the source host) or to be received by the application (at the destination host). Each channel also has associated with it a *packet queue*, a FIFO queue of packets waiting to be transmitted by the link scheduler (at the source host) or to be reassembled by the channel handler (at the destination host).

Transmission-Side Processing: In Figure 3.3(a), invocation of message transmission transfers control to the API. After traffic enforcement (traffic shaping and deadline assignment), the message is enqueued onto the corresponding channel's message queue for subsequent processing by the channel handler. Based on channel type, the channel handler is assigned to one of three CPU run queues for execution (described in Section 3.3.1). It executes in an infinite loop, dequeueing messages from the message queue and performing

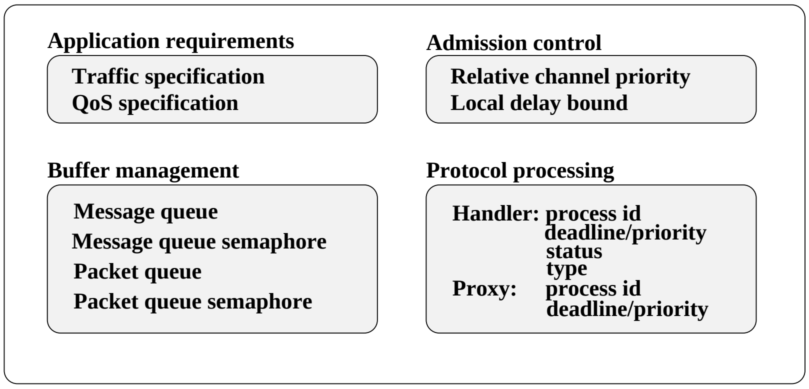


Figure 3.4: Channel state maintained at host.

protocol processing (including fragmentation). The packets thus generated are inserted into the channel packet queue and into one of three (outbound) *link packet queues* for the corresponding link, based on channel type and traffic generation, to be transmitted by the link scheduler.

Reception-Side Processing: In Figure 3.3(b), a received packet is demultiplexed to the corresponding channel's packet queue, for subsequent processing and reassembly. As in transmission-side processing, channel handlers are assigned to one of three CPU run queues for execution, and execute in an infinite loop, waiting for packets to arrive in the channel packet queue. Packets in the packet queue are processed and transferred to the channel's reassembly queue. Once the last packet of a message arrives, the channel handler completes message reassembly and inserts the message into the corresponding message queue. The application retrieves the message from the message queue by invoking the API's receive routine.

At intermediate nodes, the link scheduler relays arriving packets to the next node along the route. In the following discussion, we focus on transmission-side processing at the sending host. Much of this discussion is also applicable to reception-side processing. The issues involved in QoS-sensitive handling of incoming traffic are discussed in Chapter 4, and implementation of the receive-side architecture considered in Chapter 5.

3.3.1 Salient Features

Figure 3.4 illustrates a portion of the state associated with a channel at the host upon successful establishment. In addition to application requirements, channel state includes parameters associated with admission control, data structures associated with buffer man-

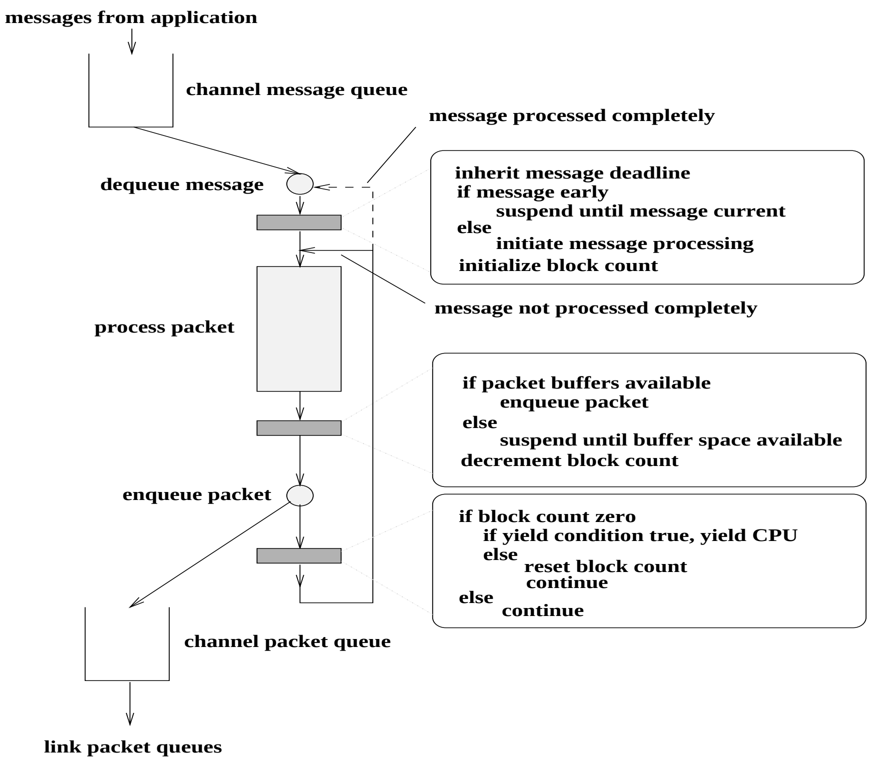


Figure 3.5: Execution profile of a channel handler.

agement, and attributes associated with protocol processing. Each channel is assigned a priority relative to other channels, as determined by the admission control procedure. The local delay bound computed during admission control at the host is used to compute deadlines of individual messages. Each handler is associated with a type, and execution deadline or priority, and execution status (runnable, blocked, etc.). In addition, two semaphores are allocated to each channel handler, one to synchronize with message insertions into the channel's message queue (the *message queue semaphore*), and the other to synchronize with availability of buffer space in the channel's packet queue (the *packet queue semaphore*).

Channel handlers are broadly classified into two types, best-effort and real-time. A best-effort handler is one that processes messages on a best-effort channel. Real-time handlers are further classified as *current* real-time and *early* real-time. A current real-time handler is one that processes *on-time* messages (obeying the channel's rate specification), while an early real-time handler is one that processes *early* messages (violating the channel's rate specification).

Figure 3.5 shows the execution profile of a channel handler at the source host. As long as messages are available, the handler executes in an infinite loop processing messages one at a time. When initialized, it simply waits for messages to process from the message queue. Once a message becomes available, the handler dequeues the message and inherits its deadline. If the message is early, the handler computes the time until the message will become current and suspends execution for that duration. If the message is current, the handler initiates protocol processing of the message. After creating each packet, the handler checks for space in the packet queue (via the packet queue semaphore); it is automatically blocked if space is not available and is woken up when space becomes available subsequently.

The packets created are enqueued onto the channel's packet queue, and if the queue was previously empty, the link packet queues are also updated to reflect that this channel has packets to transmit. That is, only the head packet from the channel's packet queue resides in the ordered link packet queues at any given time. When a packet from this channel completes transmission, another packet is transferred from the channel packet queue to the link packet queues. This design incurs a worst-case packet enqueueing overhead proportional to the number of active channels, instead of the total number of packets outstanding on all active channels. The overhead of managing per-channel packet queues is, therefore, minimal, assuming that the device driver can classify a packet (i.e., identify the corresponding channel packet queue) without any extra overhead. This is true in our architecture since only one packet is kept outstanding on the network adapter. The preemption model employed for handler execution is one of *cooperative preemption*; the currently-executing handler relinquishes the CPU to a waiting higher-priority handler after processing a block of packets, as explained below.

While the above suffices for non-work-conserving protocol processing, a mechanism is needed to continue handler execution in the case of work-conserving protocol processing. Accordingly, in addition to blocking the handler as before, a *channel proxy* is created on behalf of the handler. A channel proxy is a thread that simply signals the (blocked) channel handler to resume execution. It competes for CPU access with other channel proxies in the order of logical arrival time, and exits immediately if the handler has already woken up. This mechanism ensures that the handler is made runnable if the channel proxy obtains access to the CPU before the handler becomes current. Note that an early handler must still relinquish the CPU to a waiting handler that is already current.

Maintenance of QoS guarantees

Per-channel QoS guarantees are provided via appropriate preemptive scheduling of channel handlers and non-preemptive scheduling of packet transmissions. While CPU scheduling can be priority-based (using relative channel priorities), we consider deadline-based scheduling for channel handlers and proxies. Execution deadline of a channel handler is inherited dynamically from the deadline of the message to be processed. Execution deadline of a channel proxy is derived from the logical arrival time of the message to be processed. Channel handlers are assigned to one of two run queues based on their type (best-effort or real-time), while channel proxies (representing early real-time traffic) are assigned to a separate run queue. The relative priority assignment for handler run queues is such that on-time real-time traffic gets the highest protocol processing priority, followed by best-effort traffic and early real-time traffic in that order.

Provision of QoS guarantees necessitates bounded delays in obtaining the CPU for protocol processing. As shown in the next chapter, immediate preemption of an executing lower-priority handler results in significant overheads due to context switches and cache misses; channel admissibility is significantly improved if preemption overheads are amortized over the processing of several packets. The maximum number of packets processed in a block is a system parameter determined via experimentation on a given host architecture. Cooperative preemption provides a reasonable mechanism to bound CPU access delays while improving utilization, especially if all handlers execute within a single (kernel) address space.

Link bandwidth is managed via multi-class non-preemptive EDF scheduling with link packet queues organized similar to CPU run queues. Link scheduling is non-work-conserving to avoid stressing resources at downstream hosts; in general, the link is allowed to "work ahead" in a limited fashion, as determined by the link *horizon* [92].

Overload protection

Per-channel traffic enforcement is performed when new messages are inserted into the message queue, and again when packets are inserted into the link packet queues. The per-channel message queue absorbs message bursts on a channel, preventing violations of B_{max} and R_{max} on this channel from interfering with other, well-behaved channels. During deadline assignment, new messages are checked for violations in M_{max} and R_{max}. Before inserting each message into the message queue, the inter-message spacing is enforced according to I_{min}. For violations in M_{max}, the (logical) inter-arrival time between messages is increased in proportion to the extra packets in the message.

The number of packet buffers available to a channel is determined by the product of the maximum number of packets constituting a message (derived from M_{max}) and the maximum allowable burst length B_{max}. Under work-conserving processing, it is possible that the packets generated by a handler cannot be accommodated in the channel packet queue because all the packet buffers available to the channel are exhausted. A similar situation could arise in non-work-conserving processing with violations of M_{max}. Handlers of such violating channels are prevented from consuming excess processing and link capacity, either by blocking their execution or lowering their priority relative to well-behaved channels.

Blocked handlers are subsequently woken up when the link scheduler indicates availability of packet buffers. Blocking handlers in this fashion is also useful in that a slowdown in the service provided to a channel propagates up to the application via the message queue. Once the message queue fills up, the application can be blocked until additional space becomes available. Alternately, messages overflowing the queue can be dropped and the application informed appropriately. Note that while scheduling of handlers and packets provides isolation between traffic on different channels, interaction between the CPU and link schedulers helps police per-channel traffic.

Fairness to best-effort traffic

Under non-work-conserving processing, early real-time traffic does not consume any resources at the expense of best-effort traffic. With work-conserving processing, best-effort traffic is given processing and transmission priority over early real-time traffic.

3.3.2 Accounting for CPU Preemption Delays and Overheads

The admission control procedure (`D_order`) must account for CPU preemption overheads, access delays due to cooperative preemption, and other overheads involved in managing resources. For each new channel to be admitted, `D_order` computes *message service time*, the worst-case time for which the CPU and link must be allocated to the channel for processing a message, and *wait time*, the worst-case time spent waiting for a lower-priority handler to relinquish the CPU and link. Accordingly, it must also account for the overlap

$\mathcal{C}_{sw}$	time to switch contexts between channel handlers (EDF)
$\mathcal{C}_{cm}$	penalty due to cache misses resulting from a context switch
$\mathcal{C}_p^{1st}$	CPU processing cost for the first packet
$\mathcal{C}_p$	CPU processing cost for packets other than the first packet
$\mathcal{C}_l$	per-packet link scheduling cost
$\mathcal{P}$	number of packets processed between preemption points
$\mathcal{S}$	maximum packet size in bytes
$\mathcal{L}_x(s)$	packet transmission time for packet size s

Figure 3.6: Important system parameters in the proposed architecture.

between CPU processing and link transmission, and hence the relative bandwidths of the CPU and link. The deadlines assigned to messages are derived from the worst-case service and wait times computed by `D_order`. The next chapter presents extensions to `D_order` to account for the above-mentioned factors.

These extensions are derived in terms of certain *system parameters* that together comprise the abstraction of the underlying communication subsystem. These system parameters represent the overheads and processing costs of the target implementation platform. Of these parameters, only $\mathcal{P}$ and $\mathcal{S}$ can be explicitly configured up to a certain extent. The other parameters relate to the host CPU and memory architecture, and hence are fixed for a given platform, protocol stack, and operating system. $\mathcal{P}$ determines the granularity at which the CPU is multiplexed between channels, while $\mathcal{S}$ determines the multiplexing granularity of the link; the choice of these parameters therefore determines channel admissibility at the host, as demonstrated in the next chapter. Below we briefly discuss the factors that influence determination of $\mathcal{P}$ and $\mathcal{S}$. We also discuss determination of $\mathcal{L}_x$ since we are interested in exploring the relationship between CPU and link bandwidth.

Packets between preemptions: Selection of $\mathcal{P}$ is governed by the architectural characteristics of the host CPU, as captured by the parameters listed in Table 4.1. For a given message (and packet) size, small values of $\mathcal{P}$ imply a higher number of preemptions, increasing the total overhead incurred and reducing the CPU bandwidth available to channel handlers; this in turn reduces channel admissibility. Large values of $\mathcal{P}$, on the other hand, increase the temporal granularity at which the CPU is multiplexed between channel handlers and hence the window of non-preemptibility. This may also reduce the number of channels admitted for service. For a given host architecture, $\mathcal{P}$ can be selected such that channel admissibility is maximized while delivering reasonable data transfer throughput.

Packet size: $\mathcal{S}$ can be selected either using end-to-end transport protocol performance or host/adapter design characteristics. End-to-end protocol performance has been used to determine packet size in IP and IP-over-ATM networks for optimum TCP performance. However, since data transfer on real-time channels is unidirectional and unreliable, end-to-end protocol performance may not be the best guide for selection of $\mathcal{S}$. A particular choice for $\mathcal{S}$ determines the number of packets constituting a message, and hence total CPU and link bandwidth required to process and transmit it. In general, the latency and throughput characteristics of the adapter as a function of packet size can be used to pick a packet size that minimizes $\mathcal{L}_x$ (see below) while delivering reasonable data transfer throughput. However, the value chosen for $\mathcal{S}$ must be less than the MTU of the attached network. Note that in channels spanning heterogeneous networks, $\mathcal{S}$ can be different at each hop, as long as the cost of additional fragmentation within the network is accounted for when determining end-to-end delays.

Packet transmission time: For a typical network adapter, the transmission time for a packet of size s, $\mathcal{L}_x(s)$, depends primarily on the overhead of initiating transmission and the time to transfer the packet to the adapter and on the link. The latter is a function of the packet size and the data transfer bandwidth available between host and adapter memories. The data transfer bandwidth itself is determined by host/adapter design features such as pipelining, on-board queueing on the adapter, and the raw link bandwidth. If $\mathcal{C}_x$ is the overhead to initiate transmission on an adapter feeding a link of bandwidth $\mathcal{B}_l$ bytes/second, $\mathcal{L}_x(s)$ can be approximated as

$$\mathcal{L}_x(s) = \mathcal{C}_x + \frac{s}{\min(\mathcal{B}_l, \mathcal{B}_x)},$$

where $\mathcal{B}_x$ is the data transfer bandwidth available to/from host memory. $\mathcal{B}_x$ is determined by factors such as the mode (direct memory access (DMA) or programmed IO) and efficiency of data transfer, and the degree to which the adapter pipelines packet transmissions. $\mathcal{C}_x$ includes the cost of setting up DMA transfer operations, if any. Our experience with adapter design and the implications for packet transmission time are highlighted in [77].

Routines	Invoked By	Function Performed
`rtc_init`	receiving task	create local queue to receive messages
`rtc_create`	sending task	create real-time channel with given parameters to remote end point (queue); return channel ID
`rtc_send`	sending task	send message on the specified real-time channel
`rtc_recv`	receiving task	receive message from real-time message queue
`rtc_close`	sending task	close specified real-time channel

Table 3.1: Routines constituting the real-time channel API.

3.4 Prototype Implementation

We have implemented the proposed architecture using a communication executive derived from x-kernel (v3.1) [76] that exercises complete control over a 25 MHz Motorola 68040 CPU. Accordingly, CPU bandwidth is consumed only by communication-related activities, facilitating admission control and resource management for real-time channels.[1] x-kernel (v3.1) employs a process-per-message protocol-processing model and a priority-based non-preemptive scheduler with 32 priority levels; the CPU is allocated to the highest-priority runnable thread, while scheduling within a priority level is FIFO.

3.4.1 Architectural Configuration

Real-time communication is accomplished via a connection-oriented protocol stack in the communication executive (see Figure 3.7). The API exports routines for real-time channel establishment, channel teardown, and data transfer (see Table 3.1); it also supports routines for best-effort data transfer (not shown here). Network transport for signalling is provided by a (resource reservation) protocol layered on top of a remote procedure call (RPC) protocol derived from x-kernel's CHAN protocol. Network transport for data is provided by a fragmentation (FRAG) protocol, which packetizes large messages so that communication resources can be multiplexed between channels on a packet-by-packet basis. The FRAG transport protocol is a modified, unreliable version of x-kernel's BLAST protocol with timeout and data retransmission operations disabled. The protocol stack also provides

[1] Implementation of the reception-side architecture is a slight variation of the transmission-side architecture.

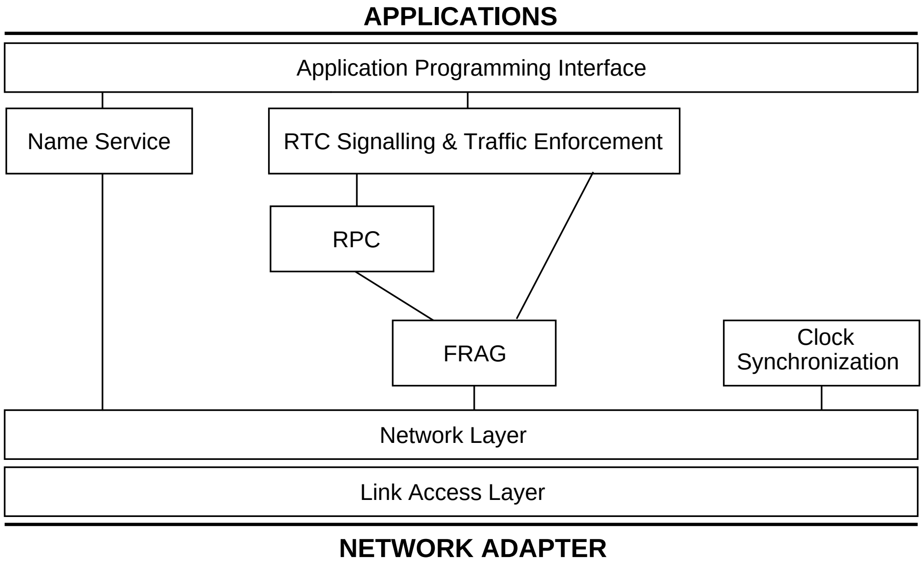


Figure 3.7: Real-time communication protocol stack (*x*-kernel).

protocols for clock synchronization and network layer encapsulation. The network layer protocol is connection-oriented and provides network-level encapsulation for data transport across a point-to-point communication network. The link access layer provides support for link scheduling and includes the network device driver.

Our choice of protocols was based on the perceived requirements for guaranteed-QoS communication. An alternative approach would be to utilize the TCP/IP suite of protocols used on the Internet. Most TCP/IP stacks do not provide a sequenced, unreliable message transport protocol that supports fragmentation. TCP is a reliable byte-stream protocol while UDP does not fragment outbound messages or support message sequencing. Thus, while TCP is suitable for transporting best-effort traffic, it is not suitable for guaranteed-QoS communication. Further, IP is a connectionless protocol and would require either modifications to make it connection-oriented or mechanisms to classify packets. Note that the proposed architecture does not preclude employing TCP to transport best-effort traffic, but this would require IP as the network layer.

3.4.2 Realizing a QoS-Sensitive Architecture

Process-per-channel model: On successful establishment, a channel is allocated a channel handler, space for its message and packet queues, and the message and packet queue semaphores. If work-conserving protocol processing is desired, a channel proxy is also allocated to the channel. A channel handler is an x-kernel process (which provides its thread of control) with additional attributes such as the type of channel (best-effort or real-time), flags encoding the state of the handler, its execution priority or deadline, and an event identifier corresponding to the most recent x-kernel timer event registered by the handler. In order to suspend execution until a message is current, a handler utilizes x-kernel's timer event facility and an *event semaphore* which is signaled when the timer expires. A channel proxy is also an x-kernel process with an execution priority or deadline. The states of all established channels are maintained in a linked list that is updated during channel signalling.

We extended x-kernel's process management and semaphore routines to support handler creation, termination, and synchronization with events such as message insertions and availability of packet buffers after packet transmissions. Each packet of a message must inherit the transmission deadline assigned to the message. We modified the BLAST protocol and message manipulation routines in x-kernel to associate the message deadline with each packet. Each outgoing packet carries a global channel identifier, allowing efficient packet demultiplexing at a receiving node.

QoS-sensitive CPU scheduling: Two policies are available for scheduling channel handlers on the CPU: (i) multi-class EDF scheduling and (ii) fixed-priority scheduling with 32 priority levels. The following discussion applies to (i). Three distinct run queues are maintained for channel handlers, one for each of the three classes mentioned in Section 3.3.1, similar to the link packet queues. Q1 is a priority queue implemented as a heap ordered by handler deadline while Q2 is implemented as a FIFO queue. Q3, utilized only when the protocol processing is work-conserving, is a priority queue implemented as a heap ordered by the logical arrival time of the message being processed by the handler. Channel proxies are also realized as x-kernel threads and are assigned to Q3. Since Q3 has the lowest priority, proxies do not interfere with the execution of channel handlers.

The multi-class EDF scheduler is layered above the x-kernel scheduler, as illustrated in Figure 3.8. When a channel handler or proxy is selected for execution from the EDF run queues, the associated x-kernel process is inserted into a designated x-kernel priority level for CPU allocation by the x-kernel scheduler. To realize this design, we modified x-kernel's context switch, semaphore, and process management routines appropriately. For example, a context switch between channel handlers involves enqueuing the currently-active handler

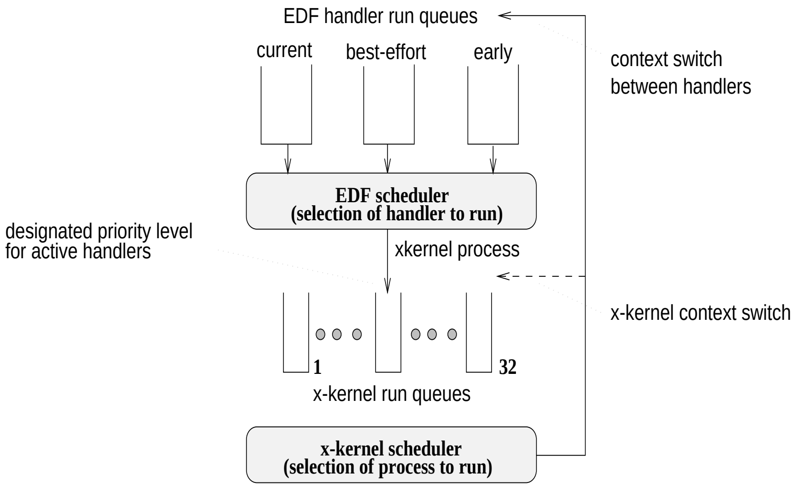


Figure 3.8: EDF CPU scheduler layered above native x-kernel scheduler.

in the EDF run queues and picking another runnable handler, before invoking the normal x-kernel code to switch process contexts. To support cooperative preemption, we added new routines to check the EDF and x-kernel run queues for waiting higher-priority handlers or native x-kernel processes, respectively, and yield the CPU accordingly.

QoS-sensitive Link scheduling: In order to support real-time communication, network adapters must provide a bounded, predictable transmission time for a packet of a given size. Since network adapters are typically best-effort in nature, their design is optimized for throughput and may be unsuitable for real-time communication, even with a bounded and predictable packet transmission time. Even when explicit support for real-time communication is provided, on-board buffer space limitations may necessitate staging of outgoing traffic in host memory, for subsequent transfer to the adapter.

To support real-time communication on these adapters, link scheduling must be provided in software on the host processor. In our implementation, packets created by channel handlers are scheduled for transmission by a non-preemptive multi-class EDF link scheduler. The implementation can be configured such that link scheduling is performed:

Option 1: via a function call by the currently executing handler or in interrupt context,

Option 2: by a dedicated process/thread, or

Option 3: by a new thread created after each packet transmission.

As shown in the next chapter, option 1 gives the best performance in terms of throughput and sensitivity of channel admissibility to $\mathcal{P}$ and $\mathcal{S}$; accordingly, we focus on option 1 in the discussion below.

1. `Mark the link as busy.`
2. `Examine Q3; transfer (via pointer manipulations) all packets that are current to Q1.`
3. `Transmit packet at` *head*`(Q1) if Q1 non-empty, else transmit packet at` *head*`(Q2).`
4. `If Q1 and Q2 are both empty, and packet at` *head*`(Q3) is not current, mark the link as idle, and register wakeup event with` x`-kernel for the time` *head*`(Q3) becomes current.`

Figure 3.9: Processing done by the link scheduler.

The organization of link packet queues is similar to that of handler run queues, except that Q3 is used for early packets when protocol processing is work-conserving. After inserting a packet into the appropriate link packet queue, channel handlers invoke the scheduler directly as a function call. If the link is busy, i.e., a packet transmission is in progress, the function returns immediately and the handler continues execution. If the link is idle, the processing shown in Figure 3.9 is performed. Scheduler processing is repeated when the network adapter indicates completion of packet transmission or the wakeup event for early packets expires. Additional packets can be kept outstanding on the network adapter as long as packet transmission time is bounded and predictable.

Per-channel traffic enforcement: A channel's message queue semaphore is initialized to B_{max}; messages overflowing the message queue are dropped. The packet queue semaphore is initialized to $B_{max} \cdot \mathcal{N}_{pkts}$, the maximum number of outstanding packets permitted on a channel. Upon completion of packet transmission, the corresponding channel's packet queue semaphore is signalled to indicate availability of packet buffers and enable execution of a blocked handler. If the overflow is due to a violation in M_{max}, the priority (or deadline) of the handler is degraded in proportion to the extra packets in its payload, so that further consumption of CPU bandwidth does not affect other well-behaved channels. Table 3.2 summarizes the policies and options available in the implementation.

3.4.3 System Parameterization

Figure 3.10(a) lists the system parameter settings for our implementation, determined via detailed performance profiling. Selection of $\mathcal{P}$ and $\mathcal{S}$ is based on the tradeoff between

Category	Available Policies	
Protocol processing model	process-per-channel	
	work-conserving	non-work-conserving
CPU scheduling	fixed-priority with 32 priority levels	
	multi-class earliest-deadline-first	
Handler execution	cooperative preemption with configurable number of packets between preemptions	
Link scheduling	multi-class EDF (options 1, 2 and 3)	
Overload protection	block handler, decay handler deadline, enforce I_{min}, drop overflow messages	

Table 3.2: Available policies in the prototype implementation.

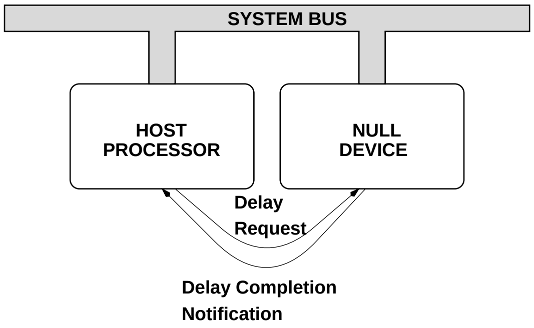
available resource capacity and channel admissibility [116], as discussed in the next chapter. To use the model of packet transmission time presented in Section 3.3.2, $\mathcal{C}_x$ and $\mathcal{B}_x$ must be determined for a given network adapter and host architecture. This in turn involves experimentally determining the latency-throughput characteristics of the adapter.

Using our implementation, a parameterization of the networking hardware available to us revealed significant performance-related deficiencies such as poor data transfer throughput and high, unpredictable packet transmission time [77]. Since these deficiences were due to adapter design, they severely limited our ability to demonstrate the capabilities of our architecture and implementation. Given our primary focus on *unidirectional* data transfer, it suffices to ensure that transmission of a packet of size s takes $\mathcal{L}_x(s)$ time units. This can be achieved by *emulating* the behavior of a network adapter such that $\mathcal{L}_x(s)$ time units are consumed for each packet being transmitted. We have implemented such a device emulator, referred to as the *null device*, that can be configured to emulate any desired packet transmission time $\mathcal{L}_x$ for a given packet size.

The device emulator is a thread that, once signalled, tracks time by consuming CPU resources for $\mathcal{L}_x(s)$ time units before signalling completion of packet transmission (see Figure 3.10(b)). This emulator is implemented on a separate processor that is connected via a backplane system bus to the processor implementing the communication subsystem (the host processor). Upon expiration of $\mathcal{L}_x(s)$ time units (i.e., completion of packet trans-

Symbol	Value	Unit
$\mathcal{C}_{sw}$	55	μs
$\mathcal{C}_{cm}$	90	μs
$\mathcal{C}_p^{1st}$	420	μs
$\mathcal{C}_p$	170	μs
$\mathcal{C}_l$	160	μs
$\mathcal{P}$	4	$packets$
$\mathcal{S}$	4096	$bytes$
$\mathcal{L}_x(\mathcal{S})$	245	μs

(a) System parameters for implementation



(b) Null network device

Figure 3.10: Implementation environment.

mission) the emulator issues an interrupt to the host processor, similar to the mechanism employed in typical network adapters. We experimentally determined $\mathcal{C}_x$ to be $\approx 40\mu s$. For the experiments reported in the next section, we select $\min(\mathcal{B}_l, \mathcal{B}_x)$ to correspond to a link (and data transfer) speed of 50 ns per byte. This corresponds to an effective packet transmission bandwidth (for 4KB packets) of 16 MB/s.

While the emulator allows us to study a variety of tradeoffs, including the effects of the relationship between CPU and link bandwidth, it is not completely accurate since no packet data is actually transferred from host memory. If packet data were transferred from host memory via DMA, there would be additional contention for the system bus and main memory, resulting in somewhat higher packet processing time and cache miss penalties upon resumption of execution after preemption. This would result in slightly optimistic estimates of the message service and wait times, and hence channel admissibility. However, a performance degradation of this nature would affect all real-time channels and best-effort traffic more or less equally, for option 1 as well as option 2.

Therefore, while the absolute performance observed may not be entirely accurate, the observed *trends* and performance *comparisons* reported continue to be valid. Also, it may be possible to extend the message service time computation to accurately account for the potential perturbation caused by the DMA transfers via careful analysis [74]. We note that at least two other efforts have employed such artificial sources and sinks of data, namely, the virtual network device in [14] that resides on a separate processor, and the "in-memory" network device used in [134].

3.5 Experimental Evaluation

We evaluate the efficacy of the proposed architecture in isolating real-time channels from each other and from best-effort traffic. The evaluation is conducted for a subset of the policies listed in Table 3.2, under varying degrees of traffic load and traffic specification violations. In particular, we evaluate the process-per-channel model with non-work-conserving multi-class EDF CPU scheduling and non-work-conserving multi-class EDF link scheduling using option 1 (Section 3.4.2). Overload protection for packet queue overflows is provided via blocking of channel handlers; messages overflowing the message queues are dropped. The parameter settings given in Figure 3.10(a) are used for the evaluation.

3.5.1 Methodology and Metrics

Using the null device, the performance of the proposed architecture is compared with and without features such as cooperative preemption and traffic enforcement. We choose a workload that stresses the resources on our platform, and is shown in Table 3.3. Similar results were obtained for other workloads, including a large number of channels with a wide variety of deadlines and traffic specifications. Message size is fixed at 60 KB; experiments with other message sizes reveal that our architecture is insensitive to message size. However, if we relax any of the constraints of our architecture, larger messages tend to introduce greater QoS violations. Three real-time channels are established (channel establishment here is strictly local) with different traffic specifications. Channels 0 and 1 are bursty while channel 2 is periodic in nature. Best-effort traffic is realized as channel 3, with a variable load depending on the experiment, and has similar semantics as the real-time traffic, i.e., it is unreliable with no retransmissions under packet loss.

Messages on each real-time channel are generated by an x-kernel process, running at the highest priority, as specified by a linear bounded arrival process with bursts of up to B_{max} messages. Violations in the specified rate (R_{max}) are realized by generating messages at rates that are multiples of R_{max}. The best-effort traffic generating process is similar, but runs at a priority lower than that of the real-time generating processes and higher than the

Channel	Type	Traffic Specification				Deadline
		M_{max} (KB)	B_{max} (messages)	R_{max} (KB/s)	I_{min} (ms)	(ms)
0	real-time (RT)	60	12	1200	50	40
1	real-time (RT)	60	8	2000	30	25
2	real-time (RT)	60	1	2000	30	30
3	best-effort (BE)	60	10	variable	–	–

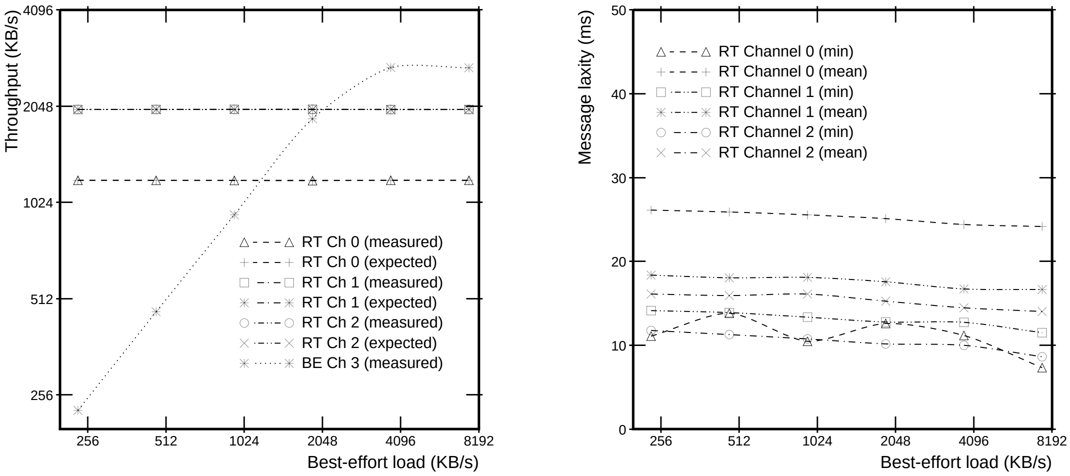
Table 3.3: Workload used for the evaluation.

x-kernel priority assigned to channel handlers. Each experiment's duration corresponds to the transmission of 32K packets; the first 2K and last 2K packets are ignored so that the evaluation is based on steady-state behavior. All the results are obtained after averaging over multiple runs; different runs consistently gave similar results, and hence low standard deviation.

Our evaluation focuses on the efficacy of the proposed architecture and the need for cooperative preemption. All the experiments reported here have traffic enforcement and deadline-based CPU and link scheduling enabled. We use the following metrics measuring per-channel performance. *Throughput* refers to the service received by each real-time channel and best-effort traffic. It is calculated by counting the number of packets successfully transmitted within the experiment duration. *Message laxity* is the difference between the transmission deadline of a real-time message and the actual time that it completes transmission. *Deadline misses* measures the number of real-time packets missing deadlines. Recall all packets of a message inherit the deadline of the message. Deadline misses are detected by checking the actual transmission time of a real-time packet against its deadline. Finally, *Packet drops* measures the number of packets dropped for both real-time and best-effort traffic. Deadline misses and packet drops account for the QoS violations on individual channels.

3.5.2 Efficacy of the Proposed Architecture

Figure 3.11 depicts the efficacy of the proposed architecture in maintaining QoS guarantees when all channels honor their traffic specifications. Figure 3.11(a) plots the throughput



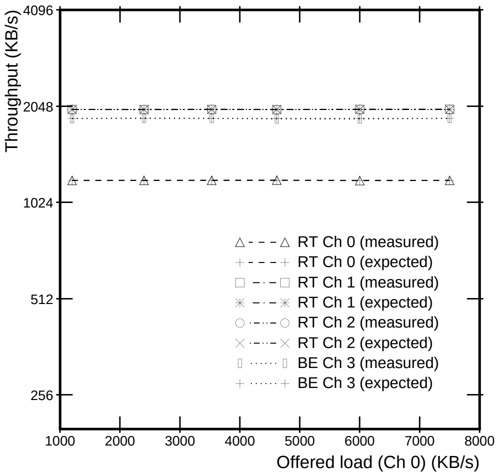
(a) Throughput (b) Message laxity

Figure 3.11: Maintenance of QoS guarantees when traffic specifications are honored.

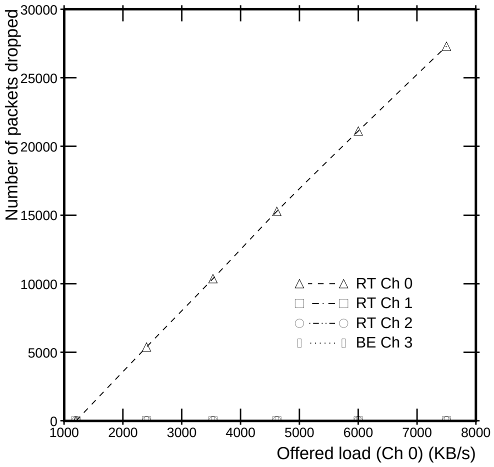
received by each real-time channel and best-effort traffic as a function of (offered) best-effort load. Several conclusions can be drawn from the observed trends. First, all real-time channels receive their desired level of throughput; since no packets were dropped (not shown here) or late (Figure 3.11(b)), the QoS requirements of all real-time channels are met. Increase in offered best-effort load has no effect on the service received by real-time channels. Second, the service received by best-effort traffic continues to increase linearly until the system capacity is exceeded. That is, real-time traffic (early as well as current) does not deny service to best-effort traffic. Third, even under extreme overload conditions, best-effort throughput saturates and declines slightly due to packet drops. However, performance of real-time traffic is not affected.

Figure 3.11(b) plots the message laxity for real-time traffic, also as a function of offered best-effort load. As can be seen, no messages miss their deadlines, since minimum laxity is non-negative for all channels. In addition, the mean laxity for real-time messages is largely unaffected by an increase in best-effort load, regardless of whether the channel is bursty or not.

Figure 3.12 demonstrates the same behavior even in the presence of traffic specification violations by real-time channels. Channel 0 generates messages at a rate faster than specified



(a) Throughput



(b) Number of packets dropped

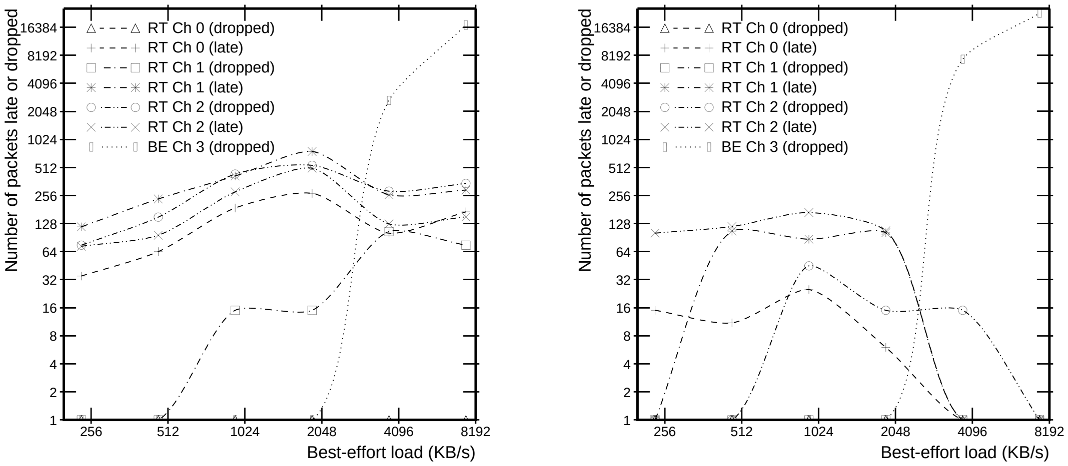
Figure 3.12: Maintenance of QoS guarantees under violation of R_{max}.

while best-effort traffic is fixed at $\approx$ 1900 KB/s. In Figure 3.12(a), not only do well-behaved real-time channels and best-effort traffic continue to receive their expected service, channel 0 also receives only its expected service. The laxity behavior is similar to that shown in Figure 3.11(b). No real-time packets miss deadlines, including those of channel 0. However, as can be from Figure 3.12(b), channel 0 overflows its message queue and drops excess messages. None of the other real-time channels or best-effort traffic incur any packet drops.

3.5.3 Need for Cooperative Preemption

The preceding results demonstrate that the features provided in the architecture are sufficient to maintain QoS guarantees. The following results demonstrate that these features are also necessary.

In Figure 3.13(a), protocol processing for best-effort traffic is non-preemptive. Even though best-effort traffic is processed at a lower priority than real-time traffic, once the best-effort handler obtains the CPU, it continues to process messages from the message queue regardless of any waiting real-time handlers. That is, CPU scheduling is QoS-insensitive. As can be seen, this introduces a significant number of deadline misses and packet drops, even at low best-effort loads. The deadline misses and packet drops increase with best-effort load until the system capacity is reached. At this point, all excess best-effort traffic is dropped, while the drops and misses for real-time channels decline. The behavior is largely



(a) Non-preemptive best-effort processing (b) Non-preemptive real-time processing

Figure 3.13: Violation of QoS guarantees with cooperative preemption disabled.

unpredictable, in that different real-time channels are affected differently, and depends on the mix of channels. Further, this behavior is exacerbated by an increase in the amount of buffer space allocated to best-effort traffic; the best-effort handler now runs longer before blocking due to buffer overflow, thereby increasing the window of non-preemptibility.

Figure 3.13(b) shows the effect of processing real-time messages with preemption only at message boundaries. In addition, early handlers are allowed to execute in a work-conserving fashion but at a priority higher than best-effort traffic. Note that all real-time traffic is still being shaped since logical arrival time is enforced. As before, we observe significant deadline misses and packet drops for all real-time channels. In this case, best-effort throughput also declines due to early real-time traffic having higher processing priority. This behavior worsens when the window of non-preemptibility is increased by draining the message queue each time a handler executes.

3.5.4 Discussion

The above results demonstrate the need for cooperative preemption, in addition to traffic enforcement and CPU scheduling, for access to the CPU. While CPU and link scheduling were always enabled, CPU access by real-time channels was also shaped due to traffic enforcement. If traffic was not shaped, one would observe significantly worse real-time and best-effort performance due to non-conformant traffic. We also note that a fully-preemptive kernel is likely to have larger, unpredictable costs for context switches and cache misses. This is because preemption due to unrelated, even lower priority, activities can occur frequently and at arbitrary instants.

Not only does this result in loss of CPU capacity to unnecessary context switches, it also increases the likelihood of disturbing the footprint in the cache [124], unless the cache is suitably partitioned [110]. This is particularly true for preemption caused by external events such as network interrupts. One can account for the cache miss penalty due to preemption via careful schedulability analysis [105], but frequent preemption still degrades available CPU capacity, as also observed in [69].

Moreover, an important implication of arbitrary preemption for the proposed architecture is that a handler may get preempted just before initiating transmission, even though it had finished preparing a packet for transmission, thus idling the link. Cooperative preemption, on the other hand, provides greater control over preemption points, which in turn improves utilization of resources that may be used concurrently. With cooperative preemption, a handler can initiate transmission on the link before yielding to any higher priority activity.

3.6 Summary and Future Work

In this chapter we have proposed and evaluated a new QoS-sensitive communication subsystem architecture for end hosts that supports guaranteed-QoS connections. The architecture provides various services for managing communication resources, such as admission control, traffic enforcement, buffer management, and CPU & link scheduling. Using our implementation of real-time channels, we demonstrated the efficacy with which the architecture maintains per-channel QoS guarantees and delivers reasonable performance to best-effort traffic. While we demonstrated the need for specific features and policies in the architecture for a relatively lightweight stack, such support will be even more important if computationally intensive services such as coding, compression, or checksums are added to the protocol stack. The usefulness of these features also depends significantly on the relationship between CPU and link bandwidths.

Our work assumes that the network adapter (i.e., the underlying network) does not provide any explicit support for QoS guarantees, other than providing a bounded and predictable packet transmission time. This assumption is valid for a large class of networks prevalent today, such as FDDI and switch-based networks. Thus, link scheduling is realized in software, requiring lower layers of the protocol stack to be cognizant of the delay-bandwidth characteristics of the network. A software-based implementation also enables experimentation with a variety of link sharing policies, especially if multiple service classes are supported. For example, alternative approaches such as setting aside a certain minimum CPU and link bandwidth for best-effort traffic can be explored [65]. The architecture can also be extended to networks providing explicit support for QoS guarantees, such as ATM. However, the communication software may need to track adapter buffer usage in order to schedule the transfer of outgoing packets to the adapter.

The architectural framework and methodology we have adopted is applicable to other host platforms as well. This requires that the host communication subsystem be parameterized accurately to capture overheads and processing costs that comprise the abstraction of the underlying communication subsystem. While we have implemented the architecture on an x-kernel platform, the architecture and the prototype implementation do not utilize any features specific to this platform. We argue that the underlying resource management policies can be supported on any operating system platform and communication subsystem. Chapter 5 demonstrates this by developing a guaranteed-QoS communication service based on this architecture, but for a completely different software and hardware platform.

For true end-to-end QoS guarantees, scheduling of channel handlers must be integrated with application scheduling. Chapter 8 examines the issues involved in realizing such an integration. The proposed architecture can be extended in several directions. We have extended the null device into a sophisticated network device emulator, called END, providing link bandwidth management [78]. Using END, we can explore additional issues involved when interfacing to adapters with support for QoS guarantees. In addition to the nature of QoS guarantees, various alternatives for adapter support for buffer management can be explored [101, 130].

While we have focused on per-channel QoS guarantees, this architecture can be easily extended to allow aggregation of multiple connections on the QoS "pipe" provided by a channel. This would then provide effective support for adaptive applications which are built with a certain degree of tolerance for QoS violations. Various elements of this architecture can also be utilized to realize *statistical* real-time channels [63]. Statistical QoS guarantees can potentially be useful to a large class of distributed multimedia applications. Finally, we have extended this architecture to realize a QoS-sensitive communication subsystem for shared-memory multiprocessor multimedia servers [119]. However, numerous issues involved in managing parallelism while providing QoS guarantees need to be explored.

CHAPTER 4

ADMISSION CONTROL EXTENSIONS FOR END HOSTS

4.1 Introduction

The previous chapter focused on architectural components within the communication subsystem to realize QoS guarantees. This chapter focuses on bridging the gap between theory and practice in the management of host CPU and link resources for real-time communication. Using our implementation of real-time channels, we illustrate the tradeoff between useful resource capacity, which is the proportion of the raw resource capacity that can be utilized effectively, and channel admissibility. In the context of real-time channels, useful resource capacity determines the number and type of real-time channels accepted for service and the performance delivered to best-effort traffic.

A channel requires a portion of the available CPU bandwidth to process each generated message and packetize it. Similarly, on a sending host it requires a portion of the available link bandwidth to transmit each packet on the link. Since these two resources typically differ in their performance characteristics, they present different tradeoffs for resource management. Assuming that both the CPU and link can be reallocated only at packet boundaries, allocating the link to transmit a packet from another channel usually has no additional overheads associated with it. However, allocating the CPU to perform protocol processing for another channel incurs significant overheads in the form of context switches and cache misses. Excessive CPU preemption, therefore, reduces available resource capacity, effectively increasing the resource usage attributed to a channel. Limiting preemption, however, increases the temporal window of priority inversion for CPU access, potentially reducing channel admissibility. Correspondingly, if packet size is increased, the total CPU bandwidth required to process a given message is reduced. However, this is accompanied by an increase in packet transmission time, and hence in the temporal window of priority inversion for link access; again, this potentially impacts channel admissibility negatively.

We demonstrate that the above-mentioned tradeoffs are affected significantly by the choice of implementation paradigms and the (temporal) grain at which CPU and link resources are multiplexed amongst active channels. To account for this effect, we extend the admission control procedure for real-time channels originally proposed using idealized resource models. Our results show that, compared to idealized resource models, practical considerations significantly reduce channel admissibility. Further, the optimum choice of the multiplexing grain depends on several factors such as resource preemption overheads, the relationship between CPU and link bandwidth, and the interaction between CPU and link bandwidth allocations.

Similar tradeoffs arise for message handling at a receiving host. In addition, the mechanism adopted for packet input must also be considered, since it affects performance and hence channel admissibility significantly. In interrupt-mode packet input, the adapter interrupts the host on each packet arrival. This may result in excessive overheads and leave the host susceptible to receive livelock [151], a scenario in which the host is continuously receiving and discarding arriving packets (due to queue overflow, say) such that effective system throughput falls to zero. Polled-mode packet input, an alternative mechanism in which the host periodically polls the network adapter for arrived packets, can be effective in preventing receive livelock [125]. Packets can also be input using a hybrid mechanism combining interrupts (under low network input load) with polling (under high network input load). We develop admission control extensions for a receiving host under interrupt-mode and polled-mode packet input, and for hosts that are both a source and destination of real-time channels.

The issues of simultaneous management of CPU and link bandwidth for real-time communication are of wide-ranging interest. Our present work is applicable to other proposals for real-time communication and QoS guarantees [8, 188]. Further, the proposed admission control extensions are general and applicable to other host and network architectures. In particular, Internet servers running TCP/IP protocol stacks supporting Integrated Services [22, 65], especially the guaranteed service class [159], can also benefit from these extensions. Similarly, Internet routers can apply these extensions when incoming packets must be fragmented before forwarding in order to reconcile the different MTUs of the attached networks.

While we only consider management of communication resources, the present work can be extended to incorporate application scheduling as well. Our analysis is directly applicable if a portion of the host processing capacity can be reserved for communication-related activities, e.g., via capacity reserves [107, 123]. Chapter 8 discusses applicability of these admission control extensions to application-level framing and user-level protocol processing architectures.

The rest of the chapter is organized as follows. The next section the issues involved in managing CPU and link bandwidth at a sending host for QoS-sensitive protocol processing and packet transmissions, respectively. The modifications required in the admission control procedure to manage CPU and link bandwidth simultaneously are presented next. The following section studies the tradeoff between useful resource capacity and channel admissibility. Admission control extensions for receiving hosts, including hosts engaged in simultaneous data transmission and reception, are developed subsequently. Finally, we conclude the chapter with a summary of the key contributions and implications.

4.2 Managing CPU and Link Bandwidth

As mentioned earlier, Algorithm `D_order` [92] computes the worst-case response time for a message. This response time has two components: the time spent waiting for resources and the time spent using resources. At the sending host, the time spent using resources is equal to the *message service time*, the time required to process and transmit all the packets constituting the message. Similarly, at the receiving host the message service time includes the time to receive and process all the packets constituting the message. To calculate the time spent waiting for resources, one must consider the preemption model used for resource access. The following discussion focuses on the sending host, but, much of the discussion is also applicable to the receiving host. Specific issues affecting receiving hosts are discussed in Section 4.5.

The real-time channel model presented in [92] accounts for non-preemptive packet transmissions, but assumes an ideal preemption model for CPU access, i.e., the CPU can be allocated to a waiting higher-priority handler immediately at no extra cost. Under this assumption, message service time is determined solely by the CPU processing bandwidth required to packetize the message, and the link bandwidth required to transmit all the packets. The time spent waiting for resources is calculated by accounting for resource usage by messages from all higher-priority channels, and the one-packet delay (due to non-preemptive packet transmission) in obtaining the link. However, as explained below, implementation issues necessitate extensions to the model to account for implementation overheads and constraints.

4.2.1 Implementation Issues

Several implementation issues impact resource management policies. These include handler execution requirements, implementation of link scheduling, and the relationship between CPU and link bandwidth.

Handler Execution

Preemption of an executing process/thread comes with a significant cost due to a context switch and the associated cache miss penalty. Preemption effectively increases the CPU usage attributed to a channel, which in turn reduces the CPU processing bandwidth available for real-time channels; immediate preemption is thus too expensive. It is desirable to limit the number of times a handler is forced to preempt the CPU in the course of processing a message.

At the other extreme, non-preemptive execution of handlers implies that the CPU can be reallocated to a waiting handler only after processing an entire message. This results in a coarser (temporal) grain of channel multiplexing on the CPU and makes admission control less effective. More importantly, admission control must consider the largest possible message size across all real-time and best-effort channels; maximum message size for best-effort traffic may not even be known *a priori*. An intermediate solution is to preempt the CPU only at specific preemption points, if needed. Since message processing involves packetization, the CPU can be preempted after processing every packet. The important parameter here is the number of packets processed between preemption points, which determines the (temporal) grain at which the CPU can be multiplexed between channels. Admission control must account for the extra delay in obtaining the CPU which may be currently allocated to a lower-priority handler.

In the absence of per-byte copying overheads, the total CPU time required to process a message is directly proportional to the number of packets constituting the message. Clearly, assuming that the communication subsystem does not copy message data unnecessarily (true for our implementation), the CPU processing time will be minimum if a single packet constituted the entire message, i.e., if the packet size was the same as the message size. However, as explained below, the total time required to transmit a packet on the link is determined primarily by the size of the packet, although initiation of transmission involves non-zero per-packet overhead.

If the set of channels requesting service have identical traffic specifications, and hence the same maximum message size, then single-packet messages maximize channel admissibility. However, under a heterogeneous mix of real-time channels (with large and small messages), a large packet size would significantly reduce the admissibility for channels with messages smaller than the chosen packet size. Packet size, therefore, also plays a significant role in determining channel admissibility.

Implementation of Link Scheduling

An assumption often made when formulating resource management policies for communication is that CPU and link bandwidth can be independently allocated to a channel. This assumption may get violated in an implementation depending on the paradigm used to implement link scheduling.

We consider three options for implementing link scheduling in software:

O1: Packets are scheduled for transmission either in the context of the currently-executing channel handler (via a function call) or in interrupt context after each packet transmission.

O2: Packets are scheduled for transmission by a dedicated process that executes at the highest priority and is signalled via semaphore operations.

O3: Packets are scheduled for transmission either in the context of the currently-executing channel handler or in the context of a new thread that is fired up after every packet transmission.

O1 and O2 differ significantly in the implications for CPU and link bandwidth allocation, since with O2 the link scheduler must also be scheduled for execution on the CPU. Since O3 presents tradeoffs similar to O2, we focus on O1 and O2 in the discussion below.

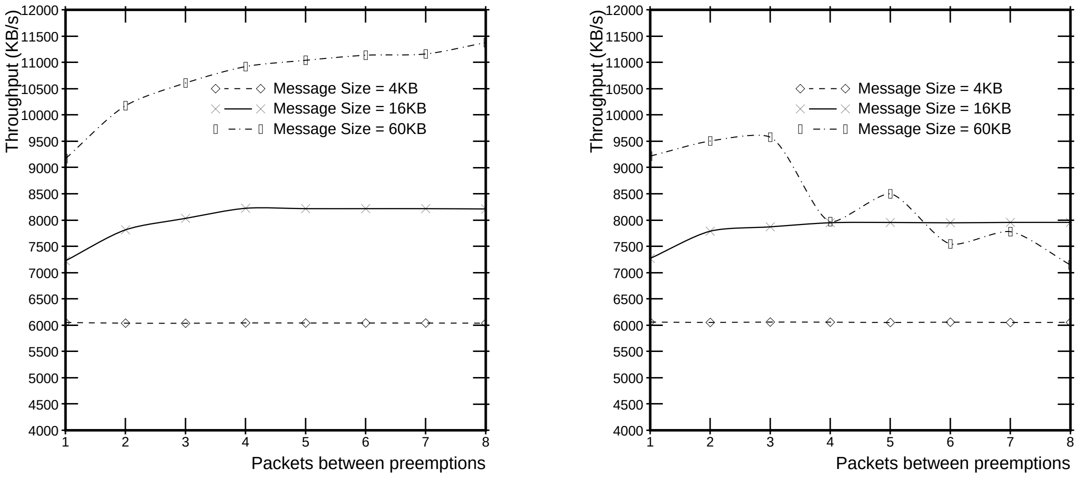
Selecting a packet for transmission incurs some overhead in addition to that of initiating transmission on the link. Additional overhead may be involved if the link scheduler must transfer packets between link packet queues [117]. In O1, the scheduler is frequently invoked from the interrupt service routine (ISR) announcing completion of packet transmission. Since the scheduling overhead involved can be substantial in the worst case, it is undesirable to incur this penalty in the ISR, as this prolongs the duration for which network interrupts are disabled. If the host is also receiving data from the network, there is now a greater likelihood of losing incoming data.

O2, on the other hand, does not suffer from this problem; since scheduler processing is scheduled for execution, it is performed outside the ISR. In addition to keeping the ISR short, this paradigm also has some software structuring benefits such as a relatively cleaner implementation. However, because the link scheduler is itself scheduled for execution on the CPU, there is now an additional overhead of a context switch and the accompanying (instruction) cache miss penalty for each packet transmission. More importantly, allocation of CPU and link bandwidth is closely coupled in O2. This coupling can potentially lower the utilization of the link and, as demonstrated in Section 4.4, significantly reduces channel admissibility and also makes it unpredictable.

Relationship Between CPU and Link Bandwidth

A conservative estimate of message service time can be obtained by adding the total CPU processing time and the total link transmission time. However, this ignores the overlap between CPU processing and link transmission of packets constituting the same message. The extent of this overlap depends largely on the relationship between the CPU and link bandwidth, i.e., on the relative speed of the two. To improve channel admissibility, message service time must be calculated to account for this overlap. The extent of the overlap also depends on the implementation option used for link scheduling. While O1 allows link utilization to be kept relatively high, O2 can cause the link to idle even when there are packets available for transmission. From another perspective, O2 forces link scheduling to be non-work-conserving while O1 allows for work-conserving packet transmissions.

While admission control can utilize the overlap between CPU processing and link trans-



(a) Option O1

(b) Option O2

Figure 4.1: Throughput as a function of packets processed between preemption points (packet size 4 KB).

mission of packets belonging to a message, it cannot do so for the potential overlap between CPU processing and link transmission of packets belonging to *different* messages. Since message arrivals serve as system renewal points, no *a priori* assumptions can be made about the presence of messages in the system. This makes admission control slightly pessimistic, but is necessary for provision of deterministic QoS guarantees.

Determination of $\mathcal{L}_x$: As mentioned earlier, the packet transmission time $\mathcal{L}_x(s)$ for a packet of size s measures the delay between initiation and completion of packet transmission on the network adapter. That is, it determines the minimum time between successive packet transmission invocations by the link scheduler. Note that with non-preemptive packet transmissions on the link, $\mathcal{L}_x(\mathcal{S})$ is also the delay experienced by a waiting highest-priority packet to commence transmission, where $\mathcal{S}$ is the maximum packet size. In order to explore the effects of the relationship between CPU and link bandwidth using the null device, we select $\min(\mathcal{B}_l, \mathcal{B}_x)$ to conform to a desired link (and data transfer) speed, measured in nanoseconds (ns) required to transfer one byte. On the null device, $\mathcal{C}_x$ is determined by the granularity of time-keeping and overhead of communication with the host processor; the measured value of $\mathcal{C}_x$ is $\approx 40\ \mu s$.

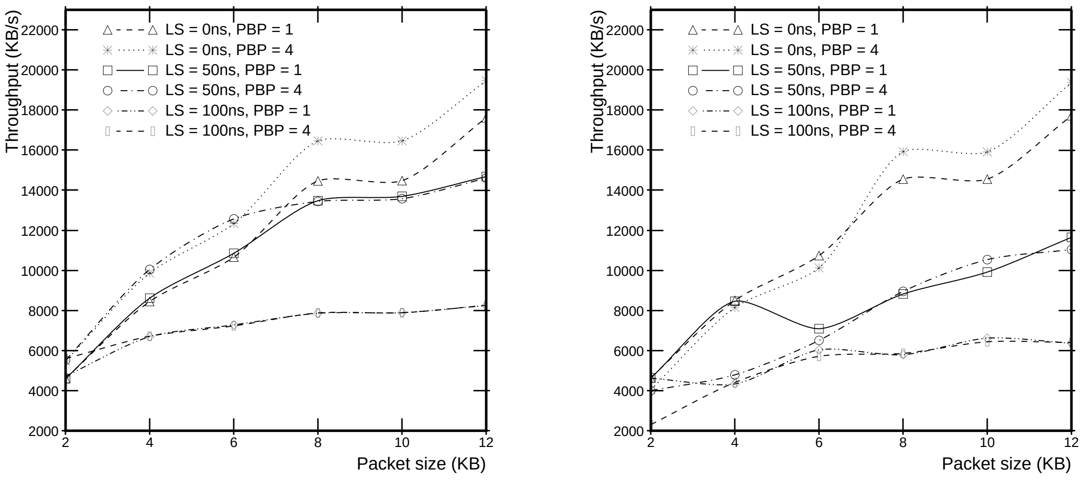
4.2.2 Performance Implications

To illustrate the performance implications of some of the above-mentioned issues, we ran several experiments using our real-time channel implementation to measure system throughput (kilobytes(KB)/second) as a function of parameters such as the number of packets processed between preemption points, the packet size, and link speed. For a given CPU processing power, varying the speed of the link allows us to explore the relationship between CPU and link bandwidth. In the experiments reported here, four best-effort channels were created and messages generated on these channels continuously; each experimental run involved transmission of over 25,000 packets. Multiple runs produced consistently repeatable results. The results reported are representative in that similar trends were obtained with more channels and other parameter settings. An experimental parameterization of our implementation yielded the values listed in Table 4.1.

Figure 4.1 shows system throughput (useful resource capacity) as a function of the number of packets processed between preemption points (PBP) for several message sizes. The packet size is fixed at 4 KB and the link speed (LS) is set at 0 ns per byte, i.e., the link is “fast” relative to the CPU. For O1 (Figure 4.1(a)), changing PBP has no effect when the message size is 4 KB; this is expected for single-packet messages. As message size increases, so do the number of packets and throughput increases until PBP equals the number of packets in the message. After this point PBP has no effect on throughput. As can be seen, for large messages an increase in PBP improves throughput significantly and consistently.

O2 (Figure 4.1(b)) reveals the same behavior as O1 for small- to medium-sized messages. However, for large messages throughput rises initially as throughput rises initially as PBP increases. Subsequently, throughput starts falling sharply, in a non-linear fashion. The decline in throughput is due to increasingly poor utilization of the link bandwidth and a corresponding increase in the time to transmit all the packets belonging to the message. The oscillations in throughput are due to subtle interactions between the CPU preemption window and link transmission, as investigated in Section 4.3.

Figure 4.2 shows the measured system throughput as a function of packet size and link speed. Three values of link speed are considered: 0 ns per byte (fast link), 50 ns per byte (medium-speed link), and 100 ns per byte (slow link). For each link speed, we consider two values of PBP, namely, 1 and 4. Message size is kept fixed at 32 KB and packet size is



(a) Option O1 (b) Option O2

Figure 4.2: Throughput as a function of packet size and link speed.

varied from 2 KB to 12 KB.

Consider system throughput for O1 (Figure 4.2(a)). For a given packet size (i.e., fixed CPU processing time), an increase in link speed results in higher throughput for O1 and O2, with O1 outperforming O2. We again notice that, with a fast link (when CPU is the bottleneck), increasing PBP from 1 to 4 provides a significant gain in throughput. For a given value of PBP and link speed, throughput increases with packet size since the CPU processing time reduces due to a reduction in the number of packets constituting the message. An increase in packet size from 8 KB to 10 KB does not change the number of packets and the throughput remains unchanged. As the link becomes slower, however, there is a saturation in the achieved throughput due to the link tending to become a bottleneck. After a certain packet size, for a given link speed, link transmission time exceeds the protocol processing time; thus any gains from a higher PBP cease to matter and the two curves converge. From Figure 4.2(a), this occurs at a packet size of 8 KB for link speed of 50 ns per byte and at 4 KB for link speed of 100 ns per byte.

Figure 4.2(b) shows the system throughput for O2. The trends are similar to those observed when the link is either very fast (CPU is the bottleneck) or very slow (link is the bottleneck) since CPU and link processing overlap almost completely. For a medium speed link (CPU and link bandwidths are more balanced), however, throughput behavior is more non-linear. Subtle interactions between CPU preemption window and link transmission time cause the link to idle until the next preemption point. This explains the drop in throughput at a packet size of 6 KB and PBP of 1. Subsequently, throughput climbs because link utilization improves and CPU requirements continue to decrease. This effect is analyzed in Section 4.3.

4.3 Worst-Case Service and Wait Times

For a channel requesting admission, `D_order` can compute the worst-case message response time (the *system time requirement* in [92]) by accounting for three components:

- the worst-case waiting time ($\mathcal{T}_w$) due to lower-priority handlers or packets,
- the worst-case service time for the message ($\mathcal{T}_s$), and
- the worst-case waiting time due to message arrivals on all existing higher-priority channels ($\mathcal{T}_t^{hp}$).

We show below how $\mathcal{T}_w$ and $\mathcal{T}_s$ can be estimated for a sending host to account for the implementation-related issues highlighted above; $\mathcal{T}_t^{hp}$ can then be recomputed using $\mathcal{T}_s$. Section 4.5 estimates $\mathcal{T}_w$ and $\mathcal{T}_s$ for a receiving host.

Suppose the CPU is reallocated to a waiting handler, if needed, every $\mathcal{P}$ packets; that is, up to $\mathcal{P}$ packets are processed between successive preemption points. Further, (see Table 4.1) let the maximum packet size be $\mathcal{S}$, context switch overhead between handlers be $\mathcal{C}_{sw}$ (this includes scheduling overhead to select a handler for execution), cache miss penalty due to a context switch be $\mathcal{C}_{cm}$, and packet transmission time be $\mathcal{L}_x(\mathcal{S})$ for packet size $\mathcal{S}$. Per-packet protocol processing cost is $\mathcal{C}_p$ and per-packet (link) scheduling overhead of selecting a packet and initiating transmission is $\mathcal{C}_l$. $\mathcal{C}_p$ includes the cost of creating a new fragment, traversing the layers of the protocol stack performing header encapsulation, and enqueueing the packet for transmission. In general, it may also include the cost of copying data, computing checksums, and performing encryption. $\mathcal{C}_l$ includes the cost of a timestamp, accessing the link packet queues, invoking device driver transmit routines, and fielding the transmission-completion interrupt.

Symbol	Description	Value
$\mathcal{C}_{sw}$	time to switch contexts between channel handlers	55 μs
$\mathcal{C}_{cm}$	cache miss penalty due to a context switch	90 μs
$\mathcal{C}_p^{1st}$	first-packet CPU processing cost	420 μs
$\mathcal{C}_p$	per-packet CPU processing cost	170 μs
$\mathcal{C}_l$	per-packet link scheduling cost	160 μs
$\mathcal{P}$	number of packets processed between preemption points	4
$\mathcal{S}$	maximum packet size	4 KB
$\mathcal{L}_x(\mathcal{S})$	link transmission time for packet of size $\mathcal{S}$	245 μs

Table 4.1: Important system parameters.

4.3.1 Estimating Service Time

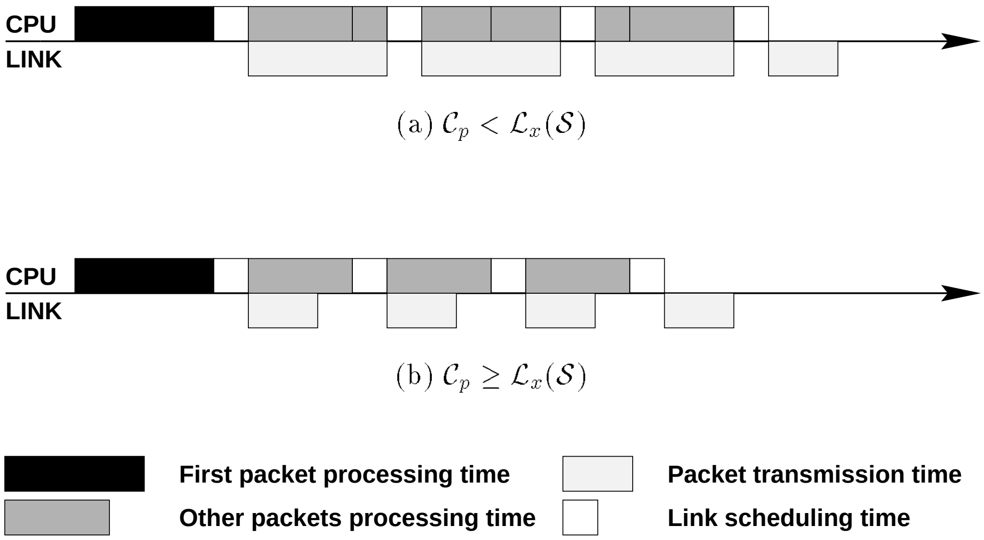
Consider a message of size $\mathcal{M}$ bytes constituting $\mathcal{N}_p$ packets, i.e., $\mathcal{N}_p = \lceil \frac{\mathcal{M}}{\mathcal{S}} \rceil$ packets, with $(\mathcal{N}_p - 1)$ packets of size $\mathcal{S}$ and the last packet of size $\mathcal{S}^{last} = (\mathcal{M} \bmod \mathcal{S})$ if $(\mathcal{M} \bmod \mathcal{S}) \neq 0$, else $\mathcal{S}^{last} = \mathcal{S}$. Thus, link transmission time is $\mathcal{L}_x(\mathcal{S})$ for all but the last packet and $\mathcal{L}_x(\mathcal{S}^{last})$ for the last packet. Protocol processing cost for the first packet is $\mathcal{C}_p^{1st}$ while subsequent fragments each incur a lower cost $\mathcal{C}_p$. $\mathcal{C}_p^{1st}$ includes the fixed cost of obtaining the message for processing, the cost of a timestamp, and the cost of preparing the first packet.[1] Both $\mathcal{C}_p^{1st}$ and $\mathcal{C}_p$ include the cost of network-level encapsulation. We estimate $\mathcal{T}_s$ for O1 and O2 separately.

Option O1: Given the system parameters listed in Table 4.1, the worst-case service time for O1 is given by

$$\mathcal{T}_s^{O1} = \begin{cases} \mathcal{C}_p^{1st} + \mathcal{L}_x^m + \mathcal{C}_l^m + \mathcal{C}_{pr} & \text{if } \mathcal{C}_p < \mathcal{L}_x(\mathcal{S}) \\ \mathcal{C}_p^m + \mathcal{C}_l^m + \mathcal{L}_x(\mathcal{S}^{last}) + \mathcal{C}_{pr} & \text{otherwise} \end{cases}$$

where $\mathcal{L}_x^m = (\mathcal{N}_p - 1)\mathcal{L}_x(\mathcal{S}) + \mathcal{L}_x(\mathcal{S}^{last})$ is the total link transmission time for the message, $\mathcal{C}_l^m = \mathcal{N}_p\mathcal{C}_l$ is the total link scheduling overhead for the message, $\mathcal{C}_{pr} = \lfloor \frac{\mathcal{N}_p - 1}{\mathcal{P}} \rfloor \mathcal{C}_{csp}$ is the total cost of preemption during the processing of the message ($\mathcal{C}_{csp} = \mathcal{C}_{cm} + \mathcal{C}_{sw}$), and $\mathcal{C}_p^m = \mathcal{C}_p^{1st} + (\mathcal{N}_p - 1)\mathcal{C}_p$ is the total protocol processing cost for the message. Protocol processing and link transmission overlap in O1 is illustrated in Figure 4.3.

[1] Our fragmentation protocol traverses a slower path for messages larger than $\mathcal{S}$ bytes; the first packet thus has a higher processing cost.



(a) $\mathcal{C}_p < \mathcal{L}_x(\mathcal{S})$

(b) $\mathcal{C}_p \geq \mathcal{L}_x(\mathcal{S})$

Figure 4.3: Protocol processing and link transmission overlap in O1.

If $\mathcal{C}_p < \mathcal{L}_x(\mathcal{S})$, there is at least the cost of processing the first packet. Since the link transmission time dominates the time to process subsequent packets (see Figure 4.3(a)), message service time is determined by link transmission time for the message and the total link scheduling and preemption overheads incurred. If $\mathcal{C}_p \geq \mathcal{L}_x(\mathcal{S})$ (Figure 4.3(b)), however, message service time corresponds to the total protocol processing time for the message plus the time to transmit the last packet, in addition to the total link scheduling and preemption overheads.

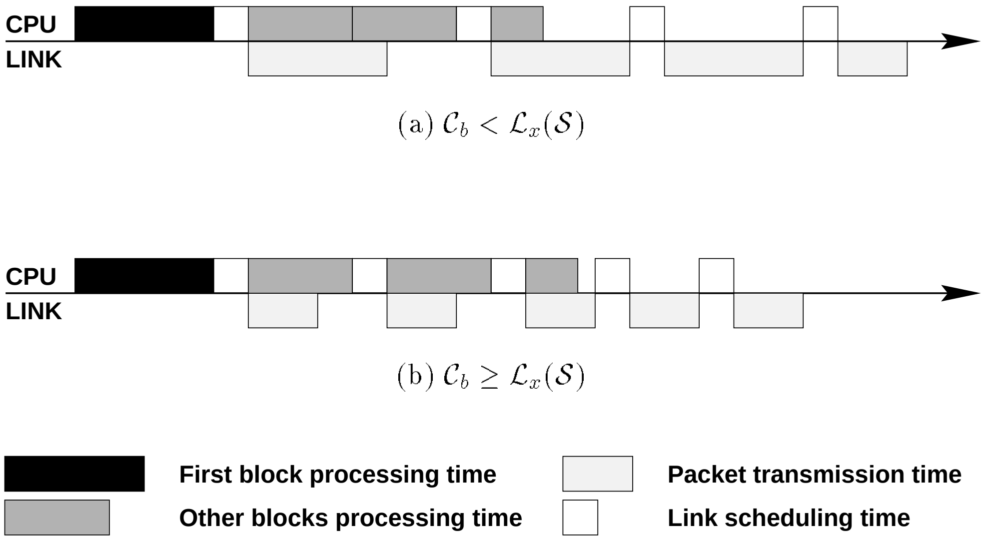
Option O2: Calculation of the worst-case service time for O2 is done similarly; however, we must now consider the processing of *blocks* of packets with each block comprising no more than $\mathcal{P}$ packets. The number of blocks in a message with $\mathcal{N}_p$ packets is given by $\mathcal{N}_b = \lfloor \frac{\mathcal{N}_p - 1}{\mathcal{P}} \rfloor + 1$. The protocol processing cost for the first block is given by $\mathcal{C}_b^{1st} = \mathcal{C}_p^{1st} + (\min(\mathcal{N}_p, \mathcal{P}) - 1)\mathcal{C}_p$, while the cost of processing the last block of packets is given by

$$\mathcal{C}_b^{last} = \begin{cases} \mathcal{C}_b & \text{if } (\mathcal{N}_p \bmod \mathcal{P}) = 0 \\ (\mathcal{N}_p \bmod \mathcal{P})\mathcal{C}_p + \mathcal{C}_{csp} & \text{otherwise} \end{cases}$$

where $\mathcal{C}_b = \mathcal{P}\mathcal{C}_p + \mathcal{C}_{csp}$ is the cost of processing the other blocks, if any. The worst-case service time is given by

$$\mathcal{T}_s^{O2} = \begin{cases} \mathcal{T}^A + \mathcal{T}_w^{O2,cpu} & \text{if } \mathcal{C}_b < \mathcal{L}_x(\mathcal{S}) \\ \mathcal{T}^B & \text{otherwise} \end{cases}$$

where $\mathcal{T}^A = \mathcal{C}_b^{1st} + \mathcal{L}_x^m + \mathcal{C}_l^m$ and



(a) $\mathcal{C}_b < \mathcal{L}_x(\mathcal{S})$

(b) $\mathcal{C}_b \geq \mathcal{L}_x(\mathcal{S})$

Figure 4.4: Protocol processing and link transmission overlap in O2.

$$\mathcal{T}_w^{O2,cpu} = \mathcal{C}_p^{1st} + (\mathcal{P} - 1)\mathcal{C}_p + \mathcal{C}_{csp} + \mathcal{C}_l^{'},$$

with $\mathcal{C}_l^{'} = \mathcal{C}_l + \mathcal{C}_{csp}$. $\mathcal{T}^B$ is given by

$$\mathcal{T}^B = \begin{cases} \mathcal{T}^A & \text{if } \mathcal{N}_b = 1 \\ \mathcal{T}_a^C + \mathcal{T}_b^C & \text{otherwise} \end{cases}$$

where $\mathcal{T}_a^C = \mathcal{C}_b^{1st} + (\mathcal{N}_b - 2)\mathcal{C}_b + \max(\mathcal{C}_b^{last}, \mathcal{L}_x(\mathcal{S}))$ and $\mathcal{T}_b^C = (\mathcal{N}_p - \mathcal{N}_b)\mathcal{L}_x(\mathcal{S}) + \mathcal{L}_x(\mathcal{S}^{last}) + \mathcal{C}_l^m$. Protocol processing and link transmission overlap in O2 is illustrated in Figure 4.4.

If $\mathcal{C}_b < \mathcal{L}_x(\mathcal{S})$ (Figure 4.4(a)), each block will complete processing its packets before a packet gets transmitted. This is because in O2, the link scheduler does not get to run until the CPU is preempted. So the message service time simply corresponds to the time to process the first block of packets and transmit each packet. However, the link scheduler may now have to wait for time $\mathcal{T}_w^{O2,cpu}$ to obtain access to the CPU and initiate packet transmission; in the worst-case this wait penalty must be assumed for each packet that is to be transmitted. If $\mathcal{C}_b \geq \mathcal{L}_x(\mathcal{S})$, however, the link transmission time is faster than the time to process the packets in a block (Figure 4.4(b)). Except for the first block, each block will overlap precisely one packet transmission. This implies that message service time is determined by the processing time for all the blocks, the transmission time of $(\mathcal{N}_p - \mathcal{N}_b)$ packets, and the total link scheduling overhead. There is no wait time in this case because the link scheduler is guaranteed to run next, even if the handler must preempt the CPU to another handler. If the last block of packets is shorter than $\mathcal{L}_x(\mathcal{S})$, only the time to transmit the last packet contributes to the message service time; else the time to process the last block must be considered.

Note that the analysis of O1 conservatively accounts for the link scheduling overhead for each packet to be transmitted. As a result, the link is not being utilized fully even when packets are ready for transmission, as is evident from Figure 4.3. This tends to reduce channel admissibility in O1, especially when $\mathcal{C}_p > \mathcal{L}_x(\mathcal{S})$, relative to O2, where we already consider processing of blocks of packets. If $\mathcal{N}_p \geq \mathcal{P}$, we only need to account for the link scheduling overhead once every $\mathcal{P}$ packets, while still accounting for interrupt overhead per packet. Between two successive yield points, the packet transmission order corresponds to the packet generation order. This is because the packet transmission time is completely overlapped with packet processing time, and a handler will not preempt the CPU before processing $\mathcal{P}$ packets. The link scheduling overhead can be reduced further by pre-selecting the packet to transmit next (thus overlapping with ongoing packet transmission) and recovering from any potential priority inversions. Moving the link scheduling function to the adapter improves performance significantly [80] by facilitating greater overlap between packet processing and packet selection, and reducing context switches and cache perturbation, thus increasing channel admissibility for both O1 and O2.

4.3.2 Estimating Wait Time

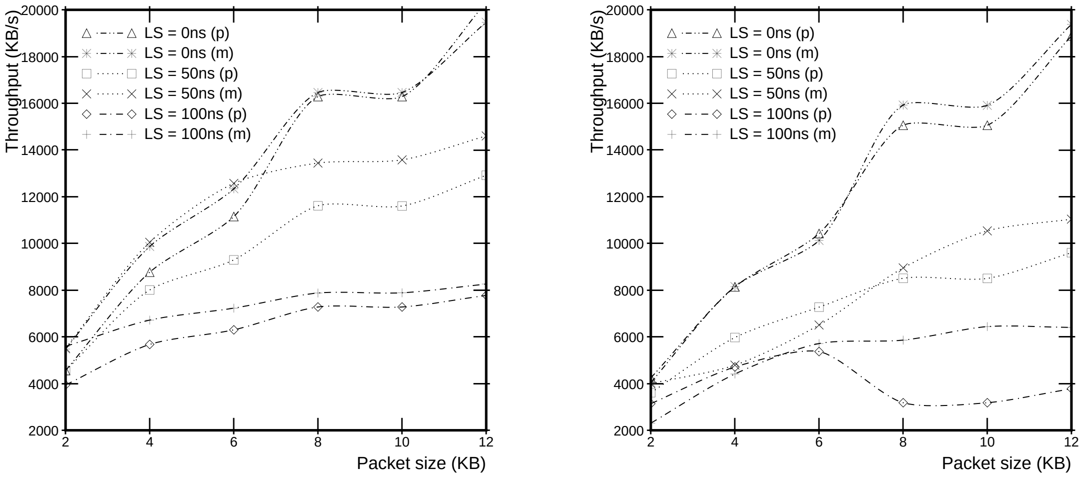
To compute the total message wait time, we first consider the time spent waiting for a lower-priority handler to relinquish the CPU, followed by the time spent waiting for the link.

Option O1: The worst-case CPU time for a block of packets is $\mathcal{C}_b^{max} = \mathcal{C}_b^{1st} + (\mathcal{P} - 1)\mathcal{C}_p$, during which up to $\lceil \frac{\mathcal{C}_b^{max}}{\mathcal{L}_x(\mathcal{S})} \rceil$ packets could complete transmission. Thus, the worst-case CPU wait time is

$$\mathcal{T}_w^{O1,cpu} = \mathcal{C}_b^{max} + \lceil \frac{\mathcal{C}_b^{max}}{\mathcal{L}_x(\mathcal{S})} \rceil \mathcal{C}_l + \mathcal{C}_{csp}.$$

Now, consider the worst-case link wait time. A lower-priority transmission could be started in an ISR just before the currently executing handler makes a packet ready for transmission. Due to non-preemptive packet transmission, the packet generated by the handler will have to wait for the lower-priority packet to complete transmission. Therefore, the worst-case link wait time is simply $\mathcal{T}_w^{O1,link} = \mathcal{L}_x(\mathcal{S})$, and $\mathcal{T}_w^{O1} = \mathcal{T}_w^{O1,cpu} + \mathcal{T}_w^{O1,link}$.

Option O2: The worst-case CPU wait time equals the time to process up to $\mathcal{P}$ packets



(a) Option O1 (b) Option O2

Figure 4.5: Comparison of measured (m) and predicted (p) throughput ($\mathcal{P} = 4$).

on a lower-priority channel followed by a context switch to the link scheduler, followed by another context switch to the waiting handler. Thus,

$$\mathcal{T}_w^{O2,cpu} = \mathcal{C}_b^{max} + \mathcal{C}_{csp} + \mathcal{C}_l',$$

where $\mathcal{C}_l' = \mathcal{C}_l + \mathcal{C}_{csp}$. Similarly, $\mathcal{T}_w^{O2,link}$ is derived as follows. If $\mathcal{L}_x(\mathcal{S}) \leq \mathcal{C}_b^{max}$, $\mathcal{T}_w^{O2,link} = 0$; else $\mathcal{T}_w^{O2,link} = \mathcal{T}_w^{O2,cpu}$. If the link scheduler completed transmission before the first block is processed, the next packet can be transmitted as soon as the handler yields the CPU. In the worst case, a lower-priority handler begins processing a non-preemptive block of packets, making the link scheduler wait for this block to end before it gets a chance to run and transmit the next higher-priority packet. Thus, $\mathcal{T}_w^{O2} = \mathcal{T}_w^{O2,cpu} + \mathcal{T}_w^{O2,link}$.

4.3.3 Experimental Validation

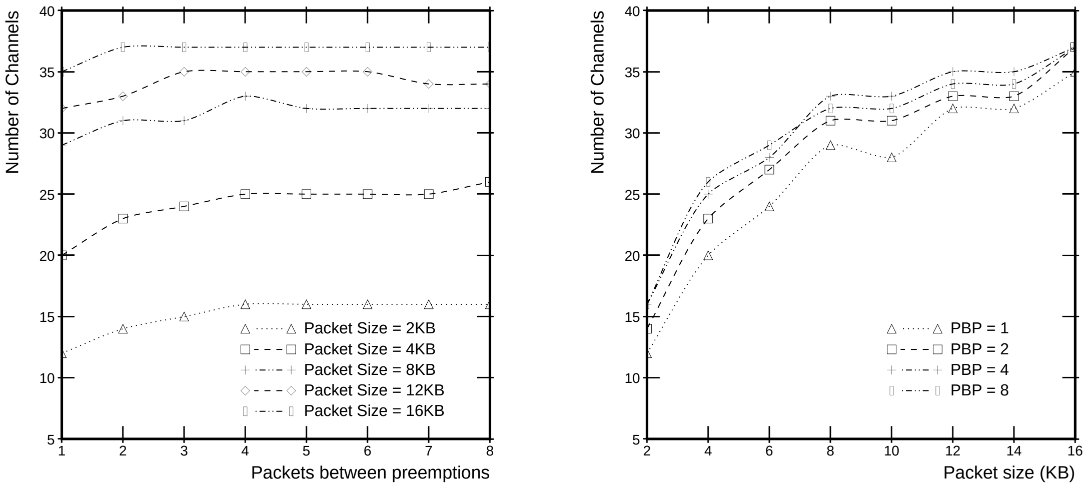
Our implementation provides admission control based on the above estimates of service and wait times. While these estimates are geared towards real-time guarantees, and therefore are necessarily conservative, it is insightful to compare the throughput predicted by these estimates and the best-effort throughput measured using the real-time channel implementation. For this purpose, we parameterized the communication subsystem, including the protocol stack, extensively to determine the system parameter values listed in Table 4.1. We validated the implementation as a function of packet size, for different values of link speed; the main results are summarized below.

Figure 4.5 compares the predicted and measured throughputs as a function of packet size, for three values of link speed (LS). Predicted throughput tracks measured throughput well for O1 (Figure 4.5(a)) as well as O2 (Figure 4.5(b)). However, for O1 with medium link speeds, the predicted and measured throughputs diverge significantly; we attribute this to overly conservative estimates of $\mathcal{C}_{sw}$ and $\mathcal{C}_{cm}$. The estimates are necessarily conservative in accounting for worst-case times which, though necessary for real-time traffic, may be relatively small on average.

These validation experiments reveal certain shortcomings in determining the system parameter values listed in Table 4.1; part of the discrepancy stems from the unpredictability introduced by caches. More refined experiments are necessary to select accurate values for $\mathcal{C}_p$, $\mathcal{C}_{sw}$ and $\mathcal{C}_{cm}$. It has been shown that the cache behavior of network protocols is protocol-specific and that cache misses play a significant role in protocol stack execution latency [16, 133]. For partitioned caches, cache behavior can be made more predictable under control of the operating system [110]. We examine some of the issues involved and the difficulty of accurate parameterization in Chapter 6.

4.4 Channel Admissibility

In this section, we demonstrate that the tradeoff between resource capacity and channel admissibility is influenced significantly by $\mathcal{P}$, the number of packets between preemptions, and $\mathcal{S}$, the packet size. As expected, the mechanism employed to implement link scheduling and the relationship between CPU and link bandwidth also have a profound effect on channel admissibility. We studied channel admissibility for O1 and O2 for a range of link speeds, message sizes, rates and deadlines. In the following, we present and compare the results for a link speed of 50 ns per byte, message size of 32 KB, and message inter-arrival of 100 ms. We admit as many channels as possible with deadline of 100 ms.



(a) Effect of $\mathcal{P}$ (PBP)

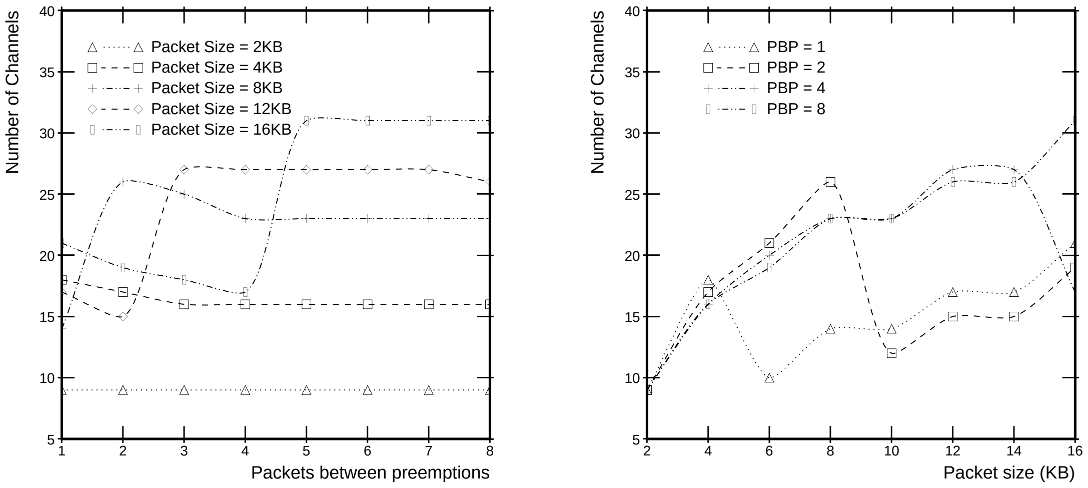
(b) Effect of $\mathcal{S}$

Figure 4.6: Effect of $\mathcal{P}$ and $\mathcal{S}$ on channel admissibility in O1.

4.4.1 Channel Admissibility in O1

From Figure 4.6, channel admissibility in O1 rises with both $\mathcal{P}$ and $\mathcal{S}$ due to the accompanying reduction in protocol processing cost and work-conserving packet transmissions. As $\mathcal{P}$ rises (Figure 4.6(a)), protocol processing costs decline, resulting in a small increase in channel admissibility. As $\mathcal{P}$ continues to rise, the marginal benefits in protocol processing costs decline. Due to an increase in the window of non-preemptibility, channel admissibility either saturates or shows a small decline. Figure 4.6(b) shows that increasing $\mathcal{S}$ increases channel admissibility substantially, since the reduction in the required CPU bandwidth more than compensates for the increase in the non-preemptibility window.

The above results might suggest that arbitrary-sized packets (i.e., sending each message as a single packet) are desirable to maximize channel admissibility. While this is true if all channels carry same-sized messages, the same cannot be said for channels with smaller (single-packet) messages. Increasing $\mathcal{P}$ and $\mathcal{S}$ arbitrarily only serves to increase the window of non-preemptibility with no reduction in CPU requirements for small messages. Large values of $\mathcal{P}$ and $\mathcal{S}$ lower admissibility for channels with small messages, especially those with tight deadlines. Selection of $\mathcal{P}$ and $\mathcal{S}$ therefore depends on system parameters as well as the targetted mix of communication traffic.



(a) Effect of $\mathcal{P}$ (PBP)

(b) Effect of $\mathcal{S}$

Figure 4.7: Effect of $\mathcal{P}$ and $\mathcal{S}$ on channel admissibility in O2.

4.4.2 Channel Admissibility in O2

In contrast to O1, channel admissibility when using O2 to schedule packets is significantly lower and the behavior is highly non-linear. This is explained easily using our preemption model. Consider the effect of $\mathcal{P}$ on channel admissibility (Figure 4.7(a)) for 8 KB packets. With the given link speed and $\mathcal{P} = 1$, link transmission time is greater than protocol processing time for a block of packets. The model in Figure 4.4(a) applies, making the channel susceptible to long idle periods (Section 4.3). For $\mathcal{P} = 2$, the transmission time for a packet remains unchanged, but the processing time for a preemption block increases, making it more than the link transmission time. This results in the scenario in Figure 4.4(b)), in which the worst-case transmission time is reduced substantially, thereby increasing channel admissibility.

As $\mathcal{P}$ increases further, the nature of overlap between CPU processing and link transmission remains unchanged. Channel admissibility either remains unchanged or declines slightly due to an increase in the window of non-preemptibility. This transition occurs for all but the smallest packet sizes. In general, the larger the packet, the greater the $\mathcal{P}$ that causes a change in the nature of overlap, namely, from packet transmission time being slower to it being faster than the processing time for the longest block of packets.

Figure 4.7(b) presents the same information as a function of $\mathcal{S}$. As packet size increases, there is an initial increase in admissibility due to reduced protocol processing load. At a certain value of $\mathcal{S}$, link transmission time becomes larger than block processing time, changing the scenario from that in Figure 4.4(b) to that in Figure 4.4(a). Further increase in packet size slowly increases channel admissibility due to reduced CPU bandwidth requirements. As seen from Figure 4.7, the best operating point for O2 depends critically on system parameters. Since a change in channel characteristics will significantly change channel admissibility, a system parameterized and optimized for a particular workload is unlikely to perform well under a heterogeneous workload.

Using a model of ideal resources, i.e., with no CPU preemption cost and an immediately preemptible CPU, we found $\approx 40\%$ improvement in channel admissibility over and above O1 with $\mathcal{P} = 1$. Thus, it is necessary to account for non-ideal characteristics (context switch overhead, cache miss penalty) of real systems.

4.5 Admission Control Extensions for Receiving Hosts

The previous sections have considered admission control extensions for sending hosts. In this section we illustrate how similar admission control extensions for communication resource management can be developed for *receiving* hosts as well. While the approach and methodology adopted are the same, the analysis must take into account key differences between sending and receiving hosts, as outlined in Section 4.5.1. We first consider a receiving host that serves as the destination for multiple channels, i.e., only handles incoming data traffic (Section 4.5.2). Then, in Section 4.5.3 we consider *simultaneous* data transmission and reception on different channels at a host. Since the tradeoffs are similar to those considered previously for a sending host, we only present the analysis and admission control extensions. A subset of these extensions, as well as the ones presented for sending hosts, have been implemented in the guaranteed-QoS communication service described in Chapter 5.

4.5.1 Reception Issues and Assumptions

Communication resource management at a receiving host differs from that at a sending host in several ways. First, with heterogeneous hosts, the preemption overheads and CPU speed of the receiver, and hence the granularity at which the receiver CPU multiplexes between handlers, could be substantially different from that of the sender. Similarly, the CPU capacity available (reserved) for communication processing at the receiver may also differ from that at the sender. Further, the raw performance of the receive path may be different from that of the transmission path, necessitating accurate parameterization of the costs incurred during message reception. In addition to protocol processing overheads such as message reassembly, a receiving host incurs the overhead of classifying arriving packets, i.e., determining which channels the packets belong to. Such support may be available directly from the adapter (e.g., the VCI in ATM network adapters), or the necessary information may be carried in link-level packet headers [172], in which case the host does not incur any significant packet classification overhead. In the absence of such support, the receiving host must rely upon efficient packet filters [59, 186] to perform the packet classification. Note that this overhead is not incurred at the sending host if the API directly associates outgoing messages with channels. If this is not the case, some classification may be required, but only on a per-message basis, since distinct channel handlers shepherd individual packets down the protocol stack.

Second, the aggregate packet arrival rate at a receiving host may be significantly higher than the aggregate message rate associated with the established channels. This is because it could potentially receive best-effort traffic from other hosts as well, depending on the configuration of the network. Note that the packet transmission rate of a sending host is determined by the aggregate message generation rate of the channels (real-time or best-effort) established at the host. The worst-case aggregate packet arrival rate at a receiving host depends in part on the configuration of the network, i.e., multi-access or point-to-point. In the worst case, a (possibly best-effort) packet could arrive every $\mathcal{L}_r(\mathcal{S}_{min})$ time units, where $\mathcal{S}_{min}$ is the *minimum* packet size (determined from the network technology) and $\mathcal{L}_r(\mathcal{S}_{min})$ is defined as before, i.e.,

$$\mathcal{L}_r(\mathcal{S}_{min}) = \mathcal{C}_r + \frac{\mathcal{S}_{min}}{\min(\mathcal{B}_l, \mathcal{B}_r)}.$$

Again, $\mathcal{B}_r$ is the data transfer bandwidth available to host memory and $\mathcal{B}_l$ is the bandwidth of the attached network link. $\mathcal{C}_r$ includes the cost of setting up DMA transfer operations, if any. We assume that the adapter signals packet arrival (via an interrupt, say) after completing any DMA operations. Also implicit is the assumption that the attached network link can be utilized maximally for back-to-back packet transmissions.

For a point-to-point network (our main focus), however, the aggregate packet arrival rate at the receiver is limited by the aggregate packet transmission rate at the sender attached to the same network link, assuming the receiving host is single-homed. This sender would likely be a router forwarding traffic to the receiver from multiple sending hosts. We assume that the router performs admission control and QoS-sensitive packet forwarding as outlined earlier, such that real-time traffic is serviced at a priority higher than best-effort traffic. It follows that no best-effort traffic will arrive at the receiver during reception of a real-time message. Further, at most one lower-priority real-time packet will arrive ahead of a higher-priority real-time message. We compute the message service and wait times as per the approach adopted in Section 4.3. Unlike the sending host, however, at the receiver we must account for any packet arrivals on other channels (real-time or best-effort) *during* the processing of a real-time message but *after* all the packets of this message have arrived.

Third, a persistent burst of packet arrivals at the receiver can result in *receive livelock* [125, 151], as explained in Chapter 2. Receive livelock can occur even in a point-to-point network with QoS-sensitive forwarding, especially when there is low real-time traffic (which is subjected to admission control) but high, persistent best-effort traffic. One way to prevent or eliminate receive livelock is to limit the rate at which the adapter interrupts the host, and/or to schedule protocol processing of arrived packets at a priority that does not starve other activities on the host.

As described in Chapter 3, in our QoS-sensitive architecture protocol processing for received packets is explicitly scheduled via channel handlers once the device driver receives the packet from the adapter and classifies it. Our architecture, therefore, facilitates provision of QoS guarantees while preventing receive livelock. We note that an alternative approach is outlined in [53], in which receive livelock is prevented by deferring protocol processing of a received packet until the receiving application is scheduled for execution. Besides coupling protocol processing priority with application priority, this approach may not be able to exploit the overlap between packet reception from the network and protocol processing on the host, as discussed in Chapter 8.

We consider provision of QoS guarantees under two different mechanisms by which arriving packets are handled by the host processor, namely, *interrupt mode* and *polled mode*. Interrupt mode handling represents the traditional and most prevalent packet handling mechanism in operating systems and network adapters. For a stable system, the overhead to field an interrupt (denoted as $\mathcal{C}_i$) and demultiplex (i.e., classify) a packet to the corresponding channel packet queue (denoted as $\mathcal{C}_o$) must be sufficiently low compared to the minimum packet interarrival time $\mathcal{L}_r(\mathcal{S}_{min})$; else, the host will be completely overloaded under persistent packet arrivals regardless of how protocol processing is scheduled. Polled mode handling is an alternative to interrupt mode handling that can be effective in preventing receive livelock [125].

Fourth, in order to keep the analysis tractable, we must make additional assumptions about the support available in the adapter for QoS guarantees, as outlined in the discussion below. The nature of adapter support depends on whether the receiving host employs interrupt mode or polled mode handling for packet input, with interrupt mode requiring comparatively less adapter intelligence. These assumptions can be validated easily using the END [78], a sophisticated network device emulator that was evolved from the null device described in Chapter 3. We have used END as a tool to study design improvements in existing adapters [79], and explore adapter-based strategies for receive livelock elimination [81], the details of which are beyond the scope of this dissertation. As for the sending host, we assume that the entire capacity of the host is available for communication processing; our results can be easily adapted for the case where only a certain fraction of the host capacity is reserved for communication processing.

4.5.2 Pure Reception

Under the assumptions made above, we develop admission control extensions for a receiving host that only handles incoming data traffic. That is, it does not generate any real-time or best-effort traffic. Some of the extensions developed continue to apply even in the presence of outgoing traffic, as discussed in Section 4.5.3. We first analyze interrupt mode packet input, followed by an analysis of polled mode packet input.

Interrupt Mode

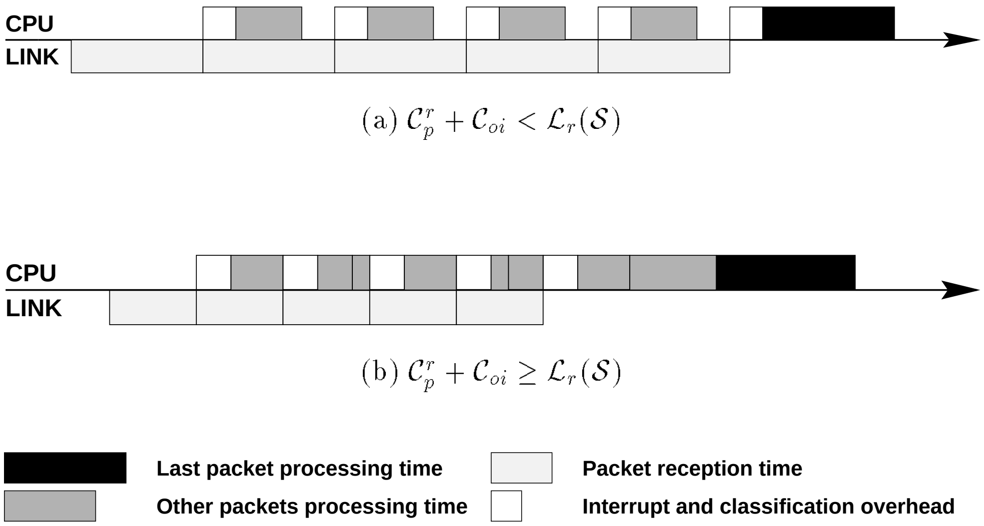
In this mode the adapter signals arrival of a packet by interrupting the host processor. As part of handling the interrupt, the ISR examines packet headers (possibly via a packet filter) to classify the packet, i.e., determine the channel it corresponds to, and enqueues the packet in the channel's input queue for subsequent processing by the corresponding channel handler [117]. Note that the handler performs message reassembly after processing the last packet of a message, and enqueues the message in the channel message queue for subsequent retrieval by the application. Let $\mathcal{C}_p^r$ denote the cost of processing a packet other than the last packet of a received message, and $\mathcal{C}_p^{last,r}$ denote the time to process the last packet, reassemble and deliver the message to the application. Then, $\mathcal{C}_p^{m,r} = (\mathcal{N}_p - 1)\mathcal{C}_p^r + \mathcal{C}_p^{last,r}$ is the total protocol processing time for a message of size $\mathcal{M}$ comprising $\mathcal{N}_p$ packets. Assuming cooperative preemption as before, the message service time at the receiving host is

$$\mathcal{T}_s^{intr} = \begin{cases} \mathcal{C}_p^{last,r} + (\mathcal{N}_p - 1)\mathcal{L}_r(\mathcal{S}) + \max(\mathcal{L}_r(\mathcal{S}^{last}), \mathcal{C}_p^r + \mathcal{C}_{oi}) & \\ \qquad\qquad + \mathcal{C}_{oi} + \mathcal{C}_i' + \mathcal{C}_{pr}^r & \text{if } \mathcal{C}_p^r + \mathcal{C}_{oi} < \mathcal{L}_r(\mathcal{S}) \\ \mathcal{C}_p^{m,r} + \mathcal{L}_r(\mathcal{S}) + \mathcal{N}_p\mathcal{C}_{oi} + \mathcal{C}_i' + \mathcal{C}_{pr}^r & \text{otherwise} \end{cases}$$

where $\mathcal{C}_{pr}^r = \lfloor \frac{\mathcal{N}_p - 1}{\mathcal{P}^r} \rfloor \mathcal{C}_{csp}^r$ is the total cost of preemption during the processing of the message ($\mathcal{C}_{csp}^r = \mathcal{C}_{cm}^r + \mathcal{C}_{sw}^r$). If $\mathcal{C}_i$ is the overhead of fielding a packet arrival interrupt (including the associated preemption overhead), then $\mathcal{N}_p\mathcal{C}_{oi}$ is the total message handling overhead for fielding and servicing these interrupts, where $\mathcal{C}_{oi} = \mathcal{C}_o + \mathcal{C}_i$. $\mathcal{C}_o$ includes the cost of classifying the packet (e.g., via a packet filter), and enqueueing it in the channel input packet queue. $\mathcal{C}_i' = \lceil \frac{\max(0, \mathcal{C}_p^r + \mathcal{C}_{oi} - \mathcal{L}_r(\mathcal{S}^{last})) + \mathcal{C}_p^{last,r}}{\mathcal{L}_r(\mathcal{S}_{min})} \rceil \mathcal{C}_{oi}$ is the overhead of packet arrival interrupts at the maximal possible rate due to packets arriving *after* the arrival of all packets belonging to the real-time message, but during the remaining CPU processing by the handler. The worst-case scenario corresponds to minimum-sized best-effort packets arriving back-to-back. Figure 4.8 illustrates link reception and protocol processing overlap in this case.

If $\mathcal{C}_p^r + \mathcal{C}_{oi} < \mathcal{L}_r(\mathcal{S})$ (Figure 4.8(a)), packet input overhead and processing time is completely overlapped by packet reception time for all but the last two packets of the message. While all packets arrive before processing of the last packet begins, processing of the preceding packet only partially overlaps with the reception time of the last packet in the message, which could be smaller than the other packets. If $\mathcal{C}_p^r + \mathcal{C}_{oi} \geq \mathcal{L}_r(\mathcal{S})$ (Figure 4.8(b)), on the other hand, packet input overhead and processing time completely overlaps packet reception time for all but the first packet of the message.

$\mathcal{C}_i'$ can be reduced further if we assume that the adapter distinguishes between best-effort and real-time traffic, and allows the host to selectively mask interrupts due to best-effort traffic during the processing of a real-time message. With this assumption, a tighter bound on packet arrival rate can be computed using the aggregate real-time traffic generation at the sender (i.e., the immediately upstream host/router). Assuming that the sender performs strict traffic shaping, the aggregate real-time *packet* arrival rate at the receiver is $\sum_{a \in \mathcal{A}} \mathcal{R}_{max}^a \lceil \frac{\mathcal{M}_{max}^c}{\mathcal{S}} \rceil$ packets/second, where $\mathcal{A}$ is the set of real-time channels already es-



(a) $\mathcal{C}_p^r + \mathcal{C}_{oi} < \mathcal{L}_r(\mathcal{S})$

(b) $\mathcal{C}_p^r + \mathcal{C}_{oi} \geq \mathcal{L}_r(\mathcal{S})$

Figure 4.8: Link reception and protocol processing overlap in interrupt mode input.

tablished between the sender and receiver. $\mathcal{C}_i^{'}$ can then be computed using this aggregate real-time packet arrival rate, which could be substantially less than the maximum possible packet arrival rate $\frac{1}{\mathcal{L}_r(\mathcal{S}_{min})}$, expressed in packets/second. Without this assumption, however, worst-case best-effort packet arrivals must be assumed as above, since, in the absence of any real-time traffic, best-effort traffic could completely consume unutilized resources at the sender.

Computation of the worst-case CPU wait time due to a lower-priority handler must consider the delay before the handler begins processing the first packet of the message being considered, and is given by

$$\mathcal{T}_w^{intr,cpu} = \mathcal{C}_b^r + \left\lceil \frac{\max(0, \mathcal{C}_b^r - \min(\lfloor \frac{\mathcal{C}_b^r}{\mathcal{L}_r(\mathcal{S})} \rfloor, \mathcal{N}_p - 1)\mathcal{L}_r(\mathcal{S}))}{\mathcal{L}_r(\mathcal{S}_{min})} \right\rceil \mathcal{C}_{oi} + \mathcal{C}_{csp}^r,$$

where $\mathcal{C}_b^r = \mathcal{C}_p^{last,r} + (\mathcal{P}^r - 1)\mathcal{C}_p^r$ is the worst-case cost of processing a block of $\mathcal{P}^r$ packets. The second term corresponds to the best-effort packet arrival interrupts in the remaining time that the lower-priority occupies the CPU, after accounting for the arrival interrupts due to the rest of the packets of the (real-time) message whose CPU wait time we are considering. Note that we only account for best-effort packet arrival interrupts if all packets of the real-time message arrive before the lower-priority handler yields the CPU. As explained earlier, the second term can be reduced further if appropriate interrupt masking is supported by the adapter, and the aggregate real-time message rate is considered.

The total worst-case wait time can only be computed after accounting for any additional queueing delays experienced by a incoming packet on the adapter. This was not an issue at the sending host, where only one outgoing packet was allowed to reside on the adapter at any time. For an adapter designed to handle QoS-sensitive traffic, an arriving packet would have to wait for at most one maximum-sized lower-priority packet to be received by the host. Thus, the worst-case "link" wait time is given by $\mathcal{T}_w^{intr,link} = \mathcal{C}_r + \frac{\mathcal{S}}{\mathcal{B}_r}$, assuming that the adapter initiated the transfer of a lower-priority packet just before the arrival of the higher-priority packet. As before, the total message wait time is $\mathcal{T}_w^r = \mathcal{T}_w^{intr,cpu} + \mathcal{T}_w^{intr,link}$.

Polled Mode

In this mode the adapter does not interrupt the host processor; instead, the host processor periodically polls the adapter to detect newly-arrived packets. Polling, which is sometimes preferred over interrupts [173], helps amortize interrupt overhead over multiple packets per polling cycle at the expense of increased packet reception latency. As mentioned before, polling can be effective in preventing receive livelock [125]. We assume that polling is realized via clocked interrupts [173] such that a polling timer expires periodically, and at each polling instant inputs up to a certain number of packets (the *quota* in [125]). The quota is determined by the host processing capacity allocated to servicing the polling timer and inputing packets. We focus below on admission control extensions when packet reception is performed purely via clocked interrupts with quotas. Our analysis can be easily extended to the case where polling is performed by an explicitly-scheduled polling thread, as described in [125].

Suppose the host configures the polling timer to expire every $\mathcal{I}_{poll}$ time units, and the quota is q packets; that is, each time the polling timer expires, up to q packets are input from the network adapter. To admit channels, admission control must use the host CPU capacity available for handlers. Assuming the host has mechanisms to schedule a periodic timer, our analysis directly accounts for the CPU capacity consumed by the polling timer service routine. We assume that the adapter has some minimum intelligence to distinguish (and maintain separate input queues for) incoming best-effort and real-time traffic, such that the latter gets higher input priority.

Since arriving packets may have to wait for input by the polling timer, we also assume that the adapter has sufficient per-channel buffers for arriving packets, as derived from the channel traffic specification. Further, the adapter must ensure (via appropriate QoS-sensitive queueing) that the host can receive real-time packets from different channels in the order in which the sender transmitted them. This was not an issue in interrupt mode handling because arriving packets interrupt the host directly, and are immediately classified and enqueued at the corresponding channel's packet queue. Without the above assumptions we cannot compute message service time for a channel in isolation, and bound the link wait time (i.e., the queueing delay on the adapter).

The maximum number of packets of size $\mathcal{S}$ completing arrival at the adapter in a polling period is $n_I = \lceil \frac{\mathcal{I}_{poll}}{\mathcal{L}_r(\mathcal{S})} \rceil$. For practical purposes $1 \leq q \leq n_I$ must hold, i.e., $\mathcal{I}_{poll}$ is chosen such that at least q packets arrive and can be input without starving channel handlers. The total link reception time for all the packets comprising a message is $\mathcal{L}_m^r = (\mathcal{N}_p - 1)\mathcal{L}_r(\mathcal{S}) + \mathcal{L}_r(\mathcal{S}^{last})$, where $\mathcal{N}_p$ is the number of packets in the message. While all the packets of the message arrive over $\lceil \frac{\mathcal{L}_m^r}{\mathcal{I}_{poll}} \rceil$ polling periods, the total number of polling periods required to input the message is $\lceil \frac{\mathcal{N}_p}{q} \rceil$.

Since $n_I \geq q$, and hence $\lceil \frac{\mathcal{L}_m^r}{\mathcal{I}_{poll}} \rceil \leq \lceil \frac{\mathcal{N}_p}{q} \rceil$, it follows that all the packets of the message will arrive before the polling timer inputs all the packets and the channel handler finishes processing them. Once a block of q packets are input and queued, the corresponding channel handler processes them in time $\mathcal{C}_q = q\mathcal{C}_p^r + \lfloor \frac{q-1}{\mathcal{P}^r} \rfloor \mathcal{C}_{csp}^r$. The message service time can then be shown to be

$$\mathcal{T}_s^{poll} = \begin{cases} (\lceil \frac{\mathcal{N}_p}{q} \rceil - 1)\mathcal{I}_{poll} + (\mathcal{N}_p \bmod q)(\mathcal{C}_o + \mathcal{C}_p^r) + (\mathcal{C}_p^{last,r} - \mathcal{C}_p^r) & \\ \qquad + \lfloor \frac{\mathcal{N}_p \bmod q - 1}{\mathcal{P}^r} \rfloor \mathcal{C}_{csp}^r & \text{if } \mathcal{C}_q < \mathcal{I}_{poll} - q\mathcal{C}_o \\ \mathcal{C}_p^{m,r} + \mathcal{C}_o^m + \mathcal{C}_{pr}^r & \text{otherwise} \end{cases}$$

It can be verified that the two expressions are the same when $\mathcal{C}_q = \mathcal{I}_{poll} - q\mathcal{C}_o$. Given our assumption that $n_I \geq q$, the above equation is valid for $\mathcal{N}_p \leq n_I$ as well as $\mathcal{N}_p > n_I$. Figure 4.9 illustrates link reception and protocol processing overlap in polled mode packet input.

If $\mathcal{C}_q < \mathcal{I}_{poll} - q\mathcal{C}_o$ (Figure 4.9(a)), the processing of all the q packets input during a polling period will be completed before the next polling period. Accordingly, the message service time is determined by one less than the number of polling periods required to input $\mathcal{N}_p$ packets, and the time to input and process the remaining $\mathcal{N}_p \bmod q$ packets in the last polling period. If $\mathcal{C}_q \geq \mathcal{I}_{poll} - q\mathcal{C}_o$ (Figure 4.9(b)), however, it suffices to account for the time to input and process all the packets of the message.

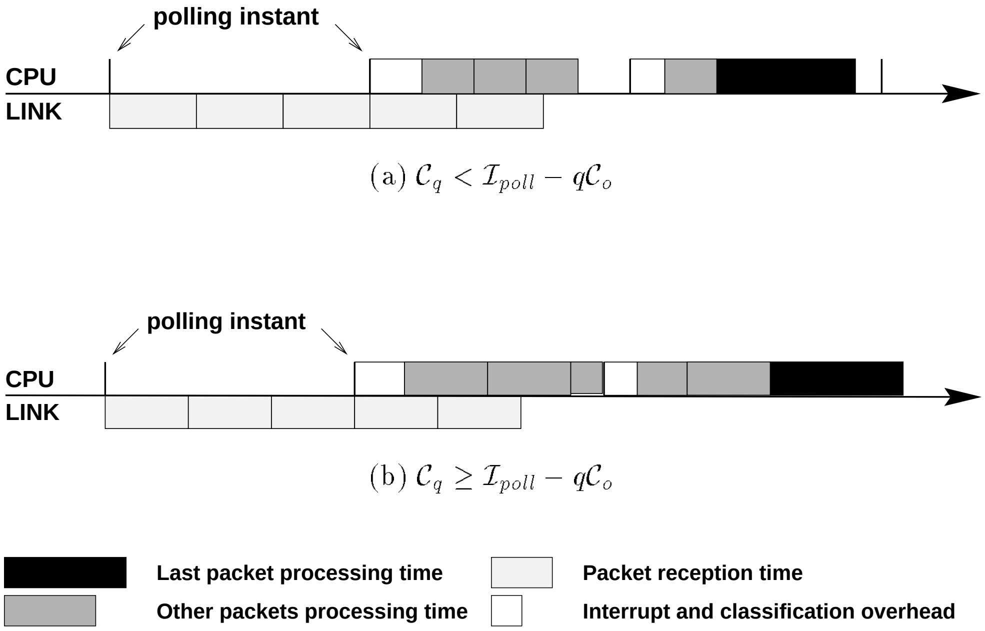


Figure 4.9: Link reception and protocol processing overlap in polled mode input.

The worst-case message wait time is incurred when the first packet of the message arrives at an empty adapter queue with the polling timer having just finishing polling; the arrived packet must wait until the next expiration of the polling timer (as shown in Figure 4.9). Further, a lower-priority handler could have been allocated the CPU just before the next expiration of the polling timer. To compute the worst-case CPU wait time due to a lower-priority handler, note that once the first packet of a message arrives, the corresponding (waiting) handler is unblocked if it is not already running (the worst-case scenario). In the absence of a higher-priority handler, the newly-awakened handler will be scheduled for execution only after the polling timer service routine completes (i.e., relinquishes the CPU), and the lower-priority handler that was preempted by the polling timer yields or completes processing (whichever occurs first). Accordingly, the worst-case CPU wait time is given by $\mathcal{T}_w^{poll,cpu} = \mathcal{I}_{poll} + \mathcal{C}_b^r + \mathcal{C}_{csp}^r$. Note that the wait time in this case is not affected by the back-to-back arrival of best-effort packets, since the host is not interrupted on packet arrival. Under the same assumptions of adapter support as in Section 4.5.2, the worst-case link wait time is $\mathcal{T}_w^{poll,link} = \mathcal{C}_r + \frac{\mathcal{S}}{\mathcal{B}_r}$, with the worst-case message wait time being $\mathcal{T}_w^r = \mathcal{T}_w^{poll,cpu} + \mathcal{T}_w^{poll,link}$.

4.5.3 Simultaneous Send and Receive

We now consider admission control extensions for a host that is both a source and destination for real-time channels, i.e., handles simultaneous outgoing and incoming traffic. We focus on the option O1 for outgoing traffic and interrupt mode handling for incoming traffic; appropriate extensions can also be developed for the other three possible configurations, such as option O1 and polled mode handling.

To make the analysis tractable, we must make additional assumptions regarding the support available on the adapter. We assume that the adapter isolates processing and queueing of incoming and outgoing traffic. This is a valid assumption; many modern high-performance adapters use separate processors and on-board logic for the transmit and receive paths. We further assume that the link is full-duplex, so that concurrent data transmission and reception can occur between two hosts on the (point-to-point) network link. Lastly, we assume that the host system bus arbitrates fairly between incoming and outgoing traffic, i.e., between the host and the adapter.

Under these assumptions, the message service times for outgoing and incoming messages must account for the worst-case extra interrupts due to simultaneous outgoing and incoming traffic. The worst-case message service time for an outgoing channel is given by

$$\mathcal{T}_s^{simul,O1} = \mathcal{T}_s^{O1} + \lceil \frac{\mathcal{T}_s^{O1}}{\mathcal{L}_r(\mathcal{S}_{min})} \rceil \mathcal{C}_{oi},$$

since minimum-sized packets could arrive back-to-back during the processing and transmission of an outgoing message. As before, this estimate can be improved by making additional assumptions about adapter and host support for simultaneous handling of best-effort and real-time traffic. For example, if the host does not allow any best-effort packets to be received if an outgoing real-time message is being processed, then the second term above can be reduced by considering the aggregate packet arrival rate $\sum_{a \in \mathcal{A}^r} \mathcal{R}_{max}^a \lceil \frac{\mathcal{M}_{max}^c}{\mathcal{S}} \rceil$ (packets/second) for the set (denoted by $\mathcal{A}^r$) of established channels with this host as the destination.

Similarly, the worst-case message service time for an incoming channel is now

$$\mathcal{T}_s^{simul,intr} = \mathcal{T}_s^{intr} + \lceil \frac{\mathcal{T}_s^{intr}}{\mathcal{L}_x(\mathcal{S}_{min})} \rceil \mathcal{C}_l,$$

since minimum-sized best-effort packets could be transmitted back-to-back during the reception and processing of an incoming message. Again, we can improve this estimate by assuming that the host does not allow any best-effort packet to be transmitted if an incoming real-time message is being processed, The second term can be reduced by considering the aggregate packet arrival rate $\sum_{a\in\mathcal{A}^s} \mathcal{R}^a_{max}\lceil\frac{\mathcal{M}^c_{max}}{\mathcal{S}}\rceil$ (packets/second) for the set (denoted by $\mathcal{A}^s$) of established channels with this host as the source.

The message wait time due to lower-priority handlers of packets would also be different. In particular, the worst-case CPU wait time would now be

$$\mathcal{T}_w^{simul,cpu} = \max(\mathcal{T}_w^{O1,cpu}, \mathcal{T}_w^{intr,cpu}),$$

without changing the worst-case link wait time. Correct admission control necessitates that outgoing and incoming channels be distinguished from each other, both in terms of the corresponding system parameters and in the service and wait time computations.

4.6 Summary

In this chapter, we focused on management of host communication resources for real-time communication. In particular, we identified the issues involved in extending and implementing resource management policies originally formulated using idealized resource models. Using our real-time channel implementation, we extended the admission control procedure to account for protocol processing and implementation overheads for two implementation paradigms realizing link scheduling at sending hosts. The extensions were validated against measured performance of the implementation and used to study the implications for channel admissibility. We also extended the admission control procedure at receiving hosts for two distinct packet input mechanisms, namely, interrupt mode and polled mode, as well as for hosts engaged in simultaneous data transmission and reception. The analysis and extensions developed in this chapter are applicable to other proposals for guaranteed real-time communication in packet-switched networks [8, 188].

Our main conclusions can be summarized as follows. In order to best utilize CPU and link bandwidth, one must account for implementation overheads that reduce useful resource capacity. Further, one must also consider the implications of the implementation paradigm adopted to manage CPU and link bandwidth. For sending hosts, we studied two implementation paradigms that both realize link scheduling but differ significantly in performance. Realization of the link scheduler as a dedicated thread degrades channel admissibility significantly (due to conservative admission control) since it couples link bandwidth allocation with CPU bandwidth allocation.

Implementing link scheduling in interrupt context provides significant performance advantages in terms of higher channel admissibility and reduced sensitivity to system parameters such as packet size and number of packets processed between preemptions. However, these advantages come at the expense of an unpredictable increase in the interrupt service time, which could be highly undesirable in systems with stringent response-time constraints. We note that the paradigm adopted for implementing link scheduling also depends on the software configuration chosen. For example, with a microkernel operating system, a thread based realization of the link scheduler becomes necessary if the QoS-sensitive communication subsystems is realized at user level.

While similar tradeoffs arise at receiving hosts, developing the admission control extensions necessitates additional assumptions regarding the support available from the network adapter. Without adequate adapter support, it would be almost impossible to provide QoS guarantees on communication. Even with these assumptions, correct formulation of the admission control extensions depends greatly on the QoS-sensitive mechanisms employed by the host to handle simultaneous incoming and outgoing traffic.

These extensions have been further refined and implemented in the guaranteed-QoS communication service described in Chapter 5, where we have considered additional overheads and constraints imposed by a microkernel operating system. Chapter 6 illustrates how the communication subsystem software can be structured to allow these extensions to work with self-parameterizing protocol stacks. Finally, Chapter 8 highlights the generality and applicability of these extensions to other protocol processing architectures, and explores their integration with QoS-sensitive application scheduling.

CHAPTER 5

GUARANTEED-QoS COMMUNICATION SERVICES

5.1 Introduction

The architecture described in Chapter 3 and the admission control extensions presented in Chapter 4 were implemented and evaluated for a relatively restricted environment. The communication executive exercised complete control over the processor, it was relatively easy to extend the underlying resource management policies to be QoS sensitive, and the API exported by the communication subsystem was relatively primitive. While the restricted environment allowed us to study subtle issues such as the relationship between CPU and link bandwidth, it did not allow us to explore QoS-sensitive communication subsystem design for contemporary operating systems.

A key issue then is to identify and resolve the challenges involved in realizing guaranteed-QoS communication services on contemporary operating systems, using appropriate enhancements of the techniques proposed in Chapters 3 and 4. This chapter describes the design, implementation, and evaluation of one such service for a microkernel operating system. Microkernel operating systems continue to play an important role in operating system design [45, 100], and are being extended to support real-time and multimedia applications [162]. With the continued upsurge in the demand for networked multimedia applications on the WWW, it is important, therefore, to examine realization of QoS-sensitive communication subsystems on contemporary microkernel operating systems. The issues and techniques explored in this chapter are also relevant to contemporary monolithic operating systems such as UNIX and its variants.

In this chapter we present our experiences in designing and implementing a guaranteed-

QoS communication service, also based on real-time channels, on OSF MK 7.2 [166], a microkernel operating system based on CMU Mach. Besides giving us access to full source code, OSF MK 7.2 allows us to exploit the Path abstraction provided in OSF's x-kernel-based `CORDS` framework [172]. Such a service would provide the underlying end-to-end real-time communication support to enable development of middleware services for distributed real-time applications.

5.1.1 Goals, Approach and Assumptions

As mentioned earlier, our primary goal is to explore the architectural issues and implementation tradeoffs that arise when realizing guaranteed-QoS communication services on microkernel operating systems. A complete service must export a QoS-aware API to applications that, in addition to allowing the specification of traffic characteristics and QoS requirements, also allows applications to indicate their intent to be recipients of QoS-sensitive traffic in a generic fashion. Moreover, applications serving as destinations for QoS-sensitive traffic must be allowed to participate in the end-to-end signalling for resource reservation. While we do not consider dynamic QoS negotiation and adaptation, QoS-aware APIs may also allow applications to specify a range of desired QoS levels and receive dynamic notifications regarding overload conditions [2, 104].

The communication subsystem must interface to the applications via appropriate mechanisms such as application libraries, maintain additional state regarding applications and active connections, and move data to/from applications as per the associated QoS requirements. Further, it must provide protocol support for end-to-end signalling and resource reservation within the host communication subsystem, in addition to support for QoS-sensitive data transfer. Depending on the implementation environment, this support may need to export appropriate interfaces to other communication modules and *map* its functionality to the appropriate local resource reservation facilities provided within the communication subsystem, the underlying operating system, or both. For correct admission control the costs imposed by any additional structuring constraints and overheads must also be accounted for. The support provided must also be sufficiently modular and generic to facilitate reuse for other middleware services and enable future extensions.

To examine the above-mentioned issues, we realize a new service architecture that extends the architecture and extensions presented in Chapters 3 and 4. This service architecture *integrates* three primary components: (i) RTCOP, a protocol that coordinates end-to-end signalling, QoS-sensitive resource allocation and reclamation, (ii) CLIPS, a support library that provides QoS-sensitive CPU scheduling for protocol processing and link scheduling of packet transmissions, and (iii) RTC API, the programming interface exported to applications that wish to use the service. These three components together ensure QoS-sensitive handling of network traffic at sending *and* receiving hosts.

We realize this service architecture as a user-level CORDS *server* running on the OSF MK 7.2 microkernel. Even though server-based protocol stacks perform poorly compared to user-level protocol libraries or in-kernel implementations [112, 168], we choose a server configuration for several reasons. A server configuration considerably eases software development and debugging, particularly the location and correction of timing-related bugs. More importantly, since several applications can establish multiple QoS connections, admission control and run-time resource management of these connections must be localized within one resource management domain. For most contemporary operating systems, this corresponds to a single protection domain and address space, in our case the CORDS server.

Further, in contrast with current trends in high-performance communication to transfer data as fast as possible, a QoS-sensitive communication subsystem can transfer data *only as fast as appropriate* as determined from the connection's traffic contract and desired QoS. In the worst case, an application can modify a buffer pending network transmission, forcing the server or the user-level library to make a copy in order to maintain data integrity. To build a guaranteed-QoS communication service, we must be conservative and account for the worst-case overheads and processing costs. Once developed and debugged, the server can be placed within the kernel to improve performance and hence relax conservative cost estimates. We note that a server-based configuration is regarded as a reasonable approach to build predictable communication services on microkernel operating systems [107]. Chapter **??** compares and contrasts various protocol processing architectures regarding the extent to which they facilitate per-connection QoS guarantees.

The functionality provided by our CORDS-based server cannot be truly effective without additional support from the underlying operating system. Such support could be in the form of capacity reserves for the service [123] or appropriate system partitioning [19], and would require proper integration of the QoS-sensitive communication subsystem with the operating system for provision of application-level QoS guarantees. We believe that the architectural support described in this chapter is complementary to the above support, and as such, the tradeoffs and issues highlighted are of significant relevance.

5.1.2 Outline

We describe a service architecture that employs well-defined interfaces to integrate the three components mentioned earlier and access local system resources. Using our prototype implementation we illustrate how this service architecture extends and exploits the `CORDS` framework to facilitate realization of guaranteed-QoS communication services. We highlight implications of a server-based configuration for provision of QoS guarantees. For example, link scheduling must be realized using option 2 (i.e., in thread context) specified in Chapter 3; a colocated server could utilize option 1 instead. We also highlight the difficulties faced and lessons learned in the course of the implementation.

We parameterize the `CORDS`-based communication subsystem via detailed profiling of the send and receive data paths. Based on this parameterization, we highlight certain overheads and constraints, imposed by the realization of `RTC API` in the `CORDS` server, that render the communication subsystem QoS-insensitive. We then propose enhancements to the run-time resource management architecture and admission control procedure to account for these overheads and constraints. An experimental evaluation in a controlled configuration demonstrates the efficacy with which QoS guarantees are maintained, under limitations of the inherent unpredictability imposed by the underlying operating system.

The rest of the chapter is organized as follows. We first present the goals and architecture of the real-time (guaranteed-QoS) communication service. The different components constituting the service are described next, with an emphasis on their internal design and interaction. The subsequent section describes our prototype implementation of the service and the tradeoffs we faced in realizing the service architecture. System profiling and parameterization of the experimentation platform and our implementation is presented next, followed by extensions to the service architecture to account for certain performance-related deficiences. We then present the results of an experimental evaluation of our prototype implementation to determine its efficacy in providing QoS guarantees. Finally, we conclude the chapter with a summary of the primary contributions and directions for future work.

5.2 Real-Time Communication Service

We first motivate the goals that our real-time communication service must satisfy. Subsequently, we highlight key aspects of the service architecture that facilitate meeting the stated goals.

5.2.1 Goals and Paradigm

Our primary goal is to provide applications and middleware with a service that can be used to request and obtain (real-time) guaranteed-QoS unicast connections between two hosts. Such a service must satisfy the three architectural requirements for guaranteed-QoS communication highlighted in Chapter 3: (i) maintenance of per-connection QoS guarantees, (ii) overload protection via per-connection traffic enforcement, and (iii) fairness to best-effort traffic. A secondary, but very important, goal is to design the service in a way that permits (a) constituent architectural components to be reused for other middleware services, and (b) flexibility and reuse in realizing other QoS paradigms and service models.

The secondary goal is important because multiple middleware services may need to co-exist and interoperate; reusing architectural components, whenever possible, makes service integration relatively easier. In order to realize generic components, the service model must be decoupled from the service architecture and its components. This in turn facilitates extension of the service architecture to more relaxed QoS models such as statistical guarantees, and QoS negotiation and adaptation [2]. To realize this service we adopt the real-time channel paradigm and base the service architecture on the architectural mechanisms and extensions described in Chapters 3 and 4.

5.2.2 Service Architecture

Figure 5.1 illustrates the software architecture of our guaranteed-QoS service at hosts; a subset of this architecture also applies to intermediate nodes. The core functionality of the service is realized via three distinct components that interact to provide guaranteed-QoS communication: real-time communication API (`RTC API`), `RTCOP`, and `CLIPS`. While applications use the service via the `RTC API`, end-to-end signalling for resource reservation and reclamation is coordinated by `RTCOP` and run-time management of resources for QoS-sensitive data transfer performed by `CLIPS`. As mentioned above, the run-time resource

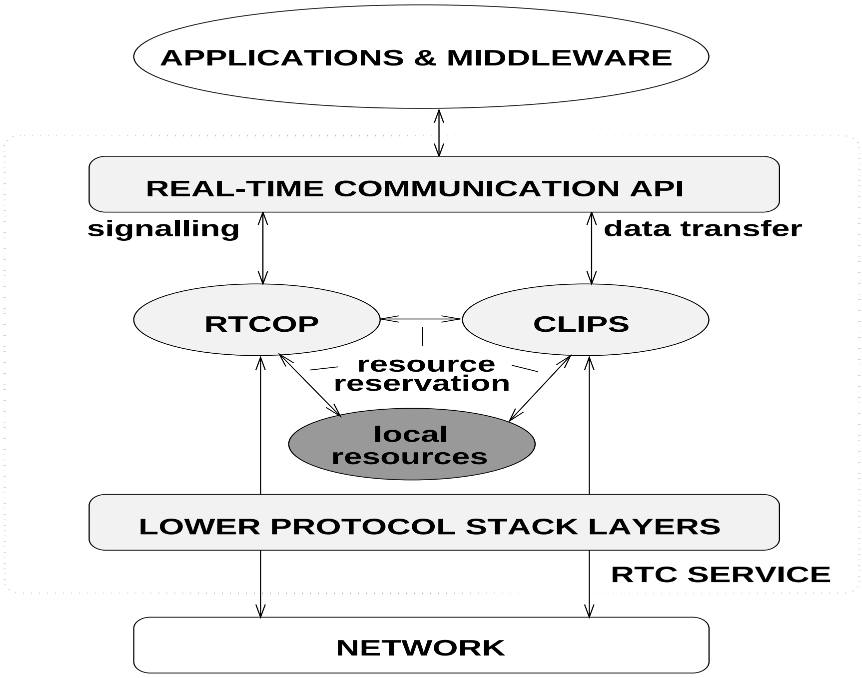


Figure 5.1: Real-time communication service architecture.

management in the service architecture is based on the architecture proposed in Chapter 3, with significant enhancements to accommodate the specific requirements of a full-fledged service and the available implementation environment. Below we give a brief overview of the three components of our service architecture; Section 5.3 gives internal details of the components and their interaction.

Invocation via RTC API: Applications establish and teardown guaranteed-QoS connections, and perform data transfer on these connections, by invoking routines exported by the `RTC API`. The API can be viewed as comprising two parts. The top half interfaces to applications and is responsible for validating application requests and creating internal state. The bottom half interfaces to `RTCOP` for signalling (i.e., connection setup and teardown), and to `CLIPS` for QoS-sensitive data transfer. As described in Section 5.3, the design of the API has been significantly influenced by the structure of the sockets API in BSD Unix [108] and its variants.

Signalling via RTCOP: End-to-end *signalling and resource reservation* is performed by `RTCOP` to establish and teardown guaranteed-QoS connections across the communicating hosts, possibly via multiple network nodes. `RTCOP` provides reliable datagram semantics for signalling requests and replies between nodes, and performs consistent channel state management at each node. It interfaces to the admission control module (that keeps track of available communication resources) to admit new connection requests, and establish appropriate connection state to store connection-specific information. Similarly, it interfaces to the routing module to select a (unicast) path on which to perform the end-to-end signalling. As shown in Figure 5.11, it interfaces to `CLIPS` and directly to local resources for resource reservation functions.

Data transfer via CLIPS: `CLIPS` facilitates *network data transport* on established real-time channels, providing services for management of CPU and link resources during data transfer in order to maintain QoS guarantees. For example, it provides services for allocation of protocol processing resources and fragmentation of application data (messages) into smaller units (packets) for network transmission. In addition to managing CPU resources for protocol processing, `CLIPS` also performs *link access scheduling* to manage link bandwidth such that all active connections receive their promised QoS. This involves abstracting the link in terms of transmission delay and bandwidth, and scheduling all outgoing packets for network access. The minimum requirement for provision of QoS guarantees is that packet transmission time on the link be bounded and predictable. `CLIPS` also performs *traffic enforcement* on a per-channel basis, forcing an application to conform to the channel's traffic specification and provide overload protection between established connections.

5.3 Service Components and Their Interaction

We begin by describing the service API exported to applications and its internal functionality. This is followed by the design of `RTCOP` and the various interfaces it exports and utilizes. Finally, we present the design of `CLIPS` and how the three components interact with each other to provide QoS guarantees.

5.3.1 Service Invocation via RTC API

The API exported to applications comprises routines for establishment and teardown of real-time channels, message transmission and reception during data transfer on established channels, and initialization and support routines. Table 5.1 lists the routines currently available in `RTC API`.

The design of `RTC API` is based in large part on the well-known sockets API in BSD

Routines	Invoked By	Function Performed
Miscellaneous		
`rtcInit`	sender, receiver	service initialization
`rtcGetParameter`	sender, receiver	query parameter on specified real-time channel
Signalling		
`rtcRegisterPort`	receiver	register local port and agent for signalling
`rtcUnRegisterPort`	receiver	unregister local signalling port
`rtcCreateChannel`	sender	create channel with given parameters to remote endpoint ; return channel identifier
`rtcAcceptChannel`	receiver	obtain the next channel already established at specified local port
`rtcDestroyChannel`	sender	destroy specified real-time channel
Data Transfer		
`rtcSendMessage`	sender	send message on specified real-time channel
`rtcRecvMessage`	receiver	receive message on specified real-time channel

Table 5.1: Routines comprising `RTC API`**.**

Unix. Each real-time channel endpoint is a 2-tuple `(IPaddr, port)` formed by the IP address of the host (`IPaddr`) and an unsigned 16-bit port (`port`) unique on this host, similar to an INET domain socket endpoint. In addition to unique endpoints for data transfer, an application may use multiple endpoints to receive signalling requests from other applications. Applications willing to be destinations of real-time channels are assumed to register their signalling endpoints with a name service or use well-known ports. Applications wishing to create real-time channels to receiving applications must first locate the corresponding endpoints before signalling can be initiated.

Each of the signalling and data transfer routines in Table 5.1 has its counterpart in the sockets API. For example, the routine `rtcRegisterPort` corresponds to the invocation of `bind` and `listen` in succession, and `rtcAcceptChannel` corresponds to `accept`. Similarly, the routines `rtcCreateChannel` and `rtcDestroyChannel` correspond to `connect` and `close`, respectively. The key aspect which distinguishes `RTC API` from the sockets API is that the receiving application *explicitly approves* the establishment and teardown of real-time channels. This is unlike the establishment of a TCP connection, for example, which is completely transparent to the peer applications. We note that extensions to the sockets API have been proposed to support the requirements of QoS-sensitive traffic, most of these realized via new socket options. The architecture described in Chapter 7 adopts a unique *control socket* approach to realize a QoS-aware sockets layer [13].

Signalling Routines

rtcRegisterPort, rtcUnregisterPort: An application indicates its intent to receive signalling requests for real-time channels by registering a signalling port with `RTCOP` via `rtcRegisterPort`. The application also specifies an *agent* function that the service must invoke in response to incoming signalling requests. This function, implemented by the receiving application, determines whether sufficient resources are available for the channel being requested. One of the responsibilities of this function is to reserve resources (i.e., allocate CPU capacity, application buffers, etc.) for the new channel, if available, based on the request received, the computation resource requirements, and resource availability. This agent function may also perform authentication checks, if needed, based on the requesting endpoint specified in the signalling request. Since communication resource management is performed by the RTC service, the application need only be concerned with management of computation resources.

Note that the application can register multiple signalling ports and the corresponding agent functions, allowing it to specify different agents for different classes of traffic. For example, the application may register a signalling port for video traffic and a different one for audio traffic. Clients (or senders) would then send signalling requests to the appropriate receiver port. The application can "free" a signalling port previously registered by invoking `rtcUnregisterPort`.

rtcCreateChannel, rtcDestroyChannel: Once the receiver registers its intent to receive, the sender invokes `rtcCreateChannel` to request establishment of a real-time channel to a remote (receiving) endpoint. This endpoint corresponds to the signalling port registered by the receiver to receive signalling requests. Besides specifying the remote endpoint, the sender must provide the traffic and QoS specification described in Section 5.2.1, and the desired overload protection policy. The overload protection policy specifies the behavior under violation of traffic specification on the corresponding channel, with the choice between blocking the application, dropping the violating message in entirety, or simply returning an error. If channel establishment is successful, a unique channel identifier is returned to the application for later reference.

When the sender no longer needs an existing channel, it invokes `rtcDestroyChannel` using the channel identifier returned earlier when the channel was established. If the channel is successfully closed, all resources allocated to this channel are released and the channel identifier reclaimed.

rtcAcceptChannel: The receiving application invokes `rtcAcceptChannel` to obtain the next real-time channel established at a given signalling port previously registered with the RTC service. We observe that the receiving application gets notified of incoming real-time channels only *after* acceptance by the application's agent function.

Data Transfer Routines

rtcSendMessage: The sender invokes `rtcSendMessage` to send a message buffer on an previously-established channel. If the application is generating traffic as per the channel's traffic specification, this routine is non-blocking, i.e., returns after the message has been copied in and enqueued for protocol processing, without waiting for the message to be transmitted. Copying the message ensures that the application does not subsequently modify the

RTCOP Requirements
Internals
network and transport layer functionality
best-effort, reliable signalling semantics
management of channel state
simplicity, isolation of real-time channel information
Interfaces
interface to `RTC API`
interface to `CLIPS`
interface to admission control (internal)
interface to system resources for reservation
interface to routing engine to query routes

Table 5.2: `RTCOP` functional requirements.

contents of the message before it has been transmitted.[1] If the channel's traffic specification is violated, the RTC service *may* block the application, especially if communication buffers overflow, if the application so specifies.

rtcRecvMessage: This routine receives the next message on a previously-established channel into an application buffer, and can be blocking or non-blocking. If no message is queued for reception, and the call is non-blocking, the routine returns immediately indicating failure. If the call is blocking, the application is blocked until a message is queued for reception. If a message is waiting, the routine retrieves the message and returns success.

5.3.2 RTCOP-based Signalling and Resource Reservation

Application requests to create and destroy channels are forwarded by the `RTC API` to `RTCOP`, which performs end-to-end signalling, as mentioned in Section 5.2.2. Table 5.2 lists the requirements that `RTCOP` must satisfy in order to meet the service goals outlined earlier, and Figure 5.2 illustrates its internal structure and external interfaces.

[1] Later versions of the RTC implementation may explore buffer management techniques to eliminate copying overhead.

External Interfaces

`RTCOP` exports an interface to `RTC API` for specification of channel establishment and teardown requests and replies, and selection of logical ports for channel endpoints. The `RTC API` uses the latter to reserve a signalling port in response to an `rtcRegisterPort` request from the application, for example. `RTCOP` also utilizes four external interfaces during signalling: an interface to the routing engine, an interface to `CLIPS`, and an interface to local system resources.

Interface to routing engine: Before initiating signalling for a new channel, `RTCOP` queries an appropriate route via its interface to the underlying routing engine. In general, the routing engine would find a route that can support the desired QoS requirements, and `RTCOP` would perform signalling on this route. The primary benefit of QoS-based routing is an increased likelihood of channel establishment and satisfaction of application requirements. However, for simplicity we use static (fixed) routes for channels since it suffices to demonstrate the capabilities of our architecture and implementation. We note, however, that using static routes may result in denial of service to an application if resources along the route are insufficient. Development of QoS-based routing algorithms is beyond the scope of this dissertation.

Interface to CLIPS: As described in Section 5.3.3 below, `CLIPS` performs run-time resource management and scheduling for messages transmitted or received on real-time channels, and exports an interface that can be used to avail of these services. `RTCOP` uses this interface to request such services for each new channel to be established.

Interface to local system resources: In some situations the host operating system may provide special resource management support for real-time communication. This may include support for QoS-sensitive buffer management and/or CPU scheduling. Such support is generally platform-dependent and often non-standard, but may increase the utility and performance of the service. For example, our prototype implementation (Section 5.4) realizes input packet queues and buffers for channels via the support available in the underlying communication subsystem. `RTCOP` avails of the local system resources directly through the available native interface.

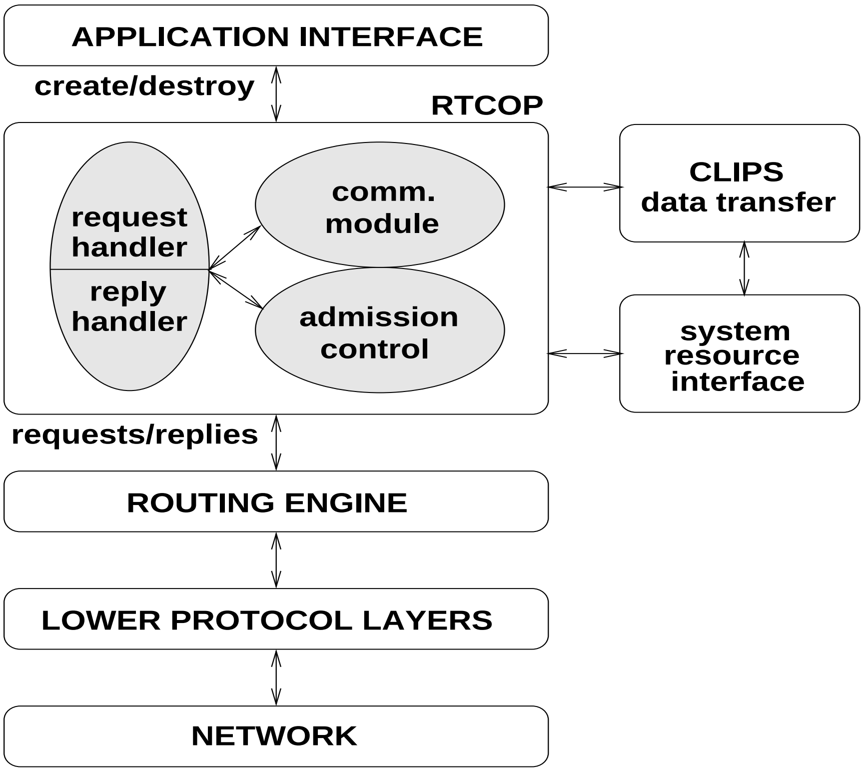


Figure 5.2: `RTCOP` internal structure and interfaces.

Internal Structure

As illustrated in Figure 5.2, `RTCOP` is organized internally as three primary modules. The *request and reply handlers* generate and process signalling messages, and invoke resource reservation and reclamation operations as needed. When processing a new signalling request, the request handler performs a multi-step admission control procedure to decide whether or not sufficient resources are available for the new request. It first checks with the *admission control module*, which currently implements the extended `D_order` algorithm to perform schedulability analysis for CPU and link bandwidth allocation. The admission control module decides if the new channel can be admitted and notifies `RTCOP`, which continues processing the request as per the notification. Note that the admission control module is functionally independent and need not be internal to `RTCOP`. However, we have subsumed it in `RTCOP` since admission control is most closely associated with signalling. `RTCOP` then requests `CLIPS` to allocate channel resources such as the message queue and buffers, the channel handler and CPU bandwidth for protocol processing, and link bandwidth for packet transmissions. As mentioned earlier, it obtains the input packet queue

CLIPS Requirements
Internals
prioritized message communication with guarantees
generic preemptible CPU bandwidth allocation
generic link bandwidth allocation
independence from real-time channel semantics
independence from protocol stack
wide applicability (unicast, multicast or broadcast)
Interfaces
interface exported to `RTC API` and `RTCOP`
interface to system resources for reservation

Table 5.3: `CLIPS` functional requirements.

and buffers via the system resource interface. Consistent channel state management at all nodes is an essential function of `RTCOP`.

The *communication module* handles the basic tasks of sending and receiving signalling messages, as well as forward data packets to and from the applications. Signalling messages are transported as best-effort traffic but delivered reliably using source-based retransmissions. Reliable signalling ensures that a channel is considered established only if channel state is successfully installed and sufficient resources reserved at all the nodes along the route. Duplicate suppression ensures that multiple reservations are not installed for the same channel establishment request. Similar considerations apply toward channel teardown where all nodes along the route must release resources and free channel state.

5.3.3 CLIPS-based Resource Scheduling for Data Transfer

Application requests to send and receive messages are forwarded to `RTC API`, which interacts with `CLIPS` to perform the desired data transfer. Table 5.3 lists the functional requirements that `CLIPS` provides for and Figure 5.3 illustrates how `CLIPS` coordinates data transfer in relation to the communication protocol stack.

`CLIPS` provides a generic mechanism for prioritized, time-bounded message processing and communication at end hosts. The design of `CLIPS` has been motivated by the requirements of the different real-time communication and middleware services. For example, a

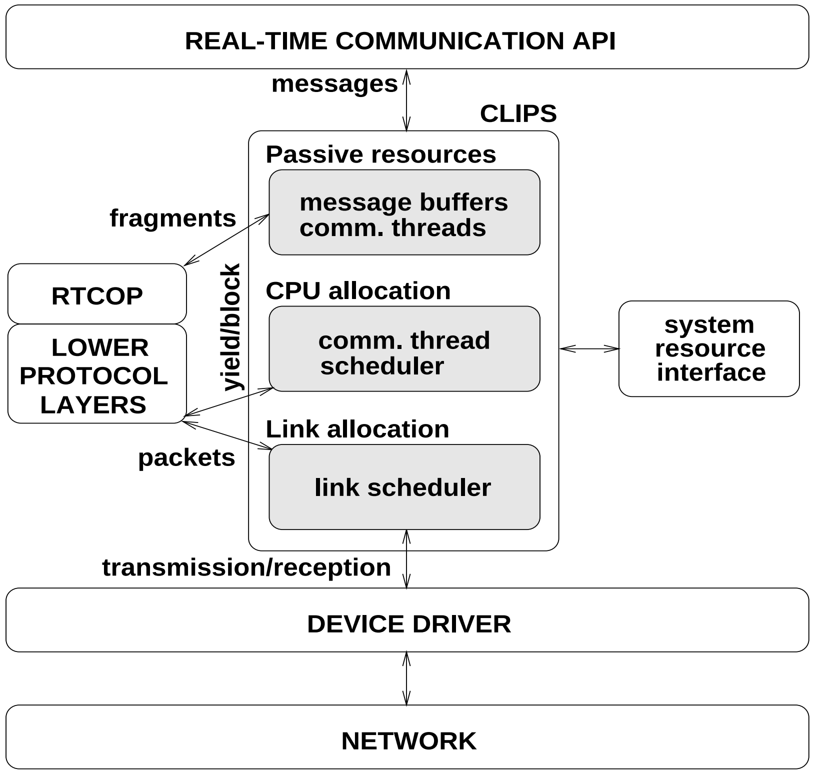


Figure 5.3: `CLIPS` **internal structure and interfaces.**

real-time multicast and membership service needs prioritized message scheduling to guarantee receipt of messages by the corresponding destinations by the specified deadline. Though modeled after the communication subsystem architecture described in Chapter 3, `CLIPS` is applicable wherever a priority-based messaging mechanism is required. Being largely independent from the semantics of a particular communication service, `CLIPS` provides generic support for CPU and link bandwidth allocation. As explained below, this support can be easily customized to manage data traversing real-time channels in a QoS-sensitive fashion.

Key Features

A connection endpoint requiring prioritized message communication must be registered with `CLIPS`, which creates a *clip* between the (external) protocol stack and the local message priority mechanisms at the specified communication endpoint. Note that a clip is similar to traditional Unix socket that has been extended with the necessary reservation states, as in [13]. However, compared to a socket a clip has more state and communication resources associated with it, for both incoming and outgoing traffic. Associated with each clip is a *message queue* to buffer messages generated or received on the corresponding endpoint, a *communication handler thread* to process these messages, and a *packet queue* to stage packets waiting to be transmitted or received. Once a clip is created for a connection endpoint, messages generated on the endpoint can be transferred in a prioritized fashion using the `CLIPS` API. Messages sent via registered endpoints can be prioritized in accordance with application semantics, with the priorities being set dynamically. Thus, two different messages sent via the same endpoint need not have the same priority.

QoS-sensitive CPU allocation: The handler thread executes in a continuous loop either dequeueing messages from the message queue and fragmenting them (at the source host), or dequeueing packets from the packet queue and reassembling messages (at the destination host). It inherits its execution priority dynamically from the priority of the message (or packet) it processes. The handler starts execution within `CLIPS`, continues through the user-defined protocol stack, and returns to `CLIPS` after processing each message/packet. Each clip is assigned a *CPU budget* based on the message processing requirements of the associated connection, estimated from the maximum number of packets constituting a message. The budget is, therefore, specified as the maximum number of packets the handler is allowed to process within a given time period, and is replenished periodically. The CPU bandwidth consumed by a handler is estimated from the count of processed packets and charged against the corresponding clip's budget. *Conformant* messages (i.e., those that arrive while the handler has non-zero budget), are served in the order of message priority.

Communication thread scheduling: The communication threads eligible for CPU allocation are multiplexed on the CPU by a priority-based *communication thread scheduler* which supports static as well as dynamic handler priorities. Recall that a handler inherits the priority of the message it processes and has an invocation period associated with it. This has two implications. One, the handler must explicitly reset its priority every time it finishes processing one message and starts processing a new message due to excess processing budget. This situation can arise, for example, when a burst of small messages is generated on the corresponding connection. Two, each time the handler's budget expires and there are pending messages, the handler is rescheduled with the priority of the next invocation, which corresponds to the priority of the next message to be processed.

Associating a budget with each handler facilitates *traffic enforcement*, i.e., appropriate handling of *non-conformant* traffic on the corresponding connection. This is because a handler is scheduled for execution only when the budget is non-zero, and the budget is not replenished until the next (periodic) invocation of the handler. These mechanisms together ensure that high-priority misbehaving connections do not consume excessive system resources at the expense of lower priority connections. An executing handler also implements cooperative preemption by voluntarily yielding the CPU after processing a certain number of packets. Thus, while processing messages, a handler may be rescheduled by the communication thread scheduler either when the handler blocks due to expiration of its budget, or it yields the CPU; this is illustrated in Figure 5.3.

QoS-sensitive link bandwidth allocation: At the source host, the packets generated by a handler are scheduled for transmission by a dynamic-priority *link scheduler*, as shown in Figure 5.3. The link scheduler, which implements the EDF scheduling policy using a priority heap for outgoing packets, is invoked by events such as packet transmissions on an idle link or completion of packet transmission. Packets on best-effort clips are maintained in a separate packet heap and serviced at a lower priority than those on real-time clips. Even though handlers execute and generate packets in priority order, packets need to be explicitly scheduled for transmission to limit the extent of priority inversion in the device driver and/or the network adapter.

Type of clips: `CLIPS` distinguishes between *outgoing*, *incoming*, or *bidirectional* connection endpoints. Incoming and outgoing endpoints differ primarily in the execution behavior of the corresponding message handler thread. At the source host (outgoing endpoint), the handler thread dequeues buffered outgoing messages and shepherds them down the user-defined protocol stack. At the destination side (incoming endpoint), the handler thread shepherds arrived packets up the protocol stack and queues up received messages in a priority heap for subsequent retrieval by the application. A bidirectional endpoint has two handler threads, one for sending and one for receiving messages. However, their budget is charged to the same capacity reserve. Each endpoint is further classified as *real-time* or *best-effort*, with real-time endpoints given priority over the best-effort ones.

Generality of CLIPS: `CLIPS` has been designed for use with a variety of middleware services and real-time communication paradigms, and hence is independent of the semantics associated with real-time channels. `CLIPS` does not make any assumptions about the composition of the user-defined protocol stack either. The protocol stack is treated as a black box of bounded execution time, with the prioritized resource management in `CLIPS` only considering handler execution time and priority to make scheduling decisions. Further, CLIPS need not be aware of the type of communication (unicast, multicast or broadcast) associated with application endpoints that are registered with it. CLIPS provides a priority enforcement mechanism independent of the (external) policy for message priority assignment.

Using CLIPS for real-time channels

We use CLIPS to provide real-time channel endpoints with QoS-sensitive allocation of CPU and link bandwidths. During channel establishment at a host, RTCOP creates a clip for the new channel using the CLIPS API. The priority of the clip (and hence of all the messages generated on this channel) is set to the local delay bound computed by D_order during admission control. Similarly, the period for invocation of the communication handler corresponds to I_{min}. The budget for the clip is set to the maximum number of packets corresponding to a message of size M_{max} bytes, derived using the maximum amount of data bytes that would fit in the MTU of the attached network. The number of message queue entries allocated for the clip corresponds to B_{max}.

5.4 Prototype Implementation: Environment and Configuration

We have developed a prototype implementation of the service architecture described in the preceding sections. Below we describe the experimental testbed and implementation environment, followed by a description of the implementation approach adopted and the CORDS-based service protocol stack. The myriad issues and tradeoffs we considered when realizing the different service components are presented in Section 5.5.

5.4.1 Testbed and Implementation Environment

Our testbed (see Figure 5.4) comprises several 133 MHz Pentium-based PCs connected by a Cisco 2900 100 Mb/s Ethernet switch, with each PC connected to the switch via 10 Mb/s Ethernet. Each PC runs the MK 7.2 microkernel operating system from the Open Software Foundation (OSF) Research Institute [166]. While not a full-fledged real-time OS, MK 7.2 includes several features that facilitate provision of QoS guarantees. Specifically, the

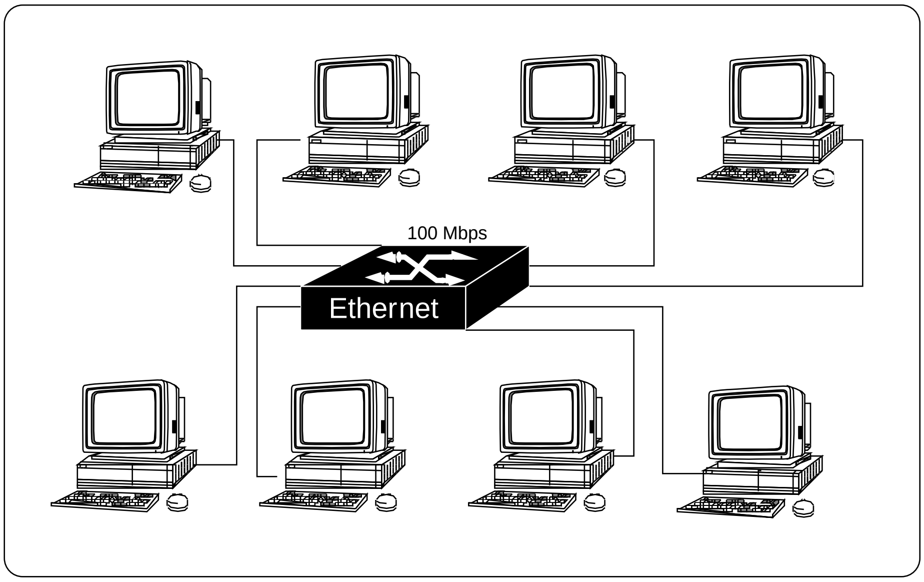


Figure 5.4: Experimental testbed.

7.2 release includes the CORDS (Communication Object for Real-time Dependable Systems) protocol implementation environment [172] in which our implementation resides.

Our implementation approach is to utilize and extend the functionality and facilities provided in OSF's CORDS environment. CORDS is based on the x-kernel object-oriented networking framework originally developed at the University of Arizona [76], with some significant extensions for controlled allocation of system resources. The primary advantage of using CORDS for our prototype implementation is the support for *paths*. As discussed in Section 5.4.2 below, paths play a significant role in the realization of our service architecture on this platform. Another advantage of using CORDS is the ease of composing protocol stacks in the x-kernel networking framework, in which a communication subsystem is implemented as a configurable graph of protocol objects. More details on the x-kernel can be found in [76, 141].

5.4.2 OSF Path Framework: Implications and Extensions

While preserving the structure and functionality of the original x-kernel, CORDS adds two abstractions, *paths* and *allocators*, to provide path-specific reservation/allocation of system resources. System resources associated with paths include dynamically allocated memory, input packet buffers, and input threads that shepherd messages up a protocol stack [172]. Paths, coupled with allocators, provide a capability for reserving and allocating resources at any protocol stack layer on behalf of a particular class of messages. With packet demultiplexing at the lowest level (i.e., performed in the device driver), it is possible to isolate packets on different paths from each other early in the protocol stack. Incoming packets are stored in buffers explicitly tied to the appropriate path and serviced by threads previously allocated to that path. Moreover, threads reserved for a path may be assigned one of several scheduling policies and priority levels.

Paths facilitate realization of a connection-oriented protocol stack, since the knowledge of paths is available at any layer in the protocol stack. While `CORDS` paths have been crucial for our implementation, they have some drawbacks that we have circumvented where possible, as described below.

Path-based packet classification: `CORDS` associates outgoing and incoming packets with paths, thus requiring data link drivers to examine outgoing packets and add the appropriate path identifier. While this is natural for networks supporting a notion of virtual circuit identifiers (VCI) such at ATM, it is not so for traditional data link technologies such as Ethernet. In the case of Ethernet, the `CORDS` driver adds a new *path identifier* to the data link header. This creates a non-standard Ethernet header that would not be understood by hosts not running OSF's `CORDS` framework. Note that this problem does not arise with the default path, which is used to transport traditional best-effort traffic, since there is no change in the data link header.

Global path name space: Since real-time channels provide QoS guarantees on a per-connection basis, it is natural to assign a distinct path to each channel. In order to realize our end-to-end service architecture, however, traffic on a particular path must be serviced consistently (i.e., according to the QoS associated with that path) by the communication subsystem on *all* hosts and routers. This in turn requires that the path name space be global across all hosts participating in the real-time communication service. To realize a global path name space, we utilize globally unique host names to construct unique path identifiers for real-time channels, as described later in Section 5.4.4.

Dynamic path creation/deletion: The `CORDS` framework envisions a relatively static use of paths, with a single path for best-effort traffic and a few paths for different classes of traffic. That is, there are never more than perhaps ten active paths, all of these long-lived and preconfigured. Accordingly, the `CORDS` path library provides no support for path tear-down or resource reclamation, operations which are necessary in a dynamic environment

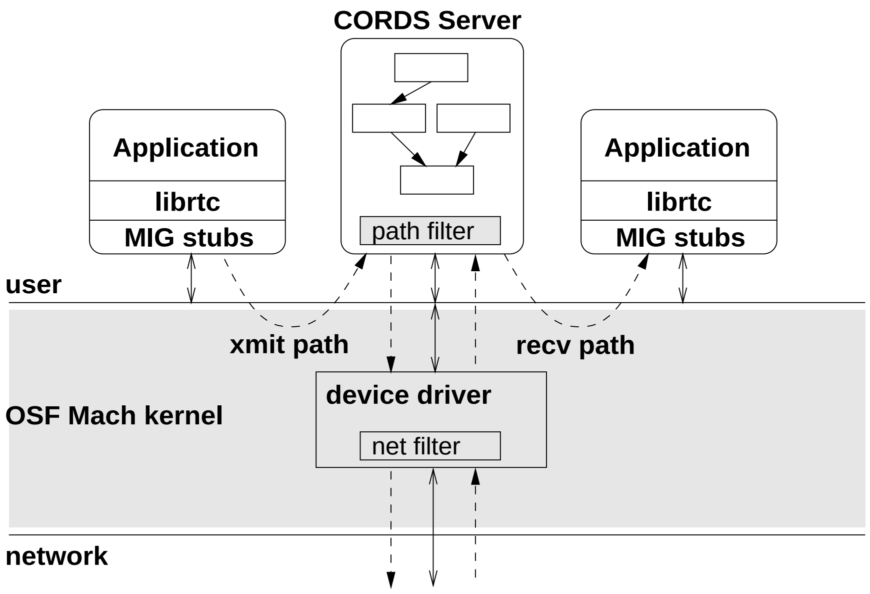


Figure 5.5: Service implementation as CORDS server.

in which paths are created and destroyed frequently. To facilitate a one-to-one association between real-time channels and paths, we have extended the CORDS framework to support path destruction and reclamation of resources associated with a path, as described in Section 5.4.4.

5.4.3 Server Configuration: Pros and Cons

Figure 5.5 shows the software configuration for our service implementation using the CORDS framework. As shown in Figure 5.5, applications link with a library, `librtc`, which implements `RTC API` and interfaces to the RTC service on behalf of the application, as described in Section 5.5.1. While the CORDS framework can be used at user-level as well in the kernel, we have developed the prototype implementation as a user-level CORDS server. There are several reasons for this choice, as mentioned in Section 5.1 and also explained below. The most obvious is the ease of development and debugging, resulting in a shorter development cycle.

Another important reason is the infeasibility of the other alternative (pure in-kernel development) available for implementation. CORDS allows an x-kernel protocol graph to span address space (and protection) boundaries via *proxies* [58]. Thus, our implementation could span user and kernel spaces, with portions of our implementation developed in user space and moved into the kernel after debugging and testing. However, this was not feasible for two primary reasons: (i) strong inter-dependencies between the service components, and (ii) interaction of CLIPS with multiple layers (especially the top and bottom anchors) of the protocol stack. It was important, therefore, to keep all service components in one address space, i.e., in the CORDS server. Moreover, use of proxies would force applications to transparently link with various other x-kernel libraries, in addition to our service library, resulting in needless code bloat and potentially degraded performance.

The final reason is related to performance. A server-based implementation is natural for a microkernel operating system, but may perform poorly compared to user-level protocol libraries due to excessive data copying and context switching [112, 168]. As mentioned in Section 5.1, it seems appropriate to be conservative when building a guaranteed-QoS communication service. It follows that in the worst case, compared to user-level libraries a server configuration only suffers from additional context switches. While this has significant implications for small messages, the relative degradation in performance is not as significant for the large data transfers performed via the guaranteed-QoS communication service, although it may affect connection admissibility.

A CORDS server-based implementation presents a number of significant problems for data input and output in our service architecture. These problems are either easily circumvented or simply do not arise for the in-kernel CORDS protocol stack or when the CORDS server is colocated in the kernel. The bottom layer of the CORDS protocol stack interfaces with the kernel device driver via a Mach device port each for input and output. As described in Section 5.4.4, device output is initiated by the link scheduler that is implemented here, as close as possible to the device driver without being in the kernel.

However, being in user space, the link scheduler cannot be invoked directly by the kernel device driver in response to transmission completion interrupts since OSF MK 7.2 does not support mapped device drivers or user-level upcalls. That is, link scheduling via option O1 in Chapter 3 cannot be realized. However, since OSF MK 7.2 allows user-level threads to perform synchronous device transfers, option O2 can be easily realized, as described in Section 5.4.4. Another potential problem arises from the multiplexing of IP traffic from the Unix server and traffic from the CORDS server into the same kernel interface queue.

Device input also presents significant challenges for QoS-sensitive handling of data. The CORDS-enabled kernel device driver is capable of routing incoming packets associated with a path to the corresponding pool of packet buffers and shepherd threads (if a pool has been

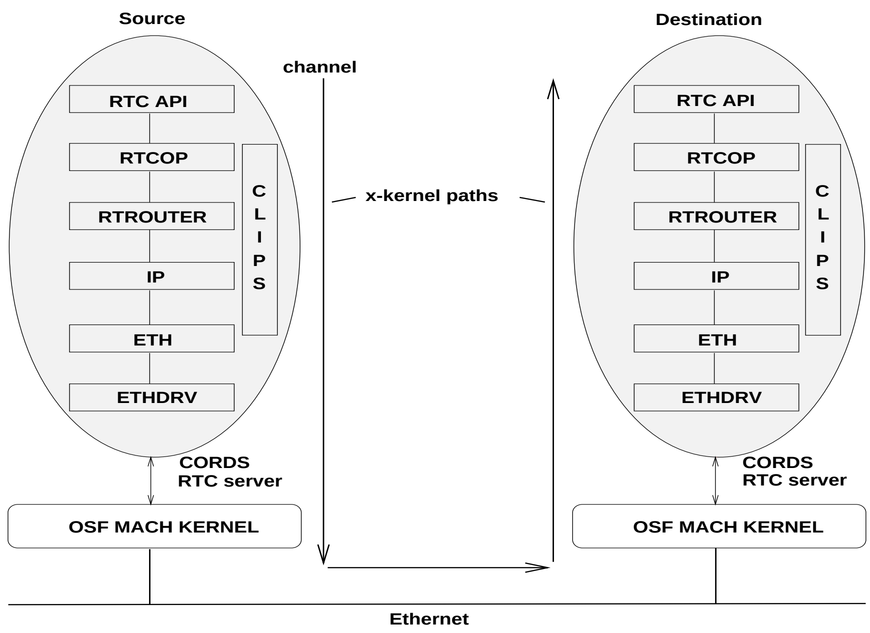


Figure 5.6: CORDS-based service protocol stack.

allocated), as illustrated by the *path filter* in Figure 5.5. However, the path filter is only applied for the in-kernel realization of CORDS; for the user-level CORDS server configuration, incoming packets are delivered to the corresponding device input port on the basis of a generic *net filter* set up when the server is initialized. Subsequent demultiplexing of packets to paths is performed by a device input thread at the lowest layer in the CORDS protocol stack. Besides making the packet classification a more expensive multi-step operation, such a configuration also introduces a common point of FIFO queueing for all data destined for the CORDS server. We discuss approaches to circumvent this later.

5.4.4 CORDS-based Service Protocol Stack

The CORDS-based protocol stack for our service is shown in Figure 5.6. Protocols comprising this stack include RTC API ANCHOR, CLIPS, RTCOP, RTROUTER, IP, ETH, and ETHDRV. We briefly describe these protocols below, and discuss the implementation of RTC API ANCHOR, RTCOP and CLIPS at length in Section 5.5.

RTC API ANCHOR: The anchor protocol for RTC API is the top-level protocol configured directly above RTCOP. It interfaces on the top with the applications via Mach Interface Generator (MIG) stubs, translating application requests to specific invocations of operations on RTCOP (for signaling) or CLIPS (for data transfer) at the bottom.

CLIPS: CLIPS spans several layers of the protocol stack, exporting an interface that is used by RTC API, RTCOP, and ETHDRV. CLIPS provides a generic mechanism for prioritized, time-bounded message processing and communication at end hosts. As described in Section 5.3, it implements support for associating communication endpoints of a protocol stack with clips, and allocating resources to each clip.

RTCOP: RTCOP is realized as an x-kernel transport protocol residing above a two-part network layer composed of RTROUTER and IP. It exports an interface to RTC API for specification of channel establishment and teardown requests and replies, and selection of logical ports for channel endpoints. RTCOP utilizes four interfaces during signalling: an (internal) interface to admission control, an interface to RTROUTER, an interface to CLIPS, and an interface to local path-specific CORDS resources such as packet queues, buffers, and thread pools.

RTROUTER: Real-time channels currently use the default IP routing. However, to keep the routing interface independent of IP, RTROUTER is provided as a go-between protocol. RTROUTER is intended to allow RTCOP to eventually work with more sophisticated routing protocols that support QoS- or policy-based routing.

IP, ETH, ETHDRV: The IP, ETH, and ETHDRV protocols are standard implementations distributed with the CORDS framework. ETH is a generic hardware-independent protocol that provides an interface between higher level protocols and the actual Ethernet driver. ETHDRV is specific to a particular user-level implementation of the CORDS server. It is an out-of-kernel device driver that interacts with the network device driver in the Mach kernel through system calls to a Mach device control port.

5.5 Realization of the Service Architecture

This section considers the realization of the service architecture, i.e., each of its constituent components, on our prototype implementation environment and configuration. We concentrate primarily on resolving the challenges faced and the degree of functionality provided by each component.

5.5.1 Service Library: librtc

The `librtc` library implements the `RTC API` presented in Section 5.2. `librtc` is in turn realized on top of MIG interface stubs which provide the necessary support for Mach IPC. Each library routine translates to an IPC call to the top-level anchor protocol `RTC API ANCHOR` in the `CORDS` server, which interacts with the service protocol stack on behalf of the service library.

As mentioned in Section 5.3, each real-time channel endpoint is a 2-tuple (`IPaddr`, `port`) formed by the IP address of the host (`IPaddr`) and an unsigned 16-bit port (`port`) that is unique on this host. Ports for channel endpoints are maintained by `RTCOP`, as discussed in Section 5.4.4. Transparent to the application, each channel endpoint is associated with a Mach port allocated at the time of channel establishment. For an application establishing a channel (i.e., a sender), the Mach port is allocated by the `CORDS` server, while for an application accepting a channel (i.e., a receiver), the Mach port is allocated by `librtc`.

The following functions are performed by `librtc` to export the guaranteed-QoS communication service to applications:

- **Channel endpoint state management:** `librtc` manages channel endpoint state for each channel established or accepted by an application. This per-channel state includes the source and destination `RTCOP` ports and IP addresses, the traffic specification and QoS requirements for this channel, the local Mach port for this channel, and a unique channel identifier allocated by `librtc`. This channel identifier is used by applications to uniquely identify established channels.

- **Signalling state management:** In response to `rtcRegisterPort`, i.e., when the application indicates its intent to receive signalling requests for channel establishment on a particular `RTCOP` port, `librtc` first allocates a Mach port to receive signalling requests on, and registers the Mach and `RTCOP` ports with the `CORDS` server, passing it send rights to the Mach port. If the specified `RTCOP` port is currently "busy," a free `RTCOP` port is selected by the `CORDS` server, associated with the Mach port, and returned to `librtc`. `librtc` then allocates a FIFO *incoming channel queue* and associates this queue and the specified signalling agent with the specified `RTCOP` port.

- **Receiver queue management:** Once a signalling agent accepts an incoming channel, information regarding the newly established channel (such as the channel identifies and source endpoint) is placed on the corresponding incoming channel queue until the application explicitly accepts an incoming channel via `rtcAcceptChannel`. Since this queue is also a "resource," incoming signalling requests are not accepted by the signalling agent if this queue fills up.

- **Server death cleanup:** During library initialization, `librtc` queries Mach name service to identify the Mach port registered by the `CORDS` server for receiving signalling requests. After obtaining the server Mach port, `librtc` requests the Mach kernel to notify it if this port, and hence the server, dies. Upon receiving such a notification, `librtc` reclaims all resources currently allocation to established channels, cleans up all state associated with these channels, marks the service as unusable, and informs the application subsequently. To allow the server to reclaim resources in case a sender dies, `librtc` allocates a Mach port representing the sender, and registers it with the `CORDS` server. This allows the `CORDS` server to request port death notifications on the sender Mach port. Note that for a receiving application, this operation is performed each time the application registers an `RTCOP` port to signal its intent to receive.

In the current implementation, the service library utilizes a single global lock to ensure mutual exclusion between multiple application threads and an internal library thread. The library thread receives and handles signalling requests from the `CORDS` server. All signalling agents specified by a receiving application run in the context of the library thread.

5.5.2 RTC API Anchor Implementation

`RTC API ANCHOR` is the top-level anchor protocol in the `CORDS` protocol stack, and serves as the interface between the rest of the protocol stack and `librtc`. The top half of `RTC API ANCHOR` provides entry points for server-side MIG interface stubs, while the bottom half of `RTC API ANCHOR` resembles an x-kernel protocol. `RTC API ANCHOR` performs the following functions in its capacity as the go-between interfacing the service protocol stack to the service library.

- **Consistent association maps:** `RTC API ANCHOR` maintains several mappings to keep track of the association between client Mach ports, `RTCOP` ports, and `RTCOP` sessions (x-kernel) for signalling as well as data transfer. These maps, implemented using the x-kernel map library, are used, for example, to demultiplex incoming signalling requests to the correct client Mach ports (at receivers), establish association between Mach ports, RTCOP ports and RTCOP sessions for data transfer (at senders), and identify all associations belonging to a client (at senders and receivers). The maps are updated each time a new channel is established or torn down, a client registers or unregisters an RTCOP port, a client awaits message arrival on a particular channel, or a client death notification is received.

- **Signalling request and reply handling:** RTC API ANCHOR relays signalling requests and replies between `librtc` and RTCOP. When a receiver registers an RTCOP port via `rtcRegisterPort`, RTC API ANCHOR first verifies with RTCOP that the specified port is free; if the port is not free, it requests RTCOP to allocate a free port. It then performs a passive open for this port with RTCOP, and creates the appropriate associations for later reference.

  On invocation of `rtcCreateChannel` by a sender, first allocates a Mach port for the new channel and creates a new *API thread* to wait in the server-side MIG interface stub for `rtcSendMessage` invocations on this channel. Per-channel API threads are necessary in order to support application blocking due to traffic specification violations, while allowing signalling and data transfers for other channels to proceed. As discussed in Section 5.7, per-channel API threads are also necessary to bound priority inversion within the CORDS server.

  RTC API ANCHOR then triggers end-to-end signalling via RTCOP and, if channel establishment is successful, creates sender-specific state for the new channel. It also associates a *message dequeue callback* function and a *link scheduler callback* function with the clip allocated by CLIPS for this channel. While the message dequeue callback function helps implement traffic enforcement, as described below, the link scheduler callback function will be utilized for intelligent buffer management in future versions of the implementation.

  Remote channel establishment requests destined for a particular RTCOP port are relayed by RTCOP to RTC API ANCHOR if a passive open had been performed earlier on this RTCOP port on behalf of a receiver. RTC API ANCHOR locates the corresponding Mach port allocated by the receiver for signalling requests, and performs an RPC to the receiver to complete signalling. If the receiver's signalling agent indicates acceptance, RTC API ANCHOR creates receiver-specific state for the new channel, and associates a *receive message callback* function with the clip allocated by CLIPS for this channel. It also requests port death notifications for the Mach port allocated by the receiver for the new endpoint; as explained below, this allows RTC API ANCHOR to perform client death cleanup. Signalling replies are generated by RTCOP as per the signalling agent's decision and returned to the sender.

Invocation of `rtcDestroyChannel` triggers a similar sequence of events, except that channel resources are reclaimed and the channel-specific state freed by RTC API, RTCOP, and CLIPS.

- **Buffer management and data transfer handling:** RTC API ANCHOR performs buffer management and application data movement during data transfer handling for outgoing as well as incoming messages. On invocation of `rtcSendMessage` by a sender, RTC API ANCHOR looks up the corresponding RTCOP session and obtains the clip associated with this session. It then creates an x-kernel message by allocating a buffer from the *path-specific* memory pool associated with the session, and copies the application message into this buffer. This x-kernel message is then handed to CLIPS for subsequent processing and transmission. If CLIPS is unable to accept (i.e., enqueue) the message due to queue overflow caused by a longer than specified burst, RTC API ANCHOR transparently *blocks* the channel's API thread, effectively blocking the application thread. The API thread resumes execution when woken by the message enqueue callback function. Instead of blocking the application, RTC API ANCHOR can also return an error indicating a traffic specification violation.

  On invocation of `rtcReceiveMessage` by a receiver, RTC API ANCHOR first identifies the RTCOP session and clip associated with the specified Mach port, and checks if a message is pending reception in the CLIPS message queue associated with this clip. If so, it obtains the x-kernel message from CLIPS, allocates a buffer from the path-specific memory pool associated with the session, and copies the data portion into this buffer. The newly allocated and filled buffer is then returned to the application. If no message is pending reception, and the receiver has made a blocking call, RTC API ANCHOR creates the appropriate state, updates its mappings to indicate that the receiver is awaiting message reception on the specified Mach port, and blocks the API thread. Upon subsequent receipt of a message on this channel, the receive message callback function invoked by the input communication handler signals the blocked API thread, which wakes up and completes the IPC call by copying out the received message. Note that an alternative would be to perform call completion and message copyout in the context of the receive message callback function. However, this would closely couple execution of the input communication handler with the memory copying bandwidth at the host and the size of the received message. By simply signaling the blocked API thread, the handler can continue processing arrived packets as fast as possible as per the QoS associated with the channel.

- **Client death cleanup:** As mentioned, `RTC API ANCHOR` maintains the appropriate state required to locate all associations specific to a particular client port. This state is initialized for each Mach port the client registers to receive signalling requests. If the client dies subsequently, port death notifications are issued by the kernel to the `CORDS` server for each Mach port for which such notifications were requested. `RTC API ANCHOR` then locates all the associations (i.e., mappings between Mach ports, `RTCOP` ports and x-kernel sessions for data transfer) corresponding to this Mach port, releases the resources allocated to these associations (e.g., it initiates channel tear down by closing the corresponding x-kernel session), and cleans up any remaining client state.

In addition to the interface to local `CORDS` resources for path-specific buffer management, `RTC API ANCHOR` utilizes the interfaces exported by `RTCOP` and `CLIPS`; these interfaces are described next in the context of the implementation of `RTCOP` and `CORDS`.

5.5.3 RTCOP and RTROUTER Implementation

`RTCOP` is implemented as a protocol object in the `CORDS` framework, and is responsible for a wide range of functions, including reliable end-to-end signalling, as outlined below.

- **End-to-end signalling:** `RTCOP` exports a set of x-kernel control operations that `RTC API ANCHOR` uses to trigger end-to-end signalling for channel establishment and tear down from a source to a destination host. It implements a state machine for reliable end-to-end signalling via source-based timers, set when a request is first transmitted, a certain fixed number of retries, and sequence numbers. Duplicate suppression is performed to ensure that resources are consistently allocated at a single node.

Request and reply processing in the state machine ensures that either all nodes along the route consider a particular channel established (if sufficient resources are available) or none of them do. Before initiating channel establishment signalling at any node (i.e., forwarding an establishment request to the next node), RTCOP first obtains the necessary local resources via admission control and resource reservation. At destination hosts, channel establishment is immediately terminated if no receiver is listening on the RTCOP port the signalling request is destined for.

- **Globally unique path and channel identifiers:** Path identifiers in CORDS must fit within 2 bytes. While these identifiers can be easily made unique inside a host, two hosts may unknowingly associate different channels with the same path identifier. This makes it impossible for intermediate and destination hosts to classify packets from different hosts (and channels) to unique paths. The limited length of the path identifier precludes use of the IP address to construct globally unique paths. Accordingly, RTCOP assumes a logical addressing scheme for hosts participating in our service. In this scheme, each real-time host (RTHost) is assigned a unique RTHost number. Path and channel identifiers are then constructed by concatenating the RTHost number of the source host with a locally unique channel counter. The mapping between RTHost addresses and IP addresses is maintained by RTROUTER, as explained later.

- **Channel state management and path allocation:** The state associated by RTCOP with each channel includes globally unique channel and path identifiers. During request and reply processing at each node, RTCOP obtains a new channel identifier and creates a new CORDS path for the new channel via the local CORDS resource interface. Associating a unique path with each channel provides an isolated pool from which buffers, queues and input threads may be allocated. Channel state is allocated from this path-specific memory pool. Unlike unique per-channel paths, all best-effort channels share the default path.

- **Admission control and resource reservation:** If these operations are successful, RTCOP invokes admission control via an interface to the admission control module, which implements D_order using the extensions outlined in Chapter 4. For a receiving host the extensions correspond to the interrupt mode packet input mechanism. D_order only allocates sufficient CPU and link bandwidth for the new channel. Once admitted by `D_order`, at a destination host `RTCOP` reserves path-specific resources such as an input pool of packet buffers, a packet queue, and a thread that shepherds packets arriving on the new channel's path through the protocol stack (i.e., the input communication handler).

`RTCOP` then obtains additional `CLIPS` resources such as message queues, reassembly buffers (at destination hosts) and communication handlers (at source hosts) by creating a clip for the new channel, with the appropriate priority and attributes, via the interface exported by `CLIPS`. After obtained all necessary communication resources, `RTCOP` "forwards" the signalling request to `RTC API ANCHOR` for final acceptance by the receiver, as described earlier. If the new channel is accepted, `RTCOP` computes available channel slack and relaxes the assigned local delay bound, updating the priority of the channel's clip in the process, before generating a successful signalling reply.

- **Port management:** `RTCOP` manages the ports used to uniquely identify real-time channel endpoints within a host. It exports an interface, implemented as x-kernel control operations, to obtain a free port arbitrarily chosen by `RTCOP`, reserve an application-specified port, and release a previously allocated port. This interface is used by `RTC API ANCHOR`, for example, during the handling of `rtcRegisterPort` to reserve the application-specified port, if any, or obtain a free port, before performing a passive open on `RTCOP`.

- **Callback function registration:** `RTCOP` exports control operations to allow `RTC API ANCHOR` at the source host to register the callback functions described earlier that will be used with clips associated with real-time channel endpoints.

- **Path deletion and resource reclamation:** `RTCOP` extends the `CORDS` path library to allow path deletion and resource reclamation on a path. This support, while limited to the user-space realization of `CORDS`, provides an interface to destroy device driver input pools consisting of input buffers and threads, as well as remove path state when channels are destroyed or signalling is unsuccessful. We have made modifications to the `ETHDRV` protocol to support extensions for resource reclamation on paths.

While implementing `RTCOP`, we resolved a minor limitation in the x-kernel uniform protocol interface: the lack of a natural way to pass traffic and QoS parameters obtained from `RTCOP` headers to `RTC API ANCHOR`. This is not a problem with protocols such as TCP in which a connection at a host is defined solely by the endpoint IP addresses and ports, and x-kernel participant lists are natural candidates for carrying this information between layers. Our solution is to use another x-kernel feature, *message attributes*, to treat the traffic and QoS parameters as attributes of an x-kernel message. This message is then demuxed to `RTC API ANCHOR`, which extracts the attributes and reconstructs the signalling request.

`RTROUTER`, which sits just below `RTCOP` currently performs the following functions: it (i) specifies the connectivity of real-time channel hosts with its pre-configured topology tables, (ii) provides a logical RTHost addressing mechanism for real-time channels, and (iii) handles forwarding of `RTCOP` packets between source and destination RTHosts.

5.5.4 CLIPS Implementation

As shown in Figure 5.6, `CLIPS` effectively spans the entire protocol stack, interacting with `RTC API ANCHOR`, `RTCOP`, and `ETHDRV` during signalling as well as data transfer. It provides support for associating a communication endpoint with a clip, and allocates CPU bandwidth, link bandwidth, and message queues and buffers to each clip. It realizes QoS-sensitive data handling via implementation of the following functions:

- **Clips management:** `CLIPS` exports an interface to create and destroy a clip, attach it to a particular endpoint, and to set its priority. This interface is used by `RTCOP` and `RTC API ANCHOR` during signalling. `CLIPS` distinguishes between incoming, outgoing, and bidirectional clips.

- **Message queue and reassembly buffer management:** `CLIPS` associates a ordered message queue with each clip created, sizing it according to a user-specified value. For an incoming clip, `CLIPS` also allocates a buffer to reassemble messages destined for the endpoint corresponds to this clip. Fully reassembled messages are placed on the incoming message queue for subsequent retrieval by `RTC API ANCHOR`.

- **Fragmentation and reassembly:** `CLIPS` fragments (reassembles) outgoing (incoming) messages whose size exceeds the maximum amount of user data that can be carried in an MTU-sized packet. The fragmentation is performed by the output communication handler, while the reassembly is performed by the input shepherd thread. CLIPS associates a small header with each fragment for correct reassembly. After creating each fragment, CLIPS invokes a user-defined function to transport the fragment; in our implementation, this function is an entry point into the RTCOP layer.

- **CPU bandwidth allocation:** CLIPS uses a user-level CPU scheduler to schedule communication handlers. The deadlines of these handlers is derived from the period and relative deadline (set by RTCOP at clip creation time) associated with the corresponding clip. The scheduler utilizes x-kernel semaphores to block handlers when they yield the CPU and to wake them up subsequently.

- **Link bandwidth allocation:** The CLIPS link scheduler is realized as a separate thread in the ETHDRV protocol. It runs at the highest x-kernel priority, serving packets first from the real-time heap and then from the best-effort heap. Note that the link scheduler does not run under control of the CLIPS CPU scheduler, and hence does not need to hold the master x-kernel lock to run; it immediately preempts any executing handler. This effectively realizes a modified version of the O2 option in Chapter 3.

5.6 System Profiling and Parameterization

We have performed a detailed profiling of our service implementation in order to parameterize it in terms of system costs and overheads. This parameterization permits the use of the admission control extensions outlined in Chapter 4 correctly. More importantly, it helps identify overheads *not* accounted for by our admission control extensions, and expose architectural deficiencies that could potentially result in QoS-insensitive data handling within the communication subsystem.

Our profiling methodology is to conduct all experiments and measurements on two hosts in the testbed (see Figure 5.4), connected by an isolated Ethernet segment. This isolation from the rest of the world is readily achieved by configuring the Ethernet switch appropriately. The service library librtc and the CORDS protocol stack are instrumented according to the desired measurements. Only one set of measurements are performed at a time in order to minimize the perturbation induced by the profiling code.

Given our primary focus on run-time resource management, we concentrate on the data

Anchor Parameter	Anchor Routine	Message Size			
		1 byte	1k bytes	10k bytes	30k bytes
$\mathcal{C}_a$	Send message	301	331	606	1191
$\mathcal{C}_a^{r,w}$	Receive message (waiting)	335	350	1097	2653
$\mathcal{C}_a^{r,nw}$	Receive message (none waiting)	54	54	53	54
$\mathcal{C}_a^{r,cb}$	Receive message callback	95	92	96	96

Table 5.4: Data transfer overheads in `RTC API` anchor (in μs).

transfer performance of our platform and prototype implementation, for both incoming and outgoing data. For all the results reported here, a single real-time channel is created from a sending client on one host to a receiving client on another host. We first present profiling results for `RTC API ANCHOR`, then present the results of profiling the protocol stack layers, and finally present profiling results for link (i.e., network) input and output.

5.6.1 Profiling the RTC API Anchor

In this set of measurements, we profile routines in `RTC API ANCHOR` that handle send and receive requests from applications. Note that these routines are the entry points into `RTC API ANCHOR` for the server-side MIG stubs. The time spent in these routines corresponds to the time spent by API threads in `RTC API ANCHOR`. We also profile the receive message callback function mentioned in Section 5.5.2. Time spent in this function corresponds to the time an input communication handler thread spends in `RTC API ANCHOR`.

Table 5.4 lists the data transfer overheads of these anchor routines for messages of size 1 byte, 1K bytes, 10K bytes and 30K bytes. $\mathcal{C}_a$ denotes the overhead incurred by the `send message` routine, while $\mathcal{C}_a^{r,w}$ denotes the overhead of the `receive message` routine when a message is already waiting at the time the application invokes the `rtcRecvMessage` library call. Similarly, $\mathcal{C}_a^{r,nw}$ denotes the overhead of the `receive message` routine when there is no message waiting and the application invokes a blocking receive call. Note that $\mathcal{C}_a^{r,nw}$ only includes the time from entry into `RTC API ANCHOR` till the time that the application thread is blocked waiting for a message to arrive. $\mathcal{C}_a^{r,cb}$ denotes the overhead of the receive message callback function invoked when a message arrives to an empty `CLIPS` queue. All measurements reported are in microseconds (μs).

For each routine, the measurements for a message size of 1 byte largely correspond to the fixed overhead introduced by the routine. Consider the overhead incurred in the

Service Library Routine	Message Size			
	1 byte	1k bytes	10k bytes	30k bytes
rtcSendMessage	1170	1210	1480	2070
rtcRecvMessage (waiting)	870	894	1660	3210

Table 5.5: Application-level send and receive latencies (in μs).

$\mathcal{C}_a$ routine at the source host, averaged over 1000 messages. Over and above the fixed overhead, the time spent in this routine increases with message size. This is because, as mentioned in Section 5.5, `RTC API ANCHOR` copies application data into path-specific message buffers in order to preserve application data integrity in the worst case. For our platform, lmbench [115] reports a memory copy bandwidth of $\approx$ 40 MB/second; the actual copy bandwidth would be somewhat lower due to the overheads imposed by the x-kernel copy routine. In any case, the increase in $\mathcal{C}_a$ can be largely attributed to the time to copy in application data.

An anomalous trend is observed for $\mathcal{C}_a^{r,w}$. If the message is already waiting, it is dequeued and copied into path-specific message buffers before returning it to the application. Once again, the overhead increases with message size. For message sizes of 10K bytes and 30K bytes, the overhead (averaged only over 500 and 300 messages, respectively, due to overflow of path buffer space) cannot be attributed only to the cost of copying the message. We believe this is partially due to the overhead introduced by the memory allocation primitive provided in `CORDS`.

Completely consistent behavior is observed for $\mathcal{C}_a^{r,nw}$, averaged over 500 messages. As expected, $\mathcal{C}_a^{r,nw}$ is independent of message size, since `RTC API ANCHOR` effectively blocks the application if no message is waiting, For the overhead measurements (averaged over 500 messages) are independent of message size. This is completely consistent since the receive message callback function simply signals the blocked API thread, if there is one. Note that the reported measurements correspond to the case when the application is already blocked waiting for a message, which is the worst-case scenario for for $\mathcal{C}_a^{r,cb}$. If the application is not waiting, the receive message callback function performs minimal processing and returns.

Application-level latencies

To validate some of these observed trends, we also measure application-level latencies for the `rtcSendMessage` and `rtcRecvMessage` routines. These latencies include the cost of an IPC call across the MIG interface and executing the corresponding anchor routine discussed above. As shown in Table 5.5, the application-level latency for `rtcSendMessage` (averaged over 1000 messages) follows a pattern similar to that of $\mathcal{C}_a$, except that the fixed overhead is significantly higher. This extra overhead is the cost of a send IPC across the MIG stubs to the `CORDS` server. Comparing `rtcSendMessage` latency to $\mathcal{C}_a$, the average extra overhead is $\approx 875\ \mu s$.

From Table 5.5, a similar observation can be made for the `rtcRecvMessage` latency, averaged over 500 messages, which follows a pattern similar to that of $\mathcal{C}_a^{r,w}$. We observe that the latency for `rtcRecvMessage` reveals an average extra overhead of $\approx 563\ \mu s$. This extra overhead is the cost of a receive IPC from the application to the `CORDS` server across the MIG interface.

We note that while the `RTC API ANCHOR` overheads are relatively high, these measurements are for an unoptimized implementation and can be improved substantially with careful performance optimizations. With appropriate buffer management and API buffering semantics [25, 130] it may even be possible to completely eliminate the copying of data within `RTC API ANCHOR`. However, more immediately we are concerned with ensuring that the overheads incurred in `RTC API ANCHOR` do not result in QoS-insensitive handling of data. We address this concern with appropriate admission control and architectural extensions in Section 5.7. While the application-level latencies are also high, this is primarily due to significant MIG IPC overhead, which we may be able to reduce with a colocated in-kernel `CORDS` server. Moreover, in a QoS-sensitive operating system, this IPC overhead can be accounted for in the application's execution.

5.6.2 Profiling the Protocol Stack

We have also profiled the other layers of the service protocol stack, for both the send and receive data paths. Since `CLIPS` is a separate component that performs fragmentation (on the source host) and reassembly (on the destination host), we profile it separately. For the other layers (`RTCOP`, `RTROUTER`, `IP`, `ETH`, `ETHDRV`) we only measure the *aggregate* overhead.

Data Path	Protocol Stack Layer	Packets in Message	
		First	Other
Send	CLIPS message dequeue and fragmentation	53	30
	RTCOP + RTROUTER + IP + ETH + ETHDRV	128	47
		Other	Last
Receive	CLIPS reassembly and message enqueue	9	32
	CLIPS+ RTCOP + RTROUTER + IP + ETH + ETHDRV	260	260

Table 5.6: Protocol stack latencies for send and receive paths (in μs).

Table 5.6 lists the profiling results for all layers other than data link-level transmission and reception, which is profiled separately in Section 5.6.3 below.

For the send path, we distinguish the measured overhead for the first packet of a message from that for the other packets. The dequeue and fragmentation costs in CLIPS are averaged over 200 messages. The aggregate protocol processing costs for the rest of the protocol stack are averaged over 100 messages. Note that the aggregate protocol processing cost for the first packet is significantly higher than that for the other packets. Recall that the output communication handler fragments packets and shepherds them down the protocol stack in a single loop. The difference in overhead between the first and other packets can be partly attributed to cache effects, which have been shown to affect protocol stack execution latency significantly [16, 133].

For the receive path, on the other hand, we distinguish between the last packet of a message and the other packets. This is because, while fragmentation cost is effectively incurred on a per-packet basis as the message is processed, the entire reassembly cost is only incurred during the processing of the last packet of a message. The receive path overheads listed in Table 5.6, averaged over 1000 messages, reveal some surprising results.

Unlike the send path, all packets in the message incur the same aggregate protocol processing cost. Moreover, since reassembly proceeds in a tighter loop than the fragmentation loop in the send path, cache effects are even more significant, as evident from the significantly lower reassembly cost compared to the fragmentation cost. Note that, some CLIPS processing is performed each time a packet is put in the reassembly buffer, and hence is included in the aggregate per-packet protocol processing latency.

ETHDRV Layer	Packet Size		
	1 byte	500 bytes	1416 bytes
Packet transmission by link scheduler	673	1510	1775

Table 5.7: Link scheduler packet transmission latencies (in μs).

5.6.3 Profiling Link Input/Output

For a full parameterization of the communication subsystem, we also profile packet transmissions by the link scheduler at the sending host, and packet reception by the CORDS server at the receiving host.

Table 5.7 lists the packet transmission latencies measured in the link scheduler as a function of the packet size, averaged over 1000 packets for each packet size. Once again, the latency measurement for 1-byte packets roughly corresponds to the fixed overhead of performing synchronous user-level transmission via the Mach device control port. This overhead includes the cost of a user-kernel context switch, invocation of the device driver transmit routine, handling of the transmission-complete interrupt, delivering of an I/O-complete notification to the Mach device control port, which in turn wakes up the waiting link scheduler, and another context switch to resume execution of the link scheduler thread. Since the measured latency includes the time to transmit the entire packet on the wire, larger packets incur higher latencies.

Table 5.8 lists the overhead incurred by the CORDS device input thread to receive an incoming packet from the device control port, classify it to determine its path, locate the corresponding buffer pools, enqueue the packet in the path input packet queue, and signal the CLIPS CPU scheduler to wake up the input communication handler associated with this path. Note that the input overhead for the first packet is significantly higher than that for the subsequent packets. We have verified that this is primarily due to the high overhead to signal (i.e., wakeup) the handler.

This CORDS device input thread overhead does not include the in-kernel cost of fielding the packet arrival interrupt, inputing the arrived packet, applying the generic net filter and dispatching the packet to the appropriate device control port. Preliminary in-kernel measurements reveal this overhead to be $\approx 650\ \mu s$ for an Ethernet MTU-sized packet.

ETHDRV Layer	Packets in Message	
	First	Other
Packet input by `CORDS` device input thread	120	20

Table 5.8: `CORDS` device input thread overhead (in μs).

5.7 Accounting for API Overheads and Threads

The profiling results for `RTC API ANCHOR` indicate that the overheads induced by `RTC API ANCHOR` can be significant and must be accounted for in the admission control procedure. However, simply accounting for these overheads does not suffice for a number of reasons.

- Due to the copying performed by `RTC API ANCHOR`, the constituent anchor overhead is directly proportional to the size of messages sent and received by applications. That is, worst-case bounds on this overhead cannot be determined *a priori*.
- The API threads obtain access to the CPU in FIFO order. This is because their execution is controlled by the CThreads library and the Mach kernel scheduler.
- Further, because all threads executing in the `CORDS` x-kernel framework must hold the x-kernel master lock, these API threads effectively run to completion.

It follows, therefore, that substantial admission control and architectural enhancements are needed to ensure that API threads execute in `RTC API ANCHOR` in a QoS-sensitive fashion.

5.7.1 Admission Control Enhancements

We first consider the necessary admission control enhancements to account for the anchor parameters profiled in Section 5.6.

Referring to the system parameters defined in Chapter 4, note that the message service time of the output communication handler remains unaffected by the anchor overheads. However, because the API threads send and receive messages on a channel-specific basis, the channel's total message service time increases by $\mathcal{C}_a$, plus an extra context switch. That is,

$$\mathcal{T}_s^{new} = \mathcal{T}_s^{old} + \mathcal{C}_a + \mathcal{C}_{csp},$$

where $\mathcal{T}_s^{old}$ is the message service time for a sending host as calculated in Chapter 4, and $\mathcal{C}_{csp}$ is the context switch and cache miss penalty as before. The message wait time increases by the same amount as well, since a output handler may have to wait for an API thread to finish executing the send routine in `RTC API ANCHOR`. Accordingly,

$$\mathcal{T}_w^{new} = \max(\mathcal{T}_w^{old}, \mathcal{C}_a + \mathcal{C}_{csp}),$$

where $\mathcal{T}_w^{old}$ is the message wait time for a sending host as calculated in Chapter 4.

Similar admission control enhancements must be derived for receiving hosts as well. In this case, however, we must consider the combined effect of $\mathcal{C}_a^{r,w}$, $\mathcal{C}_a^{r,nw}$and $\mathcal{C}_a^{r,cb}$. An input communication handler (i.e., the input shepherd thread) sees an increase in message service time since, in the worst case, it has to execute the receive message callback function after reassembling and enqueueing an incoming message. However, the total channel message service time requirements increase by the execution overhead of the API threads as well. Hence,

$$\mathcal{T}_s^{r,new} = \mathcal{T}_s^{r,old} + (\mathcal{C}_a^{r,w} + \mathcal{C}_a^{r,nw} + \mathcal{C}_a^{r,cb}) + \mathcal{C}_{csp}^r,$$

where $\mathcal{T}_s^{r,old}$ is the message service time for a receiving host as calculated in Chapter 4. The total channel message wait time is now

$$\mathcal{T}_w^{r,new} = \max(\mathcal{T}_w^{r,old} + \mathcal{C}_a^{r,cb}, \mathcal{C}_a^{r,w} + \mathcal{C}_a^{r,nw}) + \mathcal{C}_{csp}^r,$$

where $\mathcal{T}_w^{r,old}$ is the message wait time for a receiving host as calculated in Chapter 4. This is because, in the worst case, an incoming handler may have to wait for an API thread or another incoming handler to complete execution.

As mentioned, these admission control enhancements alone do not suffice in ensuring QoS-sensitive data handling within the communication subsystem. While the total message service times account for the increased CPU requirements for a channel, the channel message wait times are potentially unbounded since both $\mathcal{C}_a$and $\mathcal{C}_a^{r,w}$ increase with message size. Moreover, there is the possibility of API threads executing on behalf of a particular channel interfering with the execution of that channel's communication handler. Not only is this because API threads execute non-preemptively in FIFO order, but also because API threads could repeatedly execute the `RTC API ANCHOR` routines under a persistent burst of messages on a channel, as long as there is space on the `CLIPS` message queue. Thus, substantial architectural enhancements are also required to ensure QoS-sensitive data handling.

5.7.2 Architectural Enhancements

In the ideal case, on invocation of a service routine by an application would result in the API threads being scheduled to run on the CPU in the order of priority, with this priority derived from the "priority" of the specified channel. However, that would require appropriate support from the Mach IPC and scheduling subsystems in the kernel. In the absence of such support, we cannot control the FIFO scheduling of API threads on the CPU immediately after wake up in the server-side MIG interface stubs. However, we can exercise control over the subsequent execution of API threads to ensure that they do not run to completion non-preemptively and execute in `RTC API ANCHOR` in a QoS-sensitive fashion. We achieve this via architectural enhancements to `RTC API ANCHOR` and `CLIPS`, as described below.

We place four requirements on the execution of API threads in relation to `CLIPS` communication handlers.

1. While they are executing in `RTC API ANCHOR`, they must run under control of the `CLIPS` CPU scheduler,
2. They must not prevent the timely execution of any communication handler or API thread,
3. They must not be starved of CPU capacity since that would stall message generation and consumption on a channel, and
4. They must be allowed to consume excess CPU capacity while being fair to best-effort traffic.

To realize these requirements in practice, we make the following key observation. While admission control (i.e., `D_order`) allocates CPU capacity to this channel according to the total message service time calculated above, run-time resource management can be performed such that a portion of that capacity is used by the API thread and the rest by the communication handler.

Recall that `CLIPS` only associates a single communication handler with each clip. We extend `CLIPS` to also associate an API thread with the same clip. An API thread explicitly *checks in* with `CLIPS` on entry into any `RTC API ANCHOR` routine, and explicitly *checks out* just before exiting the routine. `CLIPS` utilizes these check-in and check-out calls to schedule the execution of the API threads. Any API thread that checks in on a given clip inherits the priority/deadline associated with the handler, with the following restrictions. Communication handlers always have execution priority over the corresponding API threads. An API thread that checks in when the handler is blocked for message or packet arrivals, is assigned to the same execution *class* as the handler and runs at the handler's deadline.

When the API thread checks out (after executing an `RTC API ANCHOR` routine), `CLIPS` lowers its execution class relative to *all* communication handlers, including those handling best-effort traffic. On the next check in, the API thread executes in this lower class, competing with other API threads in this class in the order of the logical arrival time of the message it is transferring. During check in and check out, if the corresponding handler is woken up (say if the API thread has enqueued a message for transmission), it is scheduled for execution according to the deadline of the message to be processed next.

Upon expiry of the handler's budget, which also marks the end of the handler's execution for the current invocation period, the execution class for an API thread is restored to that of the handler. On a subsequent check in, since the handler is blocked, an API thread executes in this class at the deadline of the handler. The above scenario then repeats.

These architectural enhancements to `RTC API ANCHOR` and `CLIPS` ensure that different incarnations of API threads are scheduled for execution in a QoS-sensitive fashion relative to other API threads and communication handlers. However, once scheduled the API threads still run to completion in a non-preemptive fashion. To prevent this, the `RTC API ANCHOR` routines have been modified to copy application data in well-defined *chunks*, and cooperatively yield the CPU after processing each chunk. The chunk size corresponds to the amount of data copied before yielding the CPU; the execution time of a chunk can then be ascertained from the chunk size and the memory copy bandwidth of the communication subsystem.

With these changes, the preemption mechanism employed for an API thread closely resembles cooperative preemption of a communication handler. Accordingly, both the message service time and message wait time enhancements can now be stated differently. For a sending host,

$$\mathcal{T}_s^{new} = \mathcal{T}_s^{old} + \mathcal{C}_a + \lceil \frac{\mathcal{C}_a}{\mathcal{C}_a^{chunk}} \rceil \mathcal{C}_{csp},$$

where $\mathcal{C}_a^{chunk}$ is the time to copy a chunk of data. Similarly,

$$\mathcal{T}_w^{new} = \max(\mathcal{T}_w^{old}, \mathcal{C}_a^{chunk} + \mathcal{C}_{csp}).$$

For a receiving host,

$$\mathcal{T}_s^{r,new} = \mathcal{T}_s^{r,old} + (\mathcal{C}_a^{r,w} + \mathcal{C}_a^{r,nw} + \mathcal{C}_a^{r,cb}) + \lceil \frac{\mathcal{C}_a^{r,w} + \mathcal{C}_a^{r,nw}}{\mathcal{C}_a^{chunk}} \rceil \mathcal{C}_{csp}^{r},$$

and the channel message wait time is

$$\mathcal{T}_w^{r,new} = \max(\mathcal{T}_w^{r,old} + \mathcal{C}_a^{r,cb}, \mathcal{C}_a^{r,chunk}) + \mathcal{C}_{csp}^{r}.$$

5.8 Experimental Evaluation

In this section we present results from several experiments conducted to evaluate the efficacy of our prototype implementation. The experiments demonstrate two key aspects of the QoS support provided: traffic enforcement (i.e., policing and shaping) on a single channel, and traffic isolation between mutiple channels. Traffic isolation in turn has two components, namely, the isolation between different real-time channels and the isolation between real-time and best-effort traffic.

The experiments are conducted between two hosts communicating on a private segment through the Ethernet switch shown in Figure 5.4. To avoid interference from the Unix server, all ARP traffic between the two hosts is suppressed and the `CORDS` server is configured to accept all incoming network traffic. This allows us to limit the background CPU load on each host and accurately control network traffic between them. For each experiment reported below, the corresponding channels are first created between Mach client tasks running at the two hosts. Message traffic is then generated by threads running within the Mach client task at the source host and consumed by threads running within the Mach client task at the destination host. Our metric for evaluation is the per-channel application-level throughput delivered to the receiving Mach task at the destination host.

5.8.1 Traffic enforcement

For this experiment, a real-time channel with traffic specification $B_{max} = 10$, $M_{max} =$ 40 KB, $R_{max} = 5$ messages/second, and deadline of 200 ms, is established. This channel then has a specified rate of 200 KB/s. The actual offered load on the channel is varied

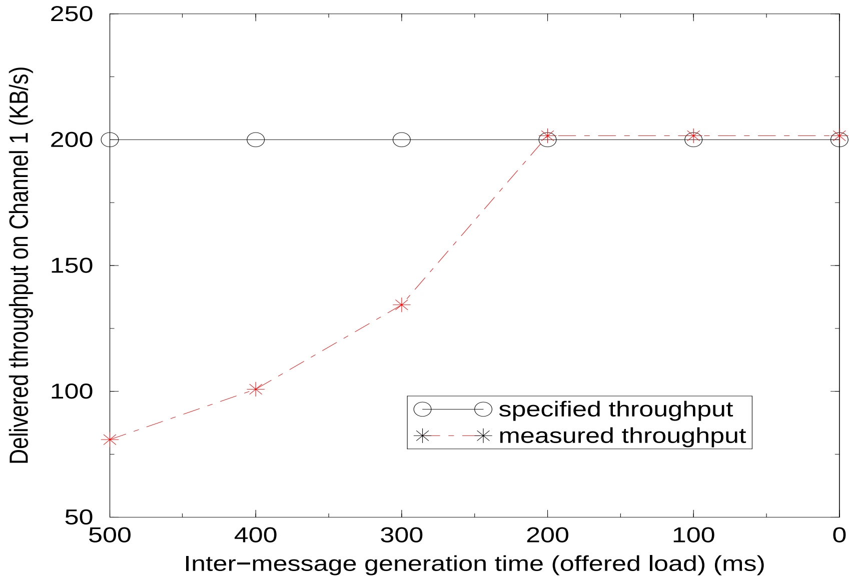


Figure 5.7: Traffic enforcement on a single real-time channel.

by changing the interval between generation of successive messages, ranging from 500 ms to 0 ms (i.e., continuous traffic generation). Figure 5.7 illustrates the efficacy with which traffic is enforced on a single real-time channel by measuring the delivered throughput as a function of the offered real-time load.

As shown, the delivered throughput increases linearly with the offered load until the offered load equals the specified channel rate. For example, at an offered load of 100 KB/s (corresponding to a message generation interval of 400 ms), the delivered throughput is 100 KB/s. Similarly, at an offered load of 200 KB/s (message generation interval of 200 ms), the delivered throughput is 200 KB/s. For offered loads beyond the specified channel rate, however, the delivered throughput equals the specified channel rate. This continues to be the case even under continuous message generation (message generation interval of 0 ms). These measurements show that the traffic enforcement mechanisms provided effectively prevent a real-time channel from violating its specified rate. In the prototype implementation, this is achieved by blocking the corresponding channel thread from generating messages at the source host until additional space is made available in the channel message queues.

5.8.2 Traffic isolation

In addition to proper traffic enforcement, we demonstrate that our prototype implementation also ensures isolation between different QoS and best-effort connections, thus verifying conformance with the real-time channel paradigm. We first consider traffic isolation between multiple real-time channels under violation of traffic specification by a real-time channel. We then consider traffic isolation between real-time and best-effort traffic under increasing best-effort load.

Multiple real-time channels

For this experiment two real-time channels are established between the hosts, with one representing a high-rate channel (channel 1) and the other representing a low-rate channel (channel 2). The high-rate channel has the same traffic and deadline specification as before (i.e., $B_{max} = 10$, $M_{max} = 40$ KB, $R_{max} = 5$ messages/second, deadline = 200 ms) for a specified channel rate of 200 KB/s. The low-rate channel has a traffic specification of $B_{max} = 10$, $M_{max} = 15$ KB, $R_{max} = 2$ messages/second, and deadline of 100 ms, for a specified channel rate of 30 KB/s. While message generation (and hence the offered load) on channel 1 is continuous, message generation on channel 2 is controlled in order to vary the offered load as in the previous experiment.

Figure 5.8 shows the delivered throughput on channels 1 and 2 as a function of the offered load on channel 2. Once again, the delivered throughput on channel 2 increases linearly with the offered load until the offered load equals the specified channel rate (200 KB/s). Subsequent increase in offered load has no effect on the delivered throughput which stays constant at the specified channel rate. The delivered throughput on channel 1, on the other hand, remains constant at approximately 30 KB/s (same as its specified channel rate) regardless of the offered load on channel 2. That is, traffic violations on one channel (even continuous message generation) have no effect on the delivered QoS for another channel.

Real-time and best-effort traffic

For this experiment we create an additional "best-effort channel" in addition to two real-time channels. As before, one real-time channel (channel 1) represents a high-rate channel and has traffic specification of $B_{max} = 10$, $M_{max} = 20$ KB, $R_{max} = 10$ messages/second and

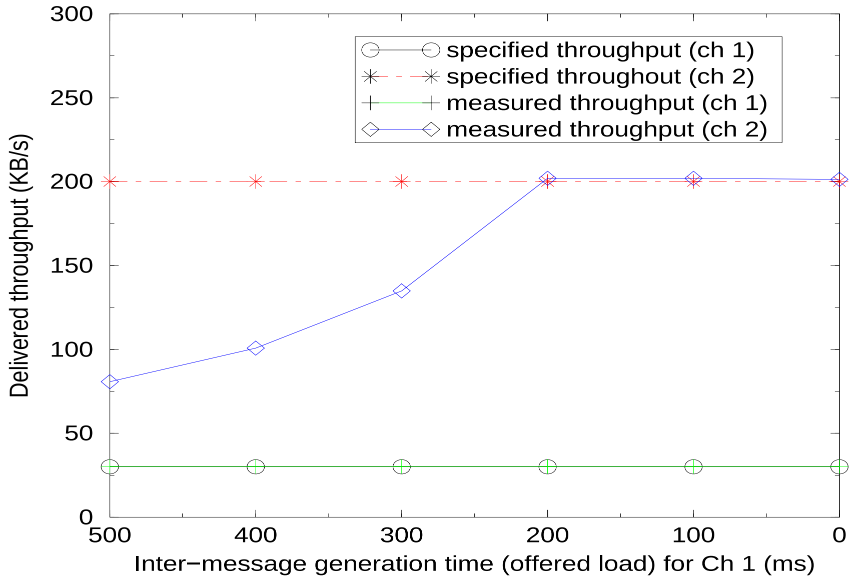


Figure 5.8: Traffic isolation between two real-time channels.

deadline of 200 ms, for a specified channel rate of 200 KB/s. The other real-time channel (channel 2) represents a low-rate channel with a traffic specification of B_{max} = 10, M_{max} = 5 KB, R_{max} = 5 messages/second and deadline of 100 ms, for a specified channel rate of 25 KB/s. Message generation on channels 1 and 2 is continuous, i.e., with a message generation interval of 0 ms. The offered load on the best-effort channel (channel 3) is varied from 50 KB/s to 350 KB/s by controlling the message generation interval as before.

Figure 5.9 plots the delivered throughput on each channel as a function of the offered best-effort load. A number of observations can be made from these measurements. First, the delivered throughput on channels 1 and 2 are roughly independent of the offered best-effort load. That is, real-time traffic is effectively isolated from best-effort traffic, except under very high best-effort loads as explained below. Second, best-effort traffic is able to utilize any excess capacity not consumed by real-time traffic, as is evidenced by the roughly linear increase in delivered throughput on channel 3 as a function of the offered best-effort load. Once the system reaches saturation (beyond an offered load of approximately 250 KB/s), however, best-effort throughput declines sharply due to buffer overflows and the resulting packet loss at the receiver.

Under very high best-effort loads, the delivered throughput on channel 1 declines slightly.

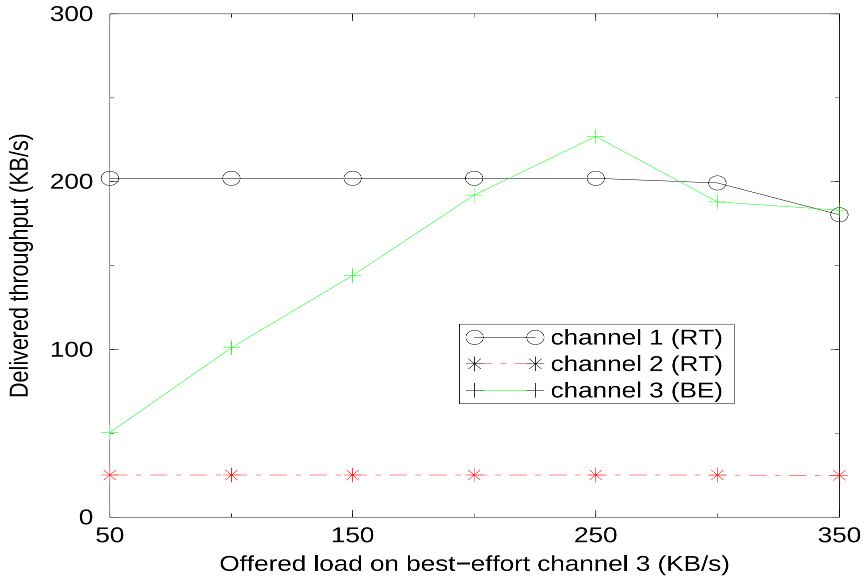


Figure 5.9: Traffic isolation between real-time and best-effort traffic.

We believe that this is due to the overheads of receiving and discarding best-effort packets, which have not been accounted for in the admission control procedure. These overheads tend to impact the delivered throughput on high-rate channels more than low-rate channels, as evidenced by the constant throughput delivered to channel 2 even under very high best-effort load.

Utilization of unused real-time capacity

While the load offered by real-time channels in the previous experiments is persistent, i.e., always greater than the reserved capacity, this experiment focuses on utilization of any capacity not utilized by a real-time channel. It is desirable that this unused capacity be utilized by best-effort traffic, as per our goal of fairness. That is, another real-time channel must not be allowed to consume this excess capacity at the expense of best-effort traffic. Two real-time channels and a best-effort channel are created as before; however, while the offered load on channel 2 is continuous, channel 1 only offers a load of 100 KB/s even though it is allocated a capacity of 200 KB/s. This is realized by generating 20 KB messages at half the specified rate of 10 messages/second (case 1).

Figure 5.10 plots the delivered throughput on all the channels as a function of the offered

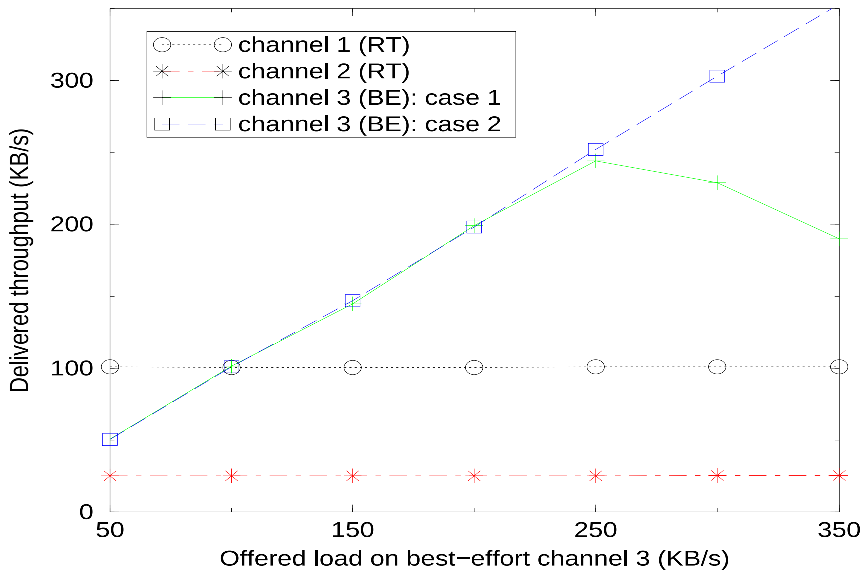


Figure 5.10: Traffic isolation and unused capacity utilization.

load on the best-effort channel (channel 3). Compared to Figure 5.9, channel 1 receives a constant 100 KB/s throughput independent of the offered best-effort load. Similarly, channel 2 only receives its allocated capacity of 25 KB/s. Channel 3, however, receives higher throughput (case 1) with the delivered throughput increasing linearly with the offered load until an offered load of 250 KB/s. Beyond this load, the delivered best-effort throughput falls as before, but continues to be higher than that obtained in Figure 5.9.

Surprisingly, though, best-effort traffic is unable to fully utilize unused capacity on channel 1. We suspect that this effect is primarily due to packet losses caused by buffer overflow at the receiver, either in the adapter or in the Mach device port queue used by the `CORDS` server to receive incoming packets. To validate this, we ran additional experiments in which channel 1 offers a load of 100 KB/s by generating 10 KB messages at a rate of 10 messages/second (case 2), the results of which are also plotted in Figure 5.10. As can be seen, the delivered best-effort throughput in this case continues to increase linearly beyond 250 KB/s and shows no decline even for a best-effort load of 350 KB/s. These results suggest that best-effort traffic is able to fully utilize unused capacity when real-time traffic is less bursty (i.e., has fewer packets in each message).

Discussion

Note that with the user-level CORDS server configuration, the receiving task is able to receive packets at an aggregate rate of 450-500 KB/s (depending on the number of packets in a message), even though the sender can send at a maximum rate of approximately 750 KB/s. This discrepancy is most likely due to CPU contention between the receiving task and the CORDS server and the resulting context switching overheads, and the high cost of IPC across the client and server. Another important reason could be a seemingly unnecessary copy performed by the lowest layer of the CORDS protocol stack every time packets from multiple paths (i.e., channel) arrive in an interleaved fashion. Since such a scenario occurs very frequently with multiple channels and under high traffic load, it is likely that this extra copy is slowing down the receiver significantly; this extra copy can only be eliminated by redesigning path buffer management in the CORDS framework. More importantly, none of these effects are accounted for in the admission control procedure, and must be addressed when the communication subsystem is integrated more closely within the host operating system. We expect to see significant improvements in the base performance for an in-kernel realization of our prototype implementation.

5.9 Summary

In this chapter we described our experiences with the design, implementation, and evaluation of a guaranteed-QoS communication service based on the architecture and extensions developed in Chapters 3 and 4. While our implementation was specifically geared towards the OSF MK 7.2 microkernel operating system, the architectural and implementation approach adopted is applicable to other microkernel as well as traditional monolithic operating systems. Our experimental results demonstrate that the architectural features provided in the service are effective in providing QoS guarantees to individual real-time channels while maintaining fairness to best-effort traffic. These results also reveal several deficiencies of a server-based implementation especially at the receiving host, that could be largely resolved by collocating the server in the kernel.

The guaranteed-QoS service described in this chapter is being utilized in the ARMADA project [9], a collaborative effort between the Real-Time Computing Laboratory at the University of Michigan and the Honeywell Technology Center. The project aims to develop

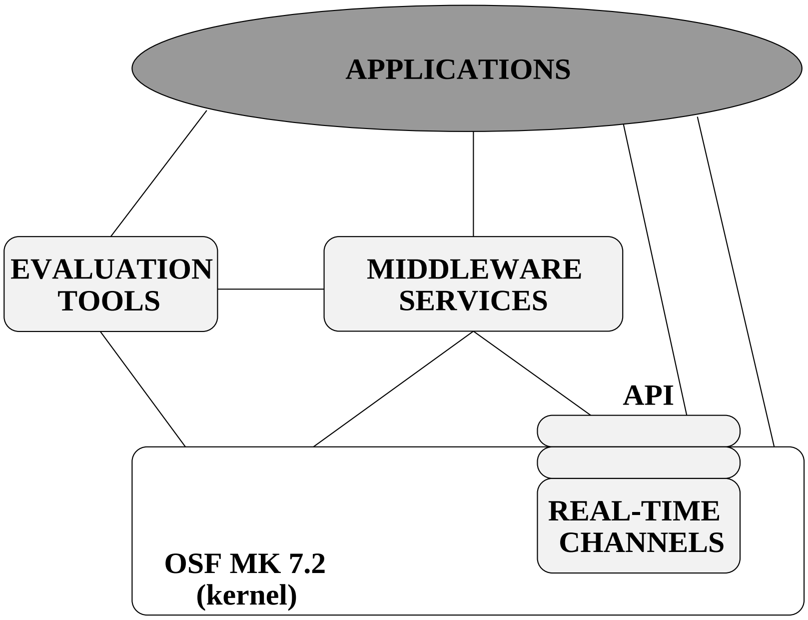


Figure 5.11: ARMADA middleware and real-time communication services.

and demonstrate an integrated set of techniques and software tools necessary to realize embedded fault-tolerant and real-time applications on distributed, evolving computing platforms. Figure 5.11 summarizes the structuring of the ARMADA environment and illustrates the three complementary thrust areas, namely, real-time communication service using real-time channels (described in this chapter), middleware services for embedded applications, and dependability evaluation and validation tools.

The overall research approach lies in the development of a suite of composable and reusable middleware services supporting a wide variety of embedded real-time and fault-tolerant applications. Accordingly, the service described in this chapter would be utilized by one or more middleware services such as real-time caching and real-time primary-backup replication [118]. In the context of the ARMADA project, numerous research avenues are being explored towards extending the guaranteed-QoS communication service to other models of real-time communication, including QoS negotiation and adaptation.

Acknowledgements

Several individuals have contributed to the realization of the guaranteed-QoS communication service described in this chapter. We gratefully acknowledge the contributions of the following individuals: Anees Shaikh for implementing `RTCOP` and `RTROUTER`, Tarek Abdelzaher for implementing `CLIPS`, and Zhiqun Wang for performing the profiling experiments. Special thanks to Anees and Tarek for incorporating numerous modifications/improvements suggested from time to time, especially the `CLIPS` enhancements proposed in Section 5.7. Special thanks also to Zhiqun for patiently repeating several profiling measurements to verify correctness and/or identify consistent trends.

CHAPTER 6

SELF-PARAMETERIZING PROTOCOL STACKS

6.1 Introduction

As is evident from the preceding chapters, realizing a QoS-sensitive communication subsystem requires that the host communication subsystem be parameterized accurately to create an abstraction of the underlying communication subsystem. This abstraction of the communication subsystem in terms of a set of system parameters greatly facilitates estimation of the communication resource requirements for a connection requiring QoS guarantees. Such an abstraction also serves to define the exact dependency between the admission control procedure and the performance characteristics of the underlying hardware and software components.

However, there are two aspects that make the realization of such an abstraction, and hence the design of a QoS-sensitive communication subsystem, extremely difficult. First, significant efforts are needed to understand and profile various communication subsystem (and operating system) overheads on sending and receiving hosts, and incorporating them in the admission control procedure. While necessary for provision of deterministic QoS guarantees, such an exercise is desirable for other classes of services as well, although not to the same level of accuracy or detail. The two prototype implementations we developed gave us first-hand experience with the difficulties of accurately profiling and parameterizing the communication subsystem manually.

Second, provision of QoS guarantees is highly platform-specific (i.e., depends on the CPU and network capacities, as well as the operating system, of a platform), especially for deterministic guarantees. Although system parameterization makes the admission control *procedure* more accurate and portable, full system parameterization must still be performed for each host and operating system platform that deploys a QoS-sensitive communication subsystem. The architectural framework and methodology adopted for designing QoS-sensitive communication software should be applicable to a variety of host platforms and networking technologies. This is necessary in order to retarget the admission control procedure and run-time management support to a given host platform and/or networking technology.

Thus, for cost-effective large-scale deployment, appropriate architectural mechanisms are needed to facilitate and enhance the portability of QoS-sensitive communication subsystems. We address this need by proposing *self-parameterizing protocol stacks* as the basis for the design and development of QoS-sensitive communication subsystems. Self-parameterizing protocol stacks extend traditional protocol stacks by providing support for efficient on-line profiling to construct a database of the system parameters that form the abstraction of the underlying communication subsystem and platform. To demonstrate feasibility, we have realized a self-parameterizing version of the guaranteed-QoS communication service described in Chapter 5. Our design and implementation methodology strives to minimize the overheads and perturbation induced in the data transfer path, while supporting relatively fine-grain performance profiling and system parameterization. Given that the only other alternative is manual profiling and parameterization, constructing communication subsystems using self-parameterizing protocol stacks is the most natural way to design portable QoS-sensitive communication software.

In the rest of the chapter, we first motivate the need for self-parameterizing protocol stacks for QoS-sensitive communication subsystems. We then compare and contrast with related work in operating system performance measurements, benchmarking, and resource monitoring. Subsequently we describe our design approach for self-parameterization of protocol stacks, and then highlight key features that ensure minimal perturbation during data transfer. Following this we describe our `CORDS`-based prototype implementation in the context of the guaranteed-QoS communication service developed for `ARMADA`, and present experimental evaluation results to demonstrate the feasibility of our approach.

Resources	Cost/Overhead Components		Attributes
CPU	API overheads protocol stack latency context switches, cache misses scheduling overheads		degree of data copying protocol stack layers CPU speed data structure efficiency
Memory buffers	buffer allocation/release buffer copy bandwidth DMA bandwidth		buffer management memory subsystem I/O bus bandwidth
Link	output	packet selection/dequeue packet transmission transmission complete interrupt	link bandwidth adapter design host support
	input	packet input (interrupt/polling) packet classification	

Table 6.1: Communication resources and their cost components.

6.2 Motivation and Problem Statement

In this section we motivate the need for mechanisms and techniques that enhance the portability of QoS-sensitive communication subsystems. We first discuss the nature of system parameters, costs and overheads that together constitute the communication subsystem abstraction utilized by admission control. We subsequently argue that a detailed manual profiling of the communication subsystem is not only complex and time-consuming, it may also be insufficient due to its static nature. We then make the case for an automated approach to profiling and parameterization.

6.2.1 Nature of System Parameters, Costs and Overheads

In developing the admission control extensions presented in Chapters 3 and 5, we identified a number of important system parameters and overheads that must be determined accurately for effective and correct admission control. Table 6.1 classifies various costs and overheads according to the associated communication resource, and highlights the key attributes that impact them. Note that one or more attributes may affect each cost component, i.e., no one-to-one correspondence is implied.

Each component listed in Table 6.1 impacts communication subsystem performance significantly and may be different for different host platforms. For example, the protocol stack latency is determined in part by CPU speed and the degree of data copying during protocol processing. A faster CPU or higher memory copy bandwidth would reduce protocol processing latency. Similarly, different processor and cache architectures would generate different context switching overheads and cache miss penalties, respectively. The cache performance of communication subsystems is also determined in large part by the composition of the protocol stack [133].

The I/O bus bandwidth in part determines the available DMA bandwidth to/from system memory. Since this affects the time spent moving data between memory buffers and the network interface, it also determines, along with link bandwidth, the transmission time for outgoing packets. Similarly, the classification applied to arriving packets may scale with CPU speed or benefit from specific demultiplexing support available on the network interface. Moreover, the parameters defining the granularity at which communication resources are multiplexed are typically platform dependent. Examples include the maximum packet size (as derived from the MTU of the attached network and protocol stack headers) and the number of packets processed between successive preemption points. These parameters cannot be fixed *a priori* and must be determined for the attached network and host platform, respectively.

It is clear from the above observations that the values taken by each of the system parameters utilized in admission control is highly platform-specific. In addition to platform dependencies, there can be workload dependencies as well due to variations in actual resource usage and non-ideal characteristics of real hardware and software resources. For example, cache miss penalties may show significant variation depending on the prevailing workload and degree of preemption. Similarly, queuing overheads on ordered data structures, such as CPU scheduling and packet selection overheads, are a function of the workload (e.g., the number of queued entries) as determined from the number of channels handlers or the number of packets awaiting transmission, respectively.

6.2.2 Infeasibility of Detailed Manual Profiling

In order to re-target the proposed architectural mechanisms and admission control extensions to other platforms, one could attempt to conduct a detailed profiling and parameterization of the communication subsystem, as we did for the prototype implementations described earlier. However, there are several practical difficulties that make such an option infeasible for large-scale deployment of QoS-sensitive communication subsystems across heterogeneous hosts and networks.

Detailed manual profiling requires intimate knowledge of the hardware and software components comprising the communication subsystem, especially those components that together provide QoS guarantees. Only the designers and developers of these components can be expected to have this knowledge. Even with this knowledge, obtaining performance traces manually and post-processing them is cumbersome and time-consuming. Further, correct and accurate parameterization also requires a deeper understanding of the admission control procedure employed in order to know *what* to profile within the communication subsystem. More importantly, detailed profiling may not even be a feasible option unless complete source code is available and instrumented carefully.

More importantly, the performance data thus generated may not even be correct, especially for workload-dependent system overheads, since the workload assumed for the profiling might not bear any resemblance to that seen in practice. While *static* (i.e., one-time) performance profiling suffices for most of the cost components listed in Table 6.1, workload-dependent overheads must be computed *dynamically*, as and when the workload changes. It follows that the static nature and complexity of detailed manual profiling makes it unsuitable for the development and deployment of QoS-sensitive communication subsystems.

6.2.3 Automated Approach to Performance Profiling

The above-mentioned arguments favor an approach that provides transparent and automated (or semi-automated) performance profiling and parameterization of QoS-sensitive communication software (and other components of the OS, in general). Self-parameterizing protocol stacks represent one such promising approach, since they are designed and implemented to allow system overheads and admission control parameters to be determined on-line and, if needed, dynamically. Coupled with user-level performance profiling tools, such an approach makes it relatively much easier to port QoS-sensitive communication software to other host and operating system platforms.

Note that our focus is primarily on accurately *determining* the system overheads and costs used in admission control. Accordingly, we do not consider mechanisms to reduce or optimize these system overheads and costs; a number of such efforts were described in Section 2.4 of Chapter 2.

6.3 Related Work

Before presenting our design approach towards self-parameterization, we compare and contrast self-parameterizing protocol stacks with related work in four areas: protocol stack performance, protocol benchmarking, operating system performance measurements, and resource monitoring in operating systems.

6.3.1 Protocol Stack Performance

As mentioned in Chapter 2, in recent years there have been numerous efforts to realize efficient protocol architectures and implementations that can preserve gigabit network throughputs at the application level. More germane to our focus, several recent efforts have also focused on experimentally quantifying, and identifying factors that influence, protocol stack performance.

A detailed study of the non-data touching processing overheads in TCP/IP protocol stacks is presented in [96]. The factors contributing to these overheads include network buffer management, protocol-specific processing, operating system functions, data structure manipulations, and error checking. This study reports an extensive breakdown of the overheads incurred at each layer of the protocol stack for a DECstation 5000/200 running the Ultrix 4.2a operating system. While very extensive in its overhead measurements and classification, this study also demonstrates the complexities involved in profiling the communication subsystem accurately. Moreover, since it is targeted for a specific host and OS platform, it reinforces the need for mechanisms to ease the task of profiling and parameterizing protocol stacks across multiple platforms.

The negative impact of data-touching overheads such as checksumming has also been studied extensively, and a number of techniques devised to improve data-copying performance [1, 97]. Similarly, much attention has been focused recently on appropriate buffer management for data copy elimination [25, 27, 130]. In contrast, our goal is to explicitly *account* for any copying cost incurred during data movement to/from applications, and *measure* this cost via appropriate profiling.

A study of the cache behavior of network protocols such as TCP and UDP reports widely variable effects on processing latency, depending on whether the cache is cold or hot [133]. Similarly, the importance of cache performance for small messages such as those found in typical signalling protocols is highlighted in [16]. This has significant implications for system parameterization since it highlights the difficulty in measuring various processing overheads accurately. Cache predictability may be improved via appropriate protocol implementation and compilation techniques [129], or via cache partitioning and appropriate OS support [110]. Any worst-case processing estimates are likely to be overly conservative. We note that this problem relates to memory subsystem design for modern processors, and is not related to the actual mechanism employed to profile communication subsystems.

All these efforts focus on quantifying protocol stack latency, identifying factors that affect this latency, or applying techniques to improve it. However, in addition to platform-specific performance evaluations, no information regarding the performance of the protocol stack is maintained within the communication subsystem. In contrast, we propose that traditional protocol stacks be extended to dynamically determine the performance of various components and maintain this information in terms of well-defined system parameters.

6.3.2 Protocol Benchmarking

The notion of *protocol benchmarks* was proposed in [160] for a comparative evaluation of different implementations of communication protocols and protocol stacks. To capture the communication behavior of real applications, a three-level model is proposed that comprises basic operations, basic applications, and compound applications. Basic operations are the primitive operations that can be invoked at the API individually, such as connect, disconnect, and the setting of data transfer options. Basic applications are sequences of basic operations, and can be used to model bulk transfers and request/response transactions. Compound applications are composed from basic applications, and are used to model the contention for resources such as the processor, memory, and access to the network; compound applications can be used to generate background load in a controlled manner.

This model has been used in a tool to compare different protocols and protocol implementations for varying levels of background load. In particular, it has been used to evaluate the performance of VMTP [36, 140], FTP and SunRPC [72], and to compare the performance of OSI TP4 and TCP [160]. This tool also allows specification of variable message sizes and the delay between successive messages sent on a given connection.

The primary focus of such protocol benchmarks and the above-mentioned tool is to compare the *end-to-end* performance of different protocols. As such, it is complementary to our focus on protocol stack performance and QoS-sensitive communication subsystem design. However, the application-level traffic generation model is of interest to us, since we also need appropriate traffic generators to trigger profiling and parameterization within the communication subsystem.

6.3.3 Operating System Performance and Resource Monitoring

Several studies have been undertaken to benchmark the performance of different operating systems with the goal of identifying performance bottlenecks. The first such study focused on the reasons behind the failure of operating system performance improvements to track performance improvements in hardware technology [142]. More recently, efforts have focused on developing a suite of portable operating system benchmarks for cross-platform performance comparisons [115] as well as detailed system analysis [24]. In all these efforts, however, the performance profiling is geared towards performance comparisons and the impact of OS-hardware interactions on operating system primitives. As such, there is no need to equip the operating system with information regarding the performance of individual components and primitives, nor is the profiling geared towards on-line system parameterization.

Self-parameterizing protocol stacks come closest in flavor to the idea of self-monitoring and self-adapting operating systems [157], although the two differ greatly in goals, scope and the approach adopted. It is proposed in [157] to perform continuous monitoring of operating system activity to construct a database of performance statistics, classify this data appropriately, and perform off-line analysis to construct a characterization of the system under normal behavior and detect anomalous behavior. In contrast, we focus exclusively on the communication subsystem, with the primary goal of parameterizing it via on-line profiling and making this information available to the admission control module. Moreover, due to the requirement of capturing constituent system overheads accurately, our on-line profiling operates at a finer time scale and may be disabled once appropriate measurements have been completed.

Several other efforts have also focused on operating system support for resource monitoring and application adaptation [104, 122, 135]. Such support becomes necessary in order to accommodate inaccurate or changing estimates of application resource requirements, and is geared primary towards adaptive multimedia applications. For our purposes, the iterative, well-defined nature of processing within the communication subsystem implies that once the application's QoS and communication requirements are known, its communication resource requirements can be derived accurately. However, the above-mentioned support for resource monitoring will be needed in the absence of an accurate specification. We note that while our primary focus is on provision of deterministic guarantees, self-parameterizing protocol stacks are desirable for provision of looser forms of QoS guarantees as well.

System support for automatic profiling and optimization of applications is described in [190], where the focus is primarily on improving application execution performance via statistical profiling and profile-based optimizations. We focus, instead, on accurate instrumentation-based profiling of the communication subsystem for system parameterization.

6.4 Design Approach for Self-Parameterization

In this section we outline our methodology and architecture for self-parameterizing protocol stacks to construct QoS-sensitive communication subsystems. We also highlight key challenges posed by self-parameterization, both in terms of feasibility and implications for admission control. As discussed in Section 6.5, the design approach adopted induces minimal overhead and perturbation during data transfer.

6.4.1 Overall Architecture

Figure 6.1 depicts our overall architecture for self-parameterizing protocol stacks for the realization of QoS-sensitive communication subsystems. The protocol stack is divided into five aggregate *modules*: API data buffering, message classification and queueing, protocol stack layers, packet classification and queueing, and link transmission and reception.

The API data buffering module refers to the API layer of the communication subsystem, such as the `RTC API ANCHOR` described in Chapter 5. The message classification and queueing module sits just below the API data buffering layer and corresponds to the per-channel message queues processed by the communication handlers. The protocol stack layers together comprise all the protocols that compose the protocol stack, except the bottom anchor protocol or the network device driver. The packet classification and queueing module corresponds to the outgoing packet queues processed by the link scheduler and the

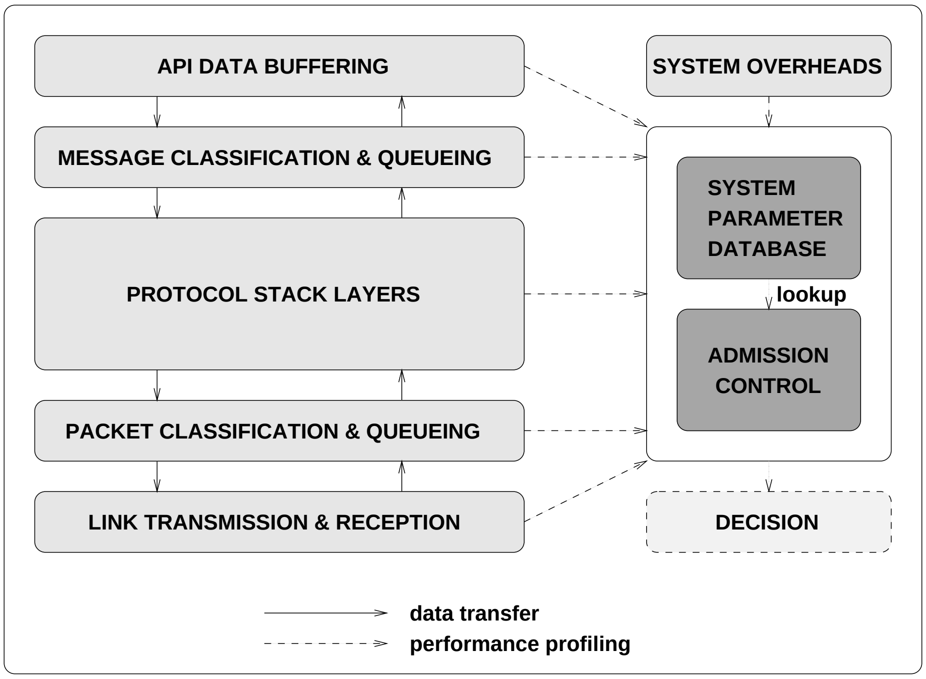


Figure 6.1: Architecture for self-parameterizing protocol stacks.

incoming packet queues processed by the communication handlers. The link transmission and reception module corresponds to the network device driver.

Within each module, the communication subsystem designer *identifies* appropriate profiling locations and puts appropriate *tracing hooks* at these locations for the send as well as the receive path. During data transfer, these trace hooks generate *profile samples* that are used to construct an in-memory *system parameter database* that resides within the communication subsystem. Other platform-specific system overheads not specific to the protocol stack, such as context switch overhead and cache miss penalty, may also be determined via on-line profiling. During the signaling performed for a new connection, admission control *looks up* the appropriate system parameters from the system parameter database in order to arrive at a decision.

A application-level *test suite* is used to perform data transfer through the protocol stack and trigger the profiling and parameterization code for the send as well as receive path. The on-line construction of the system parameter database is illustrated in Figure 6.2. As the protocol stack executes in response to data transfer, the profile samples undergo the following steps:

- *Sample classification and processing:* The incoming sample is first classified to deter-

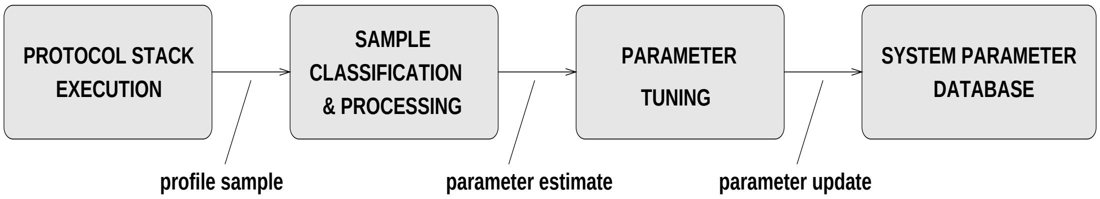


Figure 6.2: On-line construction of system parameter database.

mine the module that generated it and then processed. The processing could be as simple as storing the sample in the appropriate *trace buffer* for later reference during parameter tuning, or may involve aggregation to determine averages subsequently.

- *Parameter tuning:* From the samples thus collected, parameter estimates are derived for each of the system parameters maintained by the parameter database. This derivation could be based on simple averaging over all the samples collected, and may also keep track of the minimum and maximum values observed after ignoring outliers.

A *parameter update* is then applied to the system parameter database to record the corresponding value for the specified parameter. Note that parameter tuning would typically be performed only after a desired number of samples have been taken for each parameter of interest. However, the architecture allows parameter tuning to immediately follow sample classification and processing; this may facilitate calculation of dynamic workload-dependent overheads.

System parameters are represented in the database as unique *parameter variables* classified according to the protocol stack module associated with them. The parameter variables are queried during admission control, as explained in Section 6.4.2. Note that separation of parameter estimates from parameter updates, as shown in Figure 6.2, also helps isolate the parameter variables, and hence admission control decisions. While this does not affect the parameterization of workload-independent overheads, it has some implications for workload-dependent overheads, as discussed later. Since a host could be the source as well as destination of QoS-sensitive traffic, unique parameter variables are maintained for the send and receive paths, respectively.

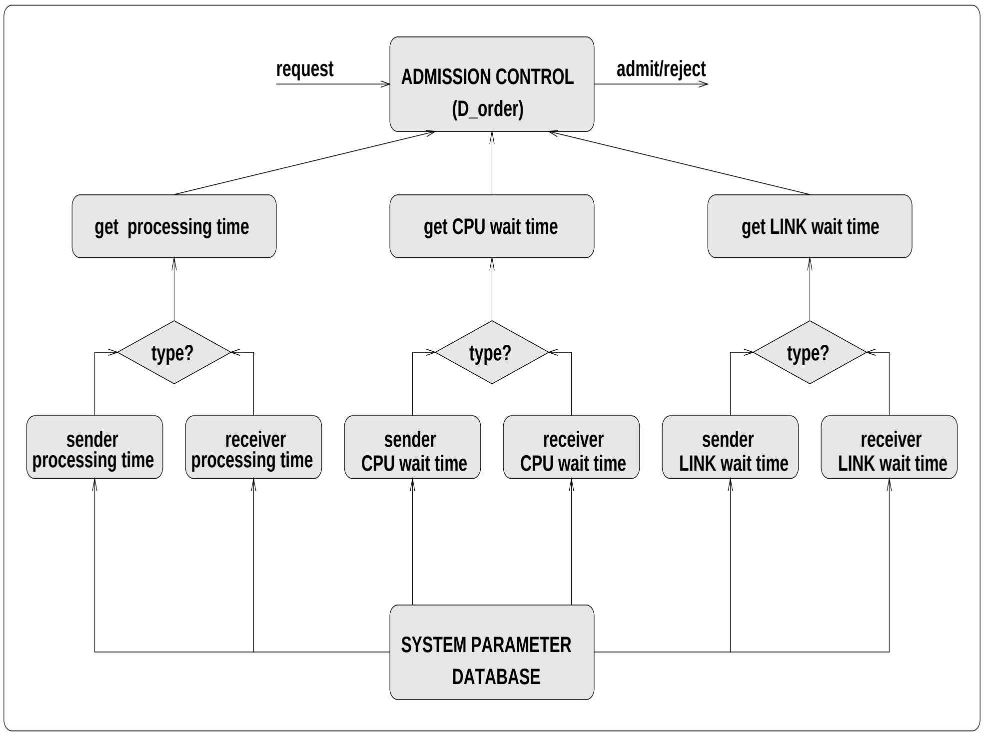


Figure 6.3: Internal structure of admission control procedure.

6.4.2 Structuring the Admission Control Module

The admission control module within the communication subsystem has to be structured appropriately in order to work correctly within the above architecture. Figure 6.3 illustrates the internal structure of the admission control procedure for sending as well as receiving hosts.

Upon invocation of admission control to admit a new request, `D_order` needs to calculate the request processing time (i.e., the message service time) and the request wait time, as explained in Chapter 4. The request wait time in turn is derived from the CPU wait time and the link wait time. The request processing and wait times are computed according to the extensions developed in Chapters 4 and 5. The type of the channel determines whether the sender or receiver processing and wait times are computed. This computation is performed using the parameter variables that constitute the system parameter database.

Querying the parameter database each time a request is processed ensures that the admission control procedure utilizes the most recent values calculated for the system parameters. Again, this has implications for dynamically computed system overheads that may vary with the resident workload on the host. Note that the system parameter database resides in the admission control module. As such, it exports a simple interface to query and update the parameter variables. Parameter variables may be queried during the parameter tuning phase to verify the consistency and accuracy of the parameter estimate generated. Parameter updates are issued only after this verification.

6.5 Minimizing Perturbation

Since profiling code actively resides in a self-parameterizing protocol stack, minimizing the perturbation during data transfer is paramount. The perturbation introduced by detailed profiling is a function of the placement of profile points, the cost of taking timestamps, and the processing of profile samples to generate parameter estimates. Also important is the degree of control exercised over the trace hooks placed at the appropriate profile points. While the profile points are placed inside different modules, it is desirable to have *centralized* control over the corresponding trace hooks; in our case this control resides within the self-parameterization module that keeps track of the generated profile samples.

6.5.1 Placement and Control of Profile Points

Referring to Figure 6.1, each module traversed by application data (i.e., a packet or message) will activate trace hooks at one or more profile points to generate profile samples. This applies to incoming as well as outgoing data, the only difference being the order of generation of the profile samples. However, without appropriate controls over the activation of trace hooks, a single packet may trigger multiple profile samples, thus exacerbating the processing latency experienced by that packet. We believe that per-packet perturbation should be minimized such that each packet or message triggers at the most one trace hook, and hence generates at most one profile sample, as it traverses the protocol stack. Multiple profile samples are gathered over different messages and packets. Note that the application-level test suite must be constructed to generate or consume sufficient number of messages sized appropriately.

```
prologue:
        if ( module_enabled )
             module_prologue_timestamp = timestamp();
        }
epilogue:
        if ( module_enabled )
            module_epilogue_timestamp = timestamp();
            value = module_epilogue_timestamp - module_prologue_timestamp;
            profile_sample (module_number, parameter_name, value);
        }
```

Figure 6.4: Prologue and epilogue processing for profile sample generation.

Number and location of profile points

Since the profile samples generated by each profile point must be stored until they can be processed, the number of profile points assumes significance. The required number of profile points is determined primarily by the granularity of overhead measurements required to achieve the desired form of system parameterization. Further, since profile points are placed in the code by the communication subsystem designer, increasing the number of profile points also increases the burden on the designer.

To minimize the number of profiling points, we associate a single profile point with each module wherever possible, for each direction of data transfer. We could instead have associated a profile point with each layer comprising the protocol stack. However, besides requiring each layer to be instrumented to add the profile points, per-layer profile points are unnecessary for correct system parameterization. For example, we only need to measure the time taken by a communication handler to process a single packet through all the protocol stack layers, i.e., the *aggregate* measurement suffices. Since protocol stack layers form a single module in Figure 6.1, it suffices to provide a single profile point each for outgoing and incoming traffic. Note that another benefit of per-module profile points is that the aggregate overheads measured include many of the cost components listed in Table 6.1, largely obviating the need to profile these components individually.

Frequency of profile sample generation

Even with per-module profile points as discussed above, it is important to exercise control over profile sample generation by these profile points. That is, it should be possible to individually enable/disable the module profile points and, when enabled, determine the frequency at which samples are taken. Figure 6.4 illustrates the profile generation methodology we adopt.

Each module that generates profile samples is assigned a unique *module_number* and a *module_enabled* flag. The *module_number* is assigned from the set $\{1, 2, \ldots, num\_modules\}$, where *num_modules* is the total number of modules generating profile samples. Each profile point corresponds to a *prologue* and an *epilogue*. The prologue is placed before, and the epilogue placed after, the cost component of interest. As data flows, the prologue for each module that is enabled triggers a timestamp as shown in Figure 6.4. Since the module is enabled, the epilogue takes another timestamp, computes the cost component, and generate a profile sample for further processing.

The above approach for profile sample generation ensures that only one module generates profile samples at a time, for as long as it is enabled. Thus, the entire profile sample collection for a given module is completed before enabling the next one. An alternate approach would be to enable all the modules for the entire duration of profile sample collection, and make sure that successive modules in the communication subsystem are activated in sequence by successive messages or packets. Thus, profile generation is performed by each module in more or less round-robin fashion. However, the approach we adopt is likely to have better cache performance since the corresponding trace buffer used to store samples may remain resident in the cache across successive message or packets.

Note that with this approach, while the API buffering module generates profile samples on a per-message basis, the protocol stack layers module generates profile samples on a per-packet basis. This facilitates on-line profiling of API data transfer overheads as well as the per-packet fragmentation and reassembly overhead incurred by the communication handlers. For the latter, the necessary distinction between the first and last packets is performed when classifying the profile samples, using the pre-defined message sizes generated by the test suite.

Timestamp cost and resolution

Timestamp cost can constitute a substantial part of the total profiling and parameterization overhead. High timestamp cost affects not just the profiling overhead, but also limits the resolution of the real-time clock. For example, for our platform timestamps using the real-time clock cost $\approx 15\ \mu s$. While this overhead is relatively high, it can be reduced drastically by using hardware performance counters provided in several modern processors [24]. Hardware cycle counters provide timestamps with resolutions of the order of just a few nanoseconds. The kernel-level profiling described in Chapter 7 is performed using such hardware counters.

For platforms with significant timestamp cost, the generated profile sample must be adjusted appropriately. For example, in Figure 6.4, the actual profile sample recorded would be $value - \tau_o$, where τ_o is the cost of taking a timestamp on the target platform. We note that on-line profiling via timestamps is no worse than off-line profiling, where timestamps must also be employed to measure individual cost components.

6.5.2 Deferred Sample Processing and Parameter Tuning

The profile samples generated by each module are classified and stored in appropriate in-memory per-parameter sample buffers. The actual processing of these samples, however, is only performed once a sufficient number of samples have been collected. This not only ensures minimal perturbation during profile sample collection, but also allows relatively sophisticated processing to be performed on the collected samples to obtain parameter estimates and related statistics. These parameter estimates may then be further tuned, e.g., made extra conservative, before updating the system parameter database. This process completes when the database has been suitably updated for each system parameter.

The processing performed on the collected samples may include calculation of simple averages and additional statistics such as the standard deviation. Often it is more accurate to discard some of the highest and the lowest values before computing parameter statistics [24]. This is true for us since the unpredictability of the underlying operating system may introduce occasional preemption between the prologue and epilogue processing in each module. An example of such preemption is the execution of the device input thread during processing by a incoming communication handler, as described in Chapter 5. The collected

Modules	Parameters	Cost(s) Represented
ANCHOR	ANCHOR_SEND_MSG	copy in and enqueue outgoing message
	ANCHOR_RECV_MSG_NW	dequeue and copy out incoming message
	ANCHOR_RECV_MSG_W	prepare to wait for incoming message
	ANCHOR_RECV_MSG_CB	callback in response to incoming message
PROTSTACK	PROT_SEND_PROC	fragment message and process outgoing packet
	PROT_RECV_PROC	process incoming packet and reassemble message
LINKDRIVER	LINK_OUTPUT_PKT	dequeue and transmit outgoing packet
	LINK_INPUT_PKT	classify, enqueue incoming packet; signal thread

Table 6.2: Modules and parameters in the prototype implementation.

samples must be sorted to eliminate the "outliers" and consider only the remaining samples for computing parameter statistics.

6.6 Implementation and Evaluation

In this section we describe our implementation and evaluation of a self-parameterizing version of the guaranteed-QoS communication service described in Chapter 5. We demonstrate experimentally that the mechanisms outlined in earlier sections suffice for self-parameterization of the protocol stack. The feasibility of our approach is demonstrated by comparing the system parameters thus computed with those obtained via manual profiling in Chapter 5.

6.6.1 Modules, Parameters and Sample Collection

The current implementation organizes the service protocol stack into three primary modules: `ANCHOR`, `PROTSTACK`, and `LINKDRIVER`. As outlined in Table 6.2, `ANCHOR` encompasses the API data buffering and message classification and queueing modules mentioned earlier, while the packet classification/queueing and link transmission/reception modules are encompassed by `LINKDRIVER`. `PROTSTACK` encompasses all the layers of the protocol stack except the top-level anchor and the bottom-level device driver. All the functionality required for self parameterization is contained within the *selfparam core*.

On initialization, each module explicitly registers with the selfparam core which in turn initializes appropriate state to ensure that profile samples generated by each module are classified and stored properly. Each module is assigned a parameter name space which uniquely identifies up to a certain number of parameters for which the module generates profile samples. State associated with each module includes flags encoding whether the module is registered or not, enabled or disabled, and the number of unique parameters associated with this module. A module remains disabled after registration until explicitly enabled by the test suite, as explained later. State associated with each parameter is more elaborate. A parameter is classified as *per-message* if the corresponding profile samples are generated once per message, or *per-packet* if the corresponding profile samples are generated once per packet. A parameter is *valid* if at least one profile sample has been generated for it.

Each parameter is associated with two timestamps, *before* and *after*, corresponding to the prologue and epilogue for a profile sample, respectively. Profile samples computed from these timestamps are stored in a trace buffer associated with each parameter. Note that this trace buffer records a scalar sample for per-message parameters and a sample vector for per-packet parameters; the sample vector keeps track of all per-packet samples for the same message. As explained below, the sample vector also facilitates distinct tuning of the first (or last) packet relative to the other packets of a message. Since parameters values can be a function of data size, samples are generated for several message sizes ranging from single-packet messages to messages with a relatively large number of packets, as explained in Section 6.6.2.

Profile points are placed at appropriate locations in the protocol stack according to the parameters utilized by admission control. Table 6.3 lists the interface the selfparam core exports to the modules for registration and sample collection. Each profile point corresponds to the invocation of `SP_tstamp`, with a flag indicating if this invocation constitutes a prologue or an epilogue. If it is the latter, the selfparam core collects a sample, classifies it as per the specified module and parameter, and stores it in the associated trace buffer.

6.6.2 Message Generation

Generation of profile samples is triggered by an application-level test suite that communicates with the selfparam core via the API listed in Table 6.3. The test suite first creates a best-effort (or real-time) channel from the sending to the receiving host, and then initializes

Interface	Routines	Description
Modules	`SP_register`	register module with self parameterization core
	`SP_tstamp`	generate timestamp and, if epilogue, collect sample
Test Suite	`SP_init`	initialize the self parameterization core
	`SP_continue`	specify message size for generated messages
	`SP_reset`	disable current module and enable the next

Table 6.3: API routines exported by the self parameterization (SP) core.

the selfparam core by informing the latter of its role as a sender (at the sending host) or a receiver (at the receiving host) via `SP_init`. Subsequently, at the sending host the test suite simply cycles through several message sizes selected *a priori*, periodically generating a pre-specified number of messages for each message size and sending them on the established channel. At the receiving host the test suite simply consumes and discards incoming messages on the channel. Each time a new message size is chosen, the selfparam core is informed by invoking `SP_continue`.

When sufficient number of messages of all sizes under consideration have been sent, the selfparam core is instructed via `SP_reset` to disable the current module and enable the next module for profiling and parameterization. Once all modules have been profiled the selfparam core initiates parameter tuning and update, as discussed next. Note that, while the API exported to the test suite is not essential, in that information could instead be shared implicitly between the test suite and the selfparam core, an explicit API affords greater flexibility in selecting an application-level traffic pattern for system parameterization.

6.6.3 Parameter Tuning and Update

The samples thus collected in the trace buffer are tuned to remove outliers and compute sample statistics such as averages, etc. Estimates of the relevant system parameters are then derived and the corresponding parameter variables updated to reflect the newly computed estimates. For per-message parameters parameter tuning is performed in a single step. The samples collected for each message size are sorted in decreasing order and the highest 20% samples (the potential outliers) discarded. An average value is then computed from the remaining samples and associated with the corresponding parameter.

Anchor Parameter	Manual Message Size		Self Message Size	
	1 byte	10k bytes	1 byte	10k bytes
ANCHOR_SEND_MSG	301	606	290	580
ANCHOR_RECV_MSG_NW	335	1097	325	1079
ANCHOR_RECV_MSG_W	53	51	43	46
ANCHOR_RECV_MSG_CB	95	96	95	98

Table 6.4: Anchor: comparison of manual and self parameterization (in μs).

Parameter tuning for per-packet parameters is performed in two steps. In the first step, the sample vector for each message (i.e., samples associated with individual packets of a message) is reduced to up to two sample estimates, one for the first or the last packet of the message, and one for the average of the remaining packet samples. The first or the last packet of a message are handled separately because the corresponding samples represent costs or overheads not present in the other packets. For example, for the parameter `PROT_RECV_PROC` the sample corresponding to the last packet of a message includes the cost of reassembling the message. Similar considerations apply for the parameter `PROT_SEND_PROC`, since the first packet of a message typically has a higher processing cost (e.g., due to cold cache misses) compared to the subsequent packets. In the second step, once the sample vector has been reduced to two sample estimates, these estimates are then averaged across the collected samples and the average values thus computed associated with the corresponding parameter.

Finally, appropriate estimates are computed for the system parameters used by admission control. Variables corresponding to parameters that are independent of data size are updated directly using the estimates already obtained. However, for parameters that depend on data size, such as `ANCHOR_SEND_MSG` which includes the time to copy in a message, two components of the overhead represented are computed: a startup component and a per-byte component. Given the message size, for example, admission control then computes a cost value for the corresponding parameter using these two components.

6.6.4 Experimental Results

We performed several experiments on the testbed presented in Chapter 5 to compare manual profiling and parameterization with self parameterization. Since the profiling mechanisms

Protocol Stack Parameter	Manual		Self	
	Packets in Message		Packets in Message	
	First	Other	First	Other
`PROT_SEND_PROC`	181	77	178	79
	Other	Last	Other	Last
`PROT_RECV_PROC (10k bytes messages)`	260	360	320	415

Table 6.5: Protocol stack: manual and self parameterization (in μs).

employed are essentially the same for both, the results produced via self parameterization should be very similar to those obtained via manual profiling. While the quantitative measurements should be mutually consistent, the qualitative benefits of self parameterization should far out-weigh those of manual profiling.

Tables 6.4, 6.5, and 6.6 list the measured values of the various system parameters for the `ANCHOR`, `PROTSTACK`, and `LINKDRIVER` modules, respectively. Results are shown for message sizes of 1 byte and 10K bytes; experiments with other message sizes yield similar results. The measurements for manual profiling are derived from the system profiling measurements presented in Chapter 5. As expected, the values obtained via self parameterization are in very good agreement with those obtained manually, typically within 5% of the latter, for parameters representing each module. Even for `PROT_RECV_PROC`, where the discrepancy between the two is unexpectedly higher, the measurements are very consistent if the total reassembly and message enqueue cost (the difference between the processing cost of the last and other packets of the message) is considered. While this cost is 100 μs (360 - 260) with manual profiling, self parameterization gives a cost of 95 μs (415 - 320). For the most part the discrepancy can be attributed to the differences in statistics computation for manual and self parameterization.

From a qualitative perspective, manual profiling and parameterization pales in comparison to self parameterization. Not only does it require careful profiling of the communication subsystem and knowledge of the system parameters used in admission control, the process of obtaining profile samples and post-processing them is tedious. The time to complete system parameterization before the service can be used typically ranges from several tens of hours to several days, especially if the various protocol stack components must be understood in order to place profile points.

Self parameterization, on the other hand, dramatically reduces this time to just a few

Link Driver Parameter	Manual		Self	
	Packet Size		Packet Size	
	1 byte	1.5k bytes	1 byte	1.5k bytes
LINK_OUTPUT_PKT	673	1775	680	1789
	Packets in Message		Packets in Message	
	First	Other	First	Other
LINK_INPUT_PKT	120	20	123	18

Table 6.6: Link driver: comparison of manual and self parameterization (in μs).

minutes, depending on the number of samples desired, the number of message sizes considered for the profiling, and the number of modules in the protocol stack. For example, if the message generation rate at the sender is 1 message/second, the number of modules is 3 (in the present implementation), the number of message sizes considered is 4, and the number of samples desired is 100, then the total time to system parameterization would be approximately 20 minutes. These gains are only possible because self parameterization takes the human out of the loop of profiling and parameterizing the communication subsystem.

In our experience, self parameterization has been extremely effective in keeping system parameterization up-to-date with software changes, performance improvements, and bug fixes. Once the profile points are in place, the designer can freely add new functionality to the communication subsystem, or optimize critical paths, and then simply run the parameterization test suite so that the system parameters reflect the performance of the latest version of the communication service. This, of course, assumes that the location of profile points is relatively static and that changes to the existing code does not invalidate the specified admission control procedure.

From the service designer's point of view, self parameterization allows for rapid performance testing of the service, thus facilitating an improved understanding of the performance characteristics of the communication subsystem. From a service user's point of view, self parameterization hides details of the actual procedure and system parameters employed for admission control. This allows rapid configuration of the service at the user's site and permits easy migration across various processing, memory, and networking hardware upgrades.

6.7 Summary and Future Work

In this chapter we addressed the issue of portability of QoS-sensitive communication subsystems, which require intimate knowledge of system parameters that together constitute an abstraction of the communication subsystem. Towards that end we proposed self-parameterizing protocol stacks that are designed with the ability to parameterize themselves appropriately during data transfer. Our design approach extends traditional protocol stacks with carefully placed and controlled profile points, in-memory trace buffers for profile samples, procedures for computing per-parameter statistics, and an in-memory system parameter database that records system parameters values utilized during admission control.

Self-parameterizing protocol stacks are the most natural way to enhance the portability of QoS-sensitive communication subsystems, especially for provision of deterministic QoS guarantees. Note that self-parameterizing protocol stacks by themselves do not make a communication subsystem portable across hetergeneous hardware and software platforms. Instead, they are designed with the goal of *easing* the burden of porting a guaranteed-QoS service, which involves not just the usual resolution of code incompatibilities, but also requires intimate knowledge of the performance of the (hardware/software) components involved. Such knowledge is best available with the service designer, who must consider system performance as an important ingredient *during* QoS-sensitive communication subsystem design. This is in sharp contrast with the design of traditional (best-effort) communication subsystems, where performance, while very important, is typically an after-thought.

There are several interesting directions in which this work can be extended. One of the most obvious is to integrate mechanisms for self-parameterization more closely within the protocol composition and path framework in `CORDS`. For example, it may be desirable to specify modules in the protocol graph itself, allowing them to be automatically registered when the appropriate x-kernel protocol is initialized. Similarly, the profile samples and associated trace buffers should be managed in a path-specific fashion. This is important since different QoS connections may take different paths through the communication subsystem, and hence incur different protocol processing costs. While we have focused primarily on static on-line profiling and parameterization, self parameterization could also be employed effectively to realize *dynamic* profiling and parameterization; with dynamic parameterization, the workload-dependent parameters are determined dynamically in response to workload fluctuations. Finally, for true end-to-end QoS guarantees, such self-parameterizing protocol stacks must be integrated with any QoS monitoring functions provided in the operating system.

CHAPTER 7

QOS SUPPORT IN TCP/IP PROTOCOL STACKS

7.1 Introduction

The preceding chapters have focused on architectural components and mechanisms to realize new communication subsystems that are QoS-sensitive. The rapid shift of the WWW (and the underlying Internet) from a text and graphics-oriented medium to one with live audio and video content, as evidenced by the emergence of applications such as InternetPhone and WebTV, has generated a significant demand for "better than best-effort" Internet connectivity.

Significant efforts are being made by the Internet Engineering Task Force (IETF) to enhance the service model of the Internet to support integrated services for voice, video, and data transport [22]. This in turn implies that the *existing* communication subsystems running TCP/IP protocol stacks in Internet hosts be modified for QoS-sensitive traffic handling according to integrated services standards. In this chapter, we concentrate on the design and implementation of QoS support on Unix-like (i.e., those supporting a sockets based communication system) Internet servers, the typical sources of multimedia data on the Internet.

In collaboration with researchers at the IBM T. J. Watson Research Center, we have designed and implemented architectural extensions to the sockets-based communication subsystem that enable RSVP-based integrated services infrastructure in the Internet [13]. One of the primary goals of our service architecture is to blend the QoS support with the existing TCP/IP stack and socket API, preserving the structure of the Unix networking subsystem. Applications not needing QoS support for communication should continue to run as is, and yet the QoS extensions should allow new applications to benefit from QoS enhanced network services.

Further, control overheads for QoS support should not have any detrimental impact on data path throughput. Our design is also influenced by the observation that with the rapid penetration of the Web, potentially any Internet site can be a content provider, and hence a source of multimedia data. This implies a wide variation in the capabilities of the Internet hosts and their connections to the Internet (network interfaces and links). Thus, the design must scale from small to large number of connections and accommodate network interfaces with widely differing capabilities. One of the primary goals of the architecture is to provide better service for QoS traffic, but not at the expense of best-effort traffic.[1]

We have developed a prototype implementation of our QoS architecture on RS/6000 based servers running AIX release 4.2 and equipped with ATM and IEEE 802.5 token ring adapters. Our prototype system supports RSVP-based QoS signaling and several IETF defined QoS classes, described in Section 7.2, with appropriate enhancements to the communication subsystems at the hosts (clients and servers) in the control as well as data planes of the protocol stack. The hosts are connected via prototype routers based on ATM switches enhanced for IP routing and IETF standards compliant QoS classes.

Given that the new service model will be widely deployed and utilized, it is important to understand the extent and nature of overheads imposed relative to the best-effort data path. That is, we are interested in ascertaining the performance impact on connections using integrated services in TCP/IP protocol stacks at end hosts to obtain QoS. Our QoS architecture has been carefully designed to have minimal impact on the default best-effort data path, which must remain as efficient as possible to maximize application throughput. The complexity and the overhead of supporting QoS is a critical factor that will ultimately determine the future evolution of integrated services on the Internet. To the best of our knowledge, there has been no experimental study that analyzes and measures the overheads of offering such services.

The primary contributions of this chapter are threefold: we (i) present architectural mechanisms for traffic policing, shaping, and buffer management at end hosts, (ii) quantify the control and data path overheads introduced by various QoS components via detailed kernel profiling, and (iii) discuss the performance implications of these overheads. We

[1] We are primarily concerned with servers, and hence the data transmission path.

focus primarily on exacerbation of data path latency due to QoS overheads for a *single* connection relative to the best-effort data path. The examination of similar effects in the presence of multiple QoS connections is left for future work. We demonstrate the efficacy of our architecture in providing QoS guarantees via application-level experiments. Our kernel-level measurements reveal that traffic shaping presents the most challenges, if it needs to be performed in software. This is primarily because of reliance on OS timer mechanisms and possible interaction with the operating system CPU scheduler.

We achieve very efficient traffic classification on the outgoing path by carrying the necessary information with each packet as it traverses the protocol stack. A key observation is that traffic policing and shaping overheads are largely offset by gains due to shorter data paths for QoS sessions relative to the best-effort sessions. These gains include (i) the savings due to pre-allocation of per-session buffers, and (ii) for the ATM network, the savings due to a faster path through the network interface layer. In the latter case the data path latency for compliant packets can even be *smaller* than the latency of the default best-effort data path. In effect, our architecture transfers some of the traffic classification and buffer management overheads from the data path to the control path. To keep the overhead of packet queueing and scheduling low, we use a very simple queueing structure or utilize hardware support if available. Using the QoS component overheads as motivation, we identify the implications of providing QoS support in TCP/IP stacks and possible approaches to mask and/or reduce some of these overheads.

We note that the overhead incurred due to traffic shaping is a function of the frequency with which an application generates non-compliant traffic, i.e., the extent of the mismatch between the stated traffic specification and actual traffic generation. It seems desirable that an application conforming to its stated traffic specifications receive better performance compared to one that frequently violates its traffic specification. At the same time, due to the difficulty associated with specifying meaningful traffic characteristics, occasional violation of traffic specifications should be tolerated. However, overheads incurred in preventing persistently misbehaved applications from consuming excess resources may constitute a significant loss of useful resource capacity. The scalability of the QoS architecture is, therefore, contingent in part upon the accuracy with which an application specifies its run-time communication behavior.

Protocol stack performance and optimizations of existing implementations has been the subject of numerous research articles, including some very recent ones [16,129,176]. However, all of these studies focus on the traditional best-effort data path. Our study assumes significance in that it quantifies the performance penalty imposed by *new* data-handling components in the protocol stack, and their impact on the best-effort data path. With the popularity of networked multimedia applications on the Internet, the overheads imposed by these components play an increasingly important role in communication subsystem and operating system design. To the best of our knowledge, ours is the first study of its kind to identify the performance impact of QoS support in TCP/IP protocol stacks While the results reported here are for host protocol stacks, some of our findings are also applicable to routers and switches participating in an integrated services Internet.

In the rest of the chapter, we first present a brief overview of RSVP and integrated services, followed by an overview of our QoS architecture and components. We then present results from application-level measurements, on our prototype implementation on RS/6000 based servers running AIX,[2] that demonstrate the efficacy of this architecture and the need for the features provided. Using detailed kernel profiling of our implementation, we quantify the data path overheads introduced by various QoS components. Based on these results, we argue that traffic shaping presents the most challenges. We then identify the implications of, and suggest approaches to mask and/or reduce, the overheads involved in traffic shaping. Finally, we conclude with a summary of the main contributions of the chapter.

7.2 RSVP and Integrated Services: An Overview

Below we present a brief overview of the RSVP protocol and the service classes under discussion in the IETF. A complete description of RSVP is provided in [23], while details on different service classes can be found in [159,183].

7.2.1 RSVP: An End-to-End View

Figure 7.1 shows an RSVP-based QoS architecture depicted data sources (S1, S2), destinations (D1, D2, D3) and IP routers (R1, R2, R3). The sources as well as the destinations run RSVP daemons that exchange RSVP messages (PATH and RESV) on behalf of their hosts. In general reservations can be made on multicast sessions, as depicted in Figure 7.1.

[2] AIX is a BSD variant operating system.

Senders S1 and S2 send PATH messages to the multicast group address comprising D1, D2, and D3. The PATH messages travel through the network to all members of the multicast group and PATH state is established at all RSVP-enabled routers in the multicast tree; each of D1, D2, and D3 receives two sets of PATH messages. PATH messages arriving at their intended receiver(s) are processed by the RSVP daemon.

The receiver D1 intends to make (possibly different) reservations on the flows originating from S1 and S2, and sends RESV messages RESV1 and RESV2 in response to PATH1 and PATH2, respectively. The receiver D2 wants to make a reservation only on the flow originating at S1 and sends RESV message RESV1. The receiver D3 on the other hand decides not to make any reservations and does not send any RESV messages in response to the PATH messages from S1 and S2. As RESV messages from the receivers traverse upstream to the senders, they are intercepted by RSVP-enabled routers and if sufficient local resources are available, reservation soft state is established in the routers.

The RESV messages are also merged at the appropriate merging points. An end-to-end reservation is successfully established when the RESV message reaches the sender and is successfully processed by the local RSVP daemon. Eventually, a reservation tree is established with the senders as the root and the receivers requesting reservations as the leaves. Note that PATH and RESV messages are independent of the data flow from the sender to the receivers although they follow the same route through the network. Hence, a reservation can be established before or any time after the data flow starts.

Additional details of refreshing PATH and RESV states and handling route changes are provided in [23]. An RSVP flow is uniquely identified by the five-tuple `<protocol,src address,src port,dst address,dst port>`. Filters are set up at routers and hosts to classify packets belonging to an RSVP flow and treat them in accordance with the reservation made on the flow. Note that, RSVP is just a signaling protocol that establishes reservation soft states at the end-hosts and routers. Honoring the reservations requires, among other things, resource and traffic management at the hosts and routers. The resource and traffic management mechanisms depend heavily on the service classes supported.

7.2.2 Service Classes

Two important service classes currently under standardization by IETF are (i) guaranteed service [159], and (ii) controlled load [183] service. Guaranteed services is targeted at provid-

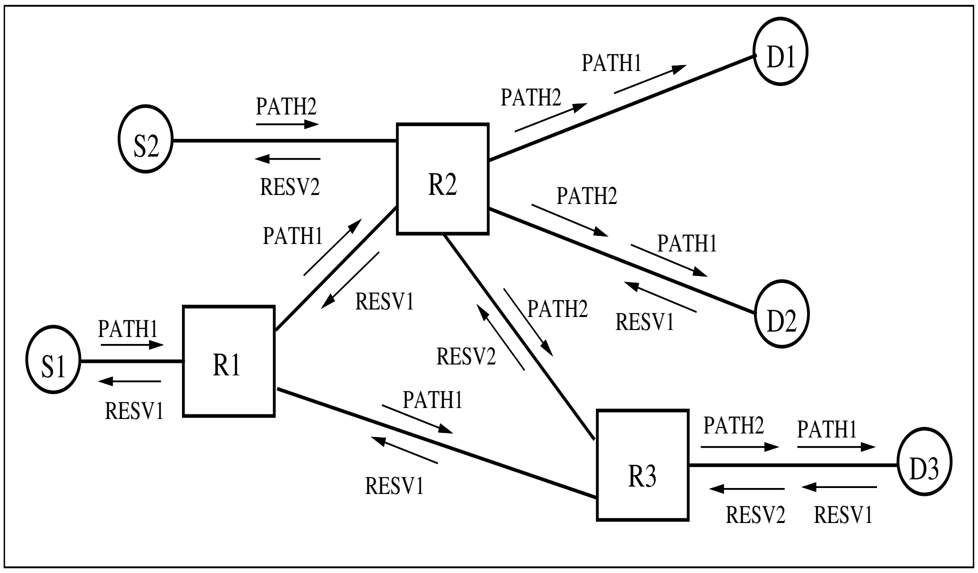


Figure 7.1: PATH and RESV messages in RSVP.

ing applications with a mathematically provable end-to-end delay bound using appropriate buffer and bandwidth reservation at all network elements. It guarantees that datagrams will arrive at the receiver within the guaranteed delivery time and will not be discarded due to queue overflows, provided the flow's traffic stays within its specified traffic parameters. This service is intended for applications that need firm guarantees on loss-less on-time datagram delivery. Some interactive audio/video applications and applications with hard real-time requirements fall in this category.

On the other hand, the controlled load service is designed for adaptive applications that do not need any specific quality of service, but can exploit the increased predictably in network performance. The end-to-end behavior provided to an application by controlled load service closely approximates the behavior visible to applications receiving best effort service under `unloaded` network conditions. Controlled load service is intended for the broad class of adaptive real-time applications (such as vic, vat, nevot, etc.) developed for today's Internet that are sensitive to overload conditions.

To avail these services, a connection has to specify a traffic envelope, called Tspec, that is carried in the PATH message and includes a long term average rate, a short term peak rate, and the maximum size of a burst of data generated by the application. An application generating MPEG coded video could specify the average rate to be the long term data rate, peak rate to be the link bandwidth, and burst size to be the maximum size of a frame.

Tspec also specifies the maximum and minimum packet sizes to be used by the application. For guaranteed service, traffic should be shaped to conform to the traffic specification. For controlled load traffic shaping at the source is not mandatory. Violating packets belonging to a controlled load session are allowed to pass the conformance check as best effort traffic. In addition to Tspec, guaranteed session specifies an Rspec containing the required rate of service and a slack term. The required rate of service should be at least as large as the long term average rate specified in the Tspec. The slack term signifies the difference between desired delay and delay obtained with the specified rate of service, and can be utilized by the network to reduce the reservation level of the flow. For controlled load service, there is no separate Rspec. Each controlled load session is guaranteed a rate of service equal to the long term mean rate specified in its Tspec. A session may receive better service if there is spare capacity in the system.

7.3 Architectural Overview and QoS Components

We now give an overview of the RSVP-based QoS architecture for end hosts, and the components that comprise this architecture. Additional details on the internals of the architecture can be found in [13]. Figure 7.3 shows the software architecture of an RSVP enabled host. In this example, a number of applications are using RSVP signaling for resource reservation. The applications use an RSVP API (RAPI) library to communicate with the RSVP daemon running on the host. The RSVP daemon is responsible for translating the RAPI calls into RSVP signaling messages and local resource management function calls. For local resource management, the RSVP daemon interacts with the `QOSMGR` over an enhanced socket interface.

QoS extensions to the protocol stack are spread across both control and data planes. The `QOSMGR` is the key component in our architecture. It plays a critical role in the control plane, and also in the data plane, of the protocol stack. It is entrusted with managing network related resources, such as network interface buffers and link bandwidth. It is also responsible for maintaining reservation states and the association between the network sessions and their reservations. Moreover, it performs traffic policing and shaping for sessions directed to network interfaces that do not perform these functions in hardware, a category that covers an overwhelming majority of present-day LAN interface adapters. Besides the introduction

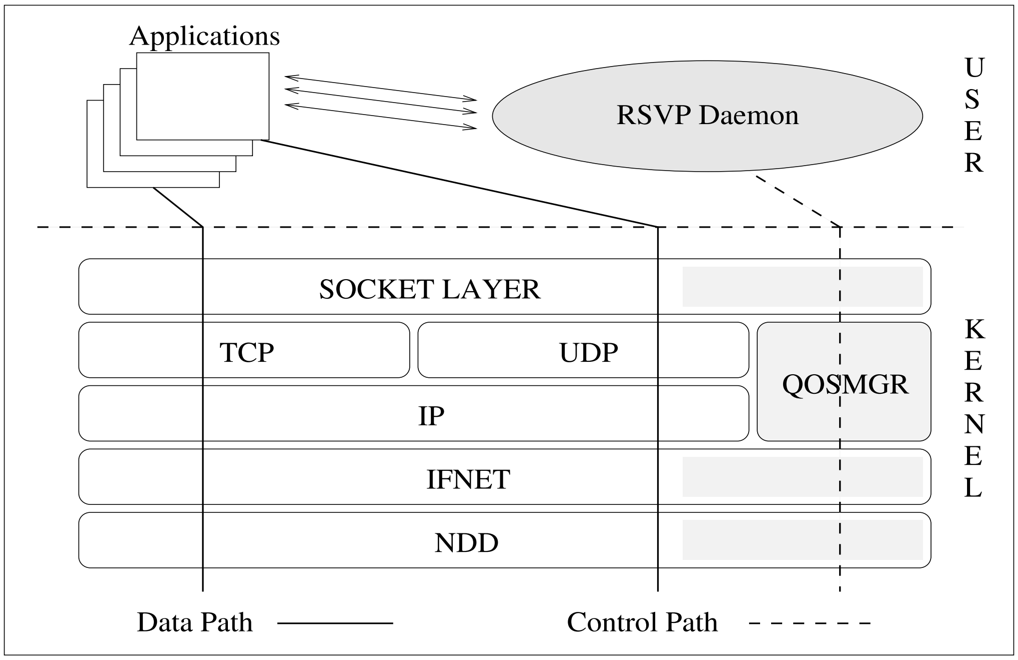


Figure 7.2: Protocol stack architecture and QoS extensions.

of `QOSMGR`, the enhancements to the protocol stack also include extensions to the socket layer, network interface drivers (IFNET), and (NDDs) for classification of network bound datagrams to the sessions they belong to, and handling them in accordance with the reservation.

7.3.1 Control Functions

The control plane is responsible for creating, managing, and removing reservations associated with different data flows. When a reservation is requested by an application, the `QOSMGR` computes and pre-allocates the buffer space required by the application on a session-specific basis. Buffers are maintained as a chain of fixed-size *mbufs*, the traditional memory buffers used by Unix networking software, with the buffer size derived from the advertised traffic specification in the application's reservation request. In cooperation with the network device drivers and the network interface layer, the `QOSMGR` also reserves network bandwidth commensurate with the reservation.

The `QOSMGR` is also responsible for binding the data socket with a connection-specific QoS handle. This handle, which directly identifies the associated reservation, reduces the task of packet classification to a single direct lookup, and is used subsequently and correctly handle traffic originating on the data socket. The traffic classification function is thus a statically compiled packet filter, as opposed to a dynamically generated one [59].

7.3.2 Data Transfer

During data transfer on a QoS connection, the socket layer interacts with `QOSMGR` to obtain a buffer for each packet generated by the application. The `QOSMGR` performs traffic enforcement (policing and/or shaping) relative to the reservation identified by the QoS handle, as explained below, and returns a buffer to the socket layer. The socket layer copies the corresponding amount of application data into this buffer before initiating further protocol processing.

In general, policing determines whether a packet is compliant or not. To perform traffic policing, `QOSMGR` maintains two auxiliary variables, t_m and t_p, for each reservation. Informally, t_m is used to check whether the application is conforming to the long term average rate specified in the Tspec; t_p is used to enforce the short term peak rate. A transmission request is non-compliant if current time t is less than $max(t_m, t_p)$ (which represents the *time to compliance*); otherwise the request is compliant. For compliant requests, `QOSMGR` allocates a buffer from the associated reservation's pre-allocated buffer pool, and returns this buffer to the socket layer. A non-compliant request, on other hand, is handled in one of several service specific ways: marked and transmitted at a lower priority, or delayed until compliance (shaping). If transmitted at a lower priority, `QOSMGR` allocates a best-effort buffer for the packet and marks it appropriately for correct handling by lower layers of the protocol stack.

If the service specification so mandates, `QOSMGR` must *shape* (i.e., delay) non-compliant traffic until compliance. Our architecture supports two mechanisms for traffic shaping: *session-level shaping* and *datalink-level shaping*, the two being differentiated by their respective position in the protocol stack. In session-level shaping (effectively performed at the session layer), `QOSMGR` blocks the corresponding application thread for $\delta = max(t_m, t_p) - t$ time units. This is transparent to the socket layer, which essentially sees a delay in the allocation of a buffer to that packet. Session-level shaping is not required if the network interface has the capability to perform datalink-level shaping, as in ATM networks, or such capability is provided in the NDD, as is possible with other LAN technologies. In datalink-level shaping, the NDD buffers packets until compliance before transmitting them into the network. In this mode, `QOSMGR` simply manages the reserved buffers and flow-controls the

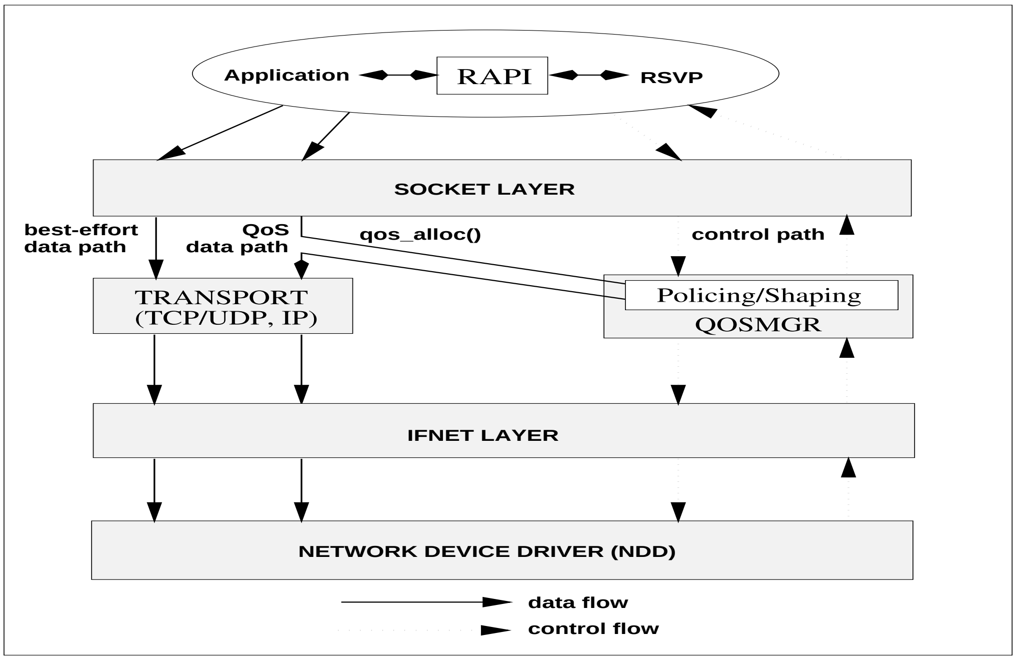


Figure 7.3: Best effort and QoS data paths.

send thread based on the availability of buffers for that connection. Hence, while session-level shaping controls execution of application threads, datalink-level shaping controls the transmission of packets.

The support required for link bandwidth management depends greatly on the capabilities of the attached network interface controller (NIC). For an ATM NIC no software support for traffic shaping and scheduling is required since these functions are supported in hardware. If the attached NIC does not support QoS functions, as in legacy Ethernet and Token Ring networks, NDD extensions are required to support per-connection QoS via packet queueing and scheduling [13].

Figure 7.3 shows best effort and QoS data paths, while Figure 7.4 illustrates the data path through `QOSMGR`. For a packet associated with a reservation, the socket layer obtains a buffer by calling `qos_alloc()`, which transfers control to the `QOSMGR`. It first checks whether the application has enabled policing for that reservation (by setting the *police flag*) (step 1). If the associated reservation does not have the police flag set, `QOSMGR` checks if a buffer is available for the reservation (step 2), and, if one is available, returns it to the socket layer after setting the buffer priority according to the type of reservation (step 3). Otherwise, if the application's send call is blocking (step 4), it blocks the calling thread until a buffer is

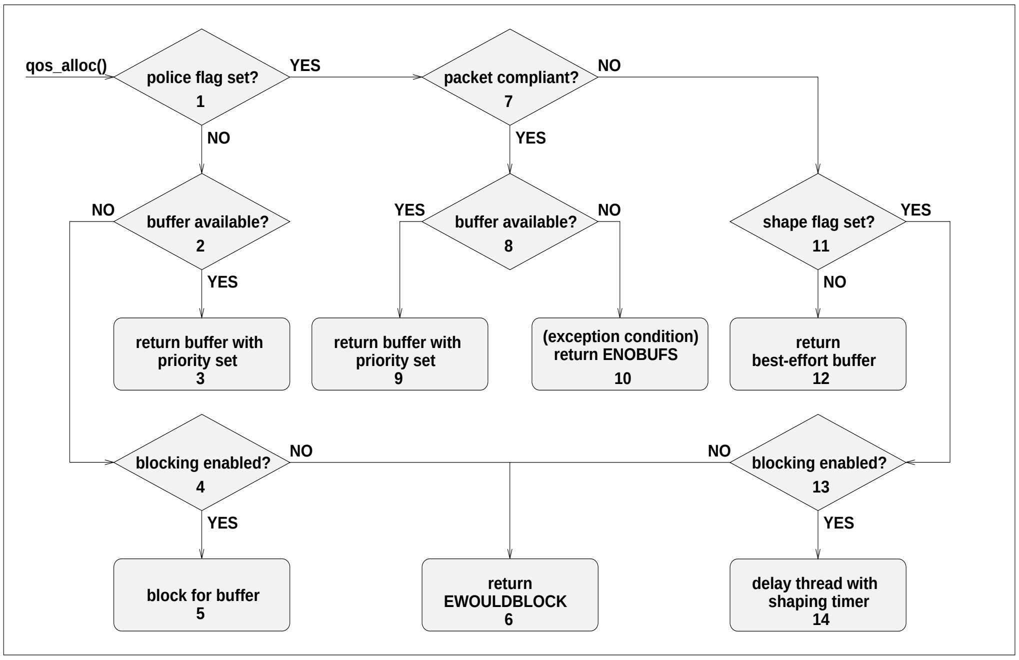


Figure 7.4: Data path through QOSMGR.

freed (step 5). Instead, if the application's send call is non-blocking, it returns `EWOULDBLOCK` to the socket layer (step 6).

On the other hand, if the application has enabled policing on the associated reservation, `QOSMGR` applies the policing function to determine if the packet is compliant (step 7). For a compliant packet, `QOSMGR` checks if a buffer is available (step 8) and returns a pre-allocated buffer if one is available (step 9), with the appropriate priority set so that the interface layer can service the packet appropriately. Unavailability of buffers for compliant packets is an exception condition since a compliant packet must always have buffers available, assuming that the reservation and policing computation is correct; in this case `QOSMGR` returns `ENOBUFS` to the socket layer (step 10).

If the packet is non-compliant, `QOSMGR` checks if the *shape flag* is set for the reservation (step 11). If not, it returns a best-effort buffer allocated using the traditional mbuf allocation calls (step 12). If the shape flag is set, but the application's send call is non-blocking (step 13), `QOSMGR` returns `EWOULDBLOCK` to the socket layer (step 6). However, if the application's send call is blocking, `QOSMGR` shapes the non-compliant packet (and hence traffic on the associated connection) by blocking the calling thread until its compliance time (step 14).

After transmitting a packet, the NDD notifies `QOSMGR` of a free buffer. The `QOSMGR` returns the freed buffer to the corresponding session-specific buffer pool and wakes up *one* of the session threads blocked waiting for buffers. Only one thread is woken up since only one buffer has been made available for consumption; this keeps the data path efficient by avoiding unnecessary thread wakeups and subsequent context switches. The thread thus woken up returns to the socket layer after obtaining the newly-freed buffer, and continues execution, i.e., processing the packet, through the protocol layers.

Note that in our architecture, applications output *packets* to the socket layer. Thus, our architecture is naturally suited to techniques such as application-level framing [41] and can be used with protocols such as RTP [155]. For example, it supports user-level fragmentation and protocol processing performed in the application's address space. Accordingly, it is well-suited to multi-threaded multimedia applications that have distinct *data generator* and *data exporter* threads. While the data generator threads produce multimedia objects (e.g., video frames) for network output, the data exporter threads may fragment these large objects into packets and transport them via UDP on QoS connections established by the `QOSMGR`.

7.4 Efficacy of the QoS Architecture

In this section we demonstrate the efficacy of our QoS architecture in providing applications with the requested QoS. We have implemented the QoS architecture described in the previous section on RS/6000 based servers running AIX release 4.2 and equipped with ATM and IEEE 802.5 token ring adapters. For the rest of the paper we focus on the ATM network implementation, although much of the discussion is also applicable to the token ring implementation. Since our ATM adapters provide QoS support in the form of VC setup/teardown, traffic policing and scheduling, traffic shaping is performed by the QoS Manager, if needed, in addition to traffic policing to guide buffer allocation. Most of this supported is needed for the token ring adapters as well.

7.4.1 User-Level Performance Measurements

As a first step, we measure user-level performance of the control and data paths in the prototype system. These experiments stress the functional aspects of the system by exercising the various data paths through the `QOSMGR`. For our experiments we have extended the `netperf` [137] program to interact with the `QOSMGR` and create QoS sessions. We have

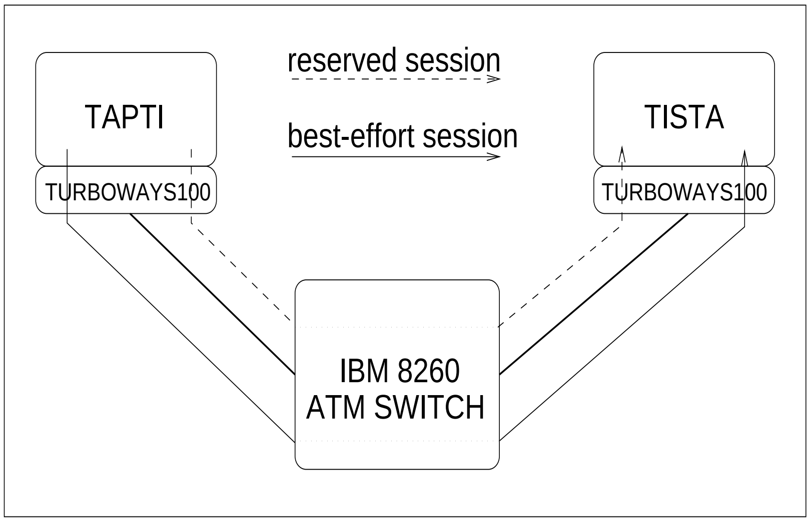


Figure 7.5: Experimental testbed for prototype implementation.

instrumented `netperf` to permit the creation of sessions with different options for local traffic control collect user-level statistics for packet transmission time. The traffic control options provided allow setting up reservations with or without policing and/or shaping. This allows us to incrementally assess the efficacy of each traffic control function performed by `QOSMGR`.

All our experiments are performed between two machines – (1) `tapti` – RS/6000 model 42T with 120MHz PowerPC 604 CPU, and (2) `tista` – an RS/6000 model 530 with a 33.4 MHz POWER CPU. Both machines run AIX version 4.2 and are equipped with 100Mb/s Turboways100 ATM adapters connected by an IBM 8260 ATM switch, as shown in Figure 7.5. The purpose of using the RS/6000 model 530 is to observe the impact of the new communication architecture on the vast existing base of older and slower servers. Although the POWER machine runs at a much lower clock speed, its overall performance is not proportionately worse than that of the PowerPC machine. The POWER and PowerPC CPUs are architecturally different and the differences in their clock speed are not directly reflective of their relative processing power.

Control Path Latency

Table 7.1 shows the application level latencies in creating (ADDRESV) and removing (DELRESV) a reservation between `tapti` and `tista`. The latency involved in creating a reservation includes the time taken to create local reservation states as well as perform signaling

Machine	ADDRESV	DELRESV
tapti	16.50ms	0.27ms
tista	27.00ms	3.40ms

Table 7.1: Control path latencies.

across the ATM network to setup a VC for the RSVP flow. The removal of a reservation includes cleaning up the local reservation state and tearing down the VC associated with the RSVP flow. However, the latency seen by an application in DELRESV does not include the time taken to tear down the VC across the ATM network. This explains the large difference between the latencies in ADDRESV and DELRESV. Note that the most significant part of ADDRESV latency is contributed by ATM signaling overhead.

Data Path Performance

As mentioned earlier, we use `netperf` to measure application-level UDP performance between `tapti` and `tista` across the ATM switch, for different packet sizes and traffic control options. Table 7.2 shows the results from four experiments with `tapti` as the traffic source and `tista` as the traffic sink.

Experiment I: Table 7.2 shows the number of packets transmitted and received without any reservations, i.e., on a best effort connection (the baseline performance). These packets traverse the best-effort path through the system and are carried on the best-effort ATM VC between the two systems. The default settings for the best-effort VC are set to UBR with a peak transmission rate of 50Mb/s.

These results reveal very low *goodput*, as measured by the number of packets that are successfully received by the receiver. This is because the receiver is unable to cope with the unrestrained transmissions by the sender and is overrun. One expects the goodput to increase once the sender's transmission rate is enforced (policed).

Experiment II: In this experiment, a controlled load reservation is created with a average traffic rate corresponding to 1000 pkts/s (the rate varies with packet size when expressed in bytes/s). To illustrate the effects of traffic policing, the peak rate for the session is also set to correspond to 1000 pkts/s, while the bucket depth is set to correspond to 10 pkts. The reservation results in the creation of a new QoS VC between `tapti` and `tista` with the appropriate traffic parameters expressed in terms of ATM cells.

Expt.	Connection Type		Message Size (in bytes) 1400	2000	4000
I	Best Effort	Transmit	35294	27481	25526
		Receive	32	19	13713
II	Reserved	Transmit	45941	38985	29614
		Receive	9838	10027	9973
III	Reserved with Policing	Transmit	28306	22803	19174
		Receive	653	5494	6723
IV	Reserved with Policing and Shaping	Transmit	10315	10243	10158
		Receive	9821	10025	9972

Table 7.2: Data path performance.

In this experiment, traffic policing is disabled in the `QOSMGR` and no packet-level policing is performed. Hence, the generated traffic is all directed to the QoS VC by the IFATM network layer. The results show that ATM level traffic policing is performed on this session and packets in excess of the requested traffic rate are dropped at the source. The receiver thus receives packets at a lower rate (as compared to that in `Experiment I` and is able to process the packets, resulting in higher goodput relative to that observed in `Experiment I`, as shown in Table 7.2. However, the goodput still does not match the packet transmission rate at the sender.

Experiment III: In this experiment, the same reservation as in `Experiment II` is created, but with traffic policing enabled in the `QOSMGR`. This corresponds to the recommended behavior for the controlled load service [183]. In this case, the `QOSMGR` performs packet level policing for the session and transmits all excess (i.e., non-compliant) traffic as best effort, i.e., it does not shape non-compliant traffic. The results show that the number of packets successfully transmitted (the goodput) is higher than that in `Experiment I` but lower than that in `Experiment II`. The sender still overruns the receiver since the excess best-effort traffic is being sent to the receiver over the default best-effort VC.

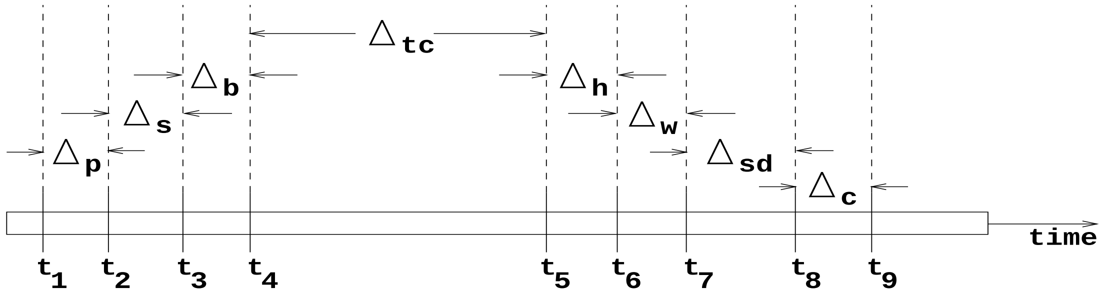
Experiment IV: In this experiment, both traffic policing and traffic shaping are enabled in the `QOSMGR`. This corresponds to the recommended behavior for the guaranteed service [159], and may also be applied to the controlled load service. In this case, the `QOSMGR` performs packet level policing and blocks the application thread whenever it tries to transmit data in excess of the associated reservation. As shown in Table 7.2, the number of packets transmitted (goodput) closely matches the reservation (sending rate). This rate is within the capabilities of the receiver, which is now able to receive most of the packets that are transmitted (on the QoS VC).

7.5 QoS Component Overheads

The results of the previous section demonstrate the efficacy of the QoS architecture in providing QoS to applications. However, as mentioned earlier, due to the expected wide-scale deployment of such QoS support, it is essential to identify the overheads imposed by the QoS support provided relative to the best-effort data path. Clearly, excessive overhead would limit the scalability of the QoS architecture and discourage use of the new service model. In this section we identify the overheads associated with the QoS components and experimentally quantify them using our prototype implementation. The next section discusses techniques to reduce or mask some of these overheads.

Figure 7.6(a) illustrates the timeline of events as a packet passes through the `QOSMGR`. An outgoing packet "arrives" at the `QOSMGR` at time t_1. The packet's time-to-compliance (Δ_{tc}) is computed as part of the policing function and by time t_2 it is known whether the packet is compliant or not; the overhead incurred due to the policing function is Δ_p. For a non-compliant packet (and hence one that needs to be shaped) a shaping timer is started at time t_2, incurring an overhead Δ_s. Subsequently, the calling thread is blocked on a semaphore, incurring an overhead Δ_b; at time t_4 the CPU can be reallocated to another thread.

At time t_5, after a delay of Δ_{tc} from time instant t_3, the shaping timer expires, and an overhead of Δ_h is incurred by the kernel in searching for the timer block and invoking the corresponding handler. The handler simply delivers a wake-up to the blocked thread and exits. Waking up a thread incurs an overhead of Δ_w, and at time t_7 the thread is residing in the CPU run queue, i.e., has been made runnable. The operating system's CPU scheduler allocates the CPU to this thread (i.e., schedules the thread for execution) at time t_8 after a scheduling delay of Δ_{sd}. The thread first stops the shaping timer, which incurs an overhead Δ_c, and continues processing the now-compliant packet from time t_9 onwards.



(a) Timeline of events in QOSMGR data path

Time	Event Description
t_1	packet arrival
t_2	time to compliance computed
t_3	shaping timer set
t_4	thread blocked
t_5	shaping timer expires
t_6	shaping timer handler invoked
t_7	blocked thread woken up
t_8	thread scheduled to run
t_9	shaping timer cancelled
$> t_9$	thread continues processing packet

(b) Time instants and corresponding events

Symbol	Overhead Component
Δ_p	policing overhead
Δ_s	overhead to start shaping timer
Δ_b	overhead to block thread
Δ_{tc}	time-to-compliance of this packet
Δ_h	overhead to handle shaping timer
Δ_w	overhead to wake up blocked thread
Δ_{sd}	delay in scheduling thread
Δ_c	overhead to stop shaping timer

(c) Overheads associated with shaping

Figure 7.6: Timeline of events and associated overheads during shaping.

7.5.1 Kernel Instrumentation and Measurement Methodology

We have instrumented the network subsystem of the modified AIX kernel for detailed profiling of the data path of reserved sessions using the native tracing facility. Measurements reported here are from both `tapti` and `tista`. The trace facility captures a sequential flow of time stamped system events, providing a fine level of detailed system activity. A trace event can take several forms, and consists of a hookword, optional data words, and an optional time stamp. The hookword is used to identify the specific event being traced. To minimize tracing overhead, our event records use only the hookword and timestamp. Timestamps are taken by reading a real-time clock. The real-time clock is an integral part of POWER and PowerPC CPUs used in RS/6000s, and can be read directly from the kernel as well as from the applications. We use a two-instruction assembly language routine to read two 32-bit clock registers with minimal overhead. The timestamps are of microsecond granularity.

Referring to Figure 7.6, we measure Δ_p, Δ_s, Δ_w, and Δ_c by taking timestamps before and after the corresponding kernel calls, computing the difference, and averaging over a large number of invocations. For this purpose, we utilized the functionality of the `QOSMGR` (which is realized as a protocol module, as mentioned earlier) to send it appropriate user-level commands and triggering our profiling code in the kernel. Computation of Δ_h is slightly more involved. Suppose a timer is set for δ time units, where a time unit corresponds to the duration of the system timer tick. If t_b is the timestamp taken before starting the timer, and t_a is the timestamp taken immediately upon entry into the timer handler, then $t_a - t_b = \delta + \Delta_h$ holds, and Δ_h can now be computed.

Note, however, that for this computation to be correct, the system must correctly count δ timer ticks from the time the timer is set. Since the time instant when the timer is started can fall anywhere within one timer tick, the timer ticks counted would not be accurate unless the timer was synchronized with the beginning of a timer tick. To achieve this synchronization we set two timers instead of one. The first timer is used to synchronize the setting of the second timer with the beginning of a timer tick; the second timer is started for δ timer ticks from within the handler for the first timer. This works because the expiry of a timer is always synchronized with the end of a timer tick, and hence the beginning of the next timer tick.

A final concern is the computation of Δ_b, the overhead to block a thread in the kernel. As such, since the call to block a thread returns only when the thread is awakened in the future, it is not possible to determine the overhead incurred to block the thread via timestamps or the timer mechanisms discussed above; the overhead would completely overlap with an outstanding timer. Therefore, we approximate Δ_b in terms of Δ_w as follows. Δ_w equals the time to remove a thread from the wait queue and enter it into the CPU run queues. Δ_b equals the time to save the thread's context, enter it into the appropriate wait queue, remove a thread from the CPU run queues and restore its context. Assuming that enqueueing a thread costs the same as dequeueing a thread, and if Δ_{cs} is the overhead to switch contexts between two threads, we have $\Delta_b \approx \Delta_w + \Delta_{cs}$.

7.5.2 Component Overheads

Table 7.3 lists the measured values of these component overheads on `tapti` and `tista`. The policing overhead (Δ_p) includes the cost of retrieving the appropriate session state variables, computing the timestamps, and checking whether a packet is compliant by comparing its expected arrival time with the current system time. The current implementation uses two 4-byte words to represent each timestamp, one for the seconds field and the other for the nanoseconds field. The policing computation can be further optimized by using only the nanoseconds field for comparing and manipulating timestamps, and handling the (rare) roll over as a special case.

The overheads of shaping non-compliant packets can be broken up into timer operations (Δ_s: set timer, Δ_h: handle timer, Δ_c: cancel timer) and thread operations (Δ_b: block thread, Δ_w: wakeup thread). For a modern machine such as `tapti`, timer operations are reasonably small relative to the base best-effort (UDP) path latency. The timer overheads correspond to the case with only one shaping timer outstanding, with perhaps a few system (kernel) timers active concurrently. For good scalability with the number of active connections, the OS timer support must scale well with the number of active timers. It is observed that the overhead of blocking threads is significant, notwithstanding faster processors, with context switching being the dominant component.

Allocation of buffers (for non-compliant packets) from the shared `mbuf` pool (overhead Δ_a^b) is significantly more expensive than allocation of buffers (for compliant packets) from the pre-allocated session-specific reserved pool (overhead Δ_a^r). Thus, the savings in buffer

Component Overheads			tapti	tista
Policing	Δ_p	compute compliance time	16.0	23.0
Buffer management	Δ_a^r	allocate reserved buffer	6.0	18.5
	Δ_f^r	free reserved buffer	15.0	33.0
	Δ_a^b	allocate best-effort buffer	18.0	35.0
Timer operations	Δ_s	set timer	7.4	14.0
	Δ_h	handle timer	7.1	30.1
	Δ_c	cancel timer	6.5	9.6
Thread operations	Δ_b	block thread	37.4	78.8
	Δ_w	wakeup thread	14.4	23.8
ARP search	Δ_{arp}	search for ARP entry	10.0	17.0

Table 7.3: Overheads of different QoS components (in μs).

allocation serves to offset some of the overheads of traffic policing and shaping. Likewise, the cost of freeing a best-effort buffer is significantly higher than the cost of freeing (i.e., returning to the session-specific buffer pool) a pre-allocated buffer. In effect, our architecture keeps the data path efficient while increasing the control path latency slightly.

7.5.3 Effective Overhead

Both policing and shaping increase the data path latencies for reserved connections over that of the best-effort data path. One would therefore expect to see higher data path latencies for reserved paths over that of the best-effort data path. However, the best-effort path differs from the reserved data path in that

- it uses the standard `mbuf` allocator for the packet, as opposed to using a pre-allocated private pool of buffers, and

- it incurs a search for the ARP (Address Resolution Protocol) entry in the ARP cache (Δ_{arp}); in the reserved path, the session handle maps directly to the appropriate virtual connection identifier, thereby eliminating this search.

As revealed by the measurements listed in Table 7.3, these differences taken together constitute a significant reduction in the data path latency for the reserved path, and partially offset the overheads due to policing and shaping.

For high-function NICs that perform traffic shaping, the only overheads incurred are those of traffic policing and buffer management, which are necessary to support the different

Connection Type	Message Size (in bytes)						
	64	128	256	512	1024	2048	4096
Best Effort	133	137	163	189	215	223	288
Reserved	159	162	172	186	217	231	298

Table 7.4: Data path latencies on tapti (in μs).

service classes such as controlled load. If $\mathcal{C}_{base}$ represents the base latency of the best-effort path, the latency seen by a compliant packet is $\approx \mathcal{C}_{base} + \Delta_p - (\Delta_a^b - \Delta_a^r) - \Delta_{arp}$, and that seen by a non-compliant packet is $\mathcal{C}_{base} + \Delta_p$. For `tapti` $\Delta_{arp} \approx 10\mu s$ (Table 7.3) for an entry that is resident in the cache. Thus, the effective increase in data path latency for a compliant packet is $\approx$ -6 μs, i.e., the data path is slightly *faster*. Similarly, the reduction in data path latency on `tista` is $\approx$ 10 μs.

Table 7.4 shows application-level latency measurements for UDP traffic on best-effort and reserved data paths for a range of message sizes. These results indicate that for 512-byte packets, compliant packets do experience a slight ($\approx$ 1.6%) decrease in latency. The same experiments also reveal that for other packet sizes, compliant packets see an increase in latency. While additional experiments are needed to investigate this further, we suspect this is primarily due to cache and OS related effects. A non-compliant packet experiences a latency increase of $\approx$ 12.5% since it incurs the policing overhead and uses best-effort buffers.

Session-level traffic shaping incurs significant additional overheads in the form of timer and thread operations. Further, session-level shaping may also incur a variable (CPU load dependent) delay in scheduling a blocked thread on the CPU (Δ_{sd}). The next section discusses the performance implications of using session-level shaping, and also outlines techniques to reduce or mask some of the shaping overheads and delays.

7.6 Performance Implications

In this section we discuss various performance implications of session-level traffic shaping and alternatives such as datalink-level shaping. We note that, in order to realize application-level QoS, the mechanisms provided in the architecture need to be integrated with QoS-sensitive CPU scheduling policies within the operating system. However, the architectural mechanisms presented and evaluated in preceding sections are needed regardless of the particular CPU scheduling policy employed.

7.6.1 Accommodating variations in the shaping latency

With session-level shaping, in the absence of other background activity a blocked thread would be allocated the CPU immediately after it is made runnable, i.e., $\Delta_{sd} \approx 0$ ignoring context switching overheads. However, the presence of other CPU-intensive background activity can introduce additional, variable delays. Before the thread can get to run to consume the buffers now available, there is

- the latency of making the thread runnable and dispatching the CPU scheduler to select a thread (the scheduler dispatch latency), and
- the latency due to competition with other runnable threads/processes for access to the CPU.

The additional delays imply that the thread in question gets blocked longer than that required as per the time to compliance (the desired shaping latency). That is, Δ_{sd} is non-zero and a function of the background load.

Before considering alternatives to "absorb" variations in the shaping latency, we note that shaping occurs only when an application generates non-compliant traffic relative to the stated traffic specification. Accordingly, the frequency of occurrence of shaping by `QOSMGR` is largely a function of the likelihood of the application exceeding its traffic specification. This in turn is related to the ability of the application to accurately "predict" its run-time communication behavior. While this is possible for relatively simple multimedia applications, such as video playback (via lookahead or on-line smoothing), it may be extremely difficult for more complicated mixed-media applications. Assuming that an application correctly estimates its run-time communication behavior, the performance degradation due to traffic shaping would not be substantial.

In other situations, the exacerbation in shaping latency may result in a QoS connection *losing credits* for access to the network, if it is unable to send compliant data even when buffers are available. Further, since the shaping latency is partly determined by the background load on the CPU, unpredictable variations in shaping latency must also be accounted for. For provision of QoS, the scheduler dispatch latency Δ_{dis} must be bounded in the worst case. If so, one can account for the extra delay of Δ_{dis}^{max} by only shaping for $\Delta_{tc} - \Delta_{dis}^{max}$ time units, assuming a worst-case scenario for scheduler dispatch; the average value of Δ_{dis} may be significantly smaller than Δ_{dis}^{max}. The second latency component, the delay waiting for CPU access, can be accounted for via measurement and adaptation. If the desired shaping delay is Δ_{tc}, and the thread actually gets delayed by Δ_{act} ($\geq \Delta_{tc}$), then it lost credits for ($\Delta_{act} - \Delta_{tc}$) time units. A lost credit implies that the thread could have utilized the available buffers for outgoing packets and utilized its share of link bandwidth, if it had been able to run as per the desired shaping delay.

Since guarantees are provided on the long-term rate as well, one can envision a scheme where the shaping delay applied adapts according to fluctuations in the background load. Thus, if the thread got delayed by an extra delay $\Delta_{ext}(t) = \Delta_{act}(t) - \Delta_{tc}(t)$ at shaping instant t, it is only blocked for $\max(\Delta_{tc}(t') - \Delta_{ext}(t), 0)$ time units at the next shaping instant t'. Our policing computation keeps track of these lost credits and accounts for any extra shaping delay by subsequently treating more packets as compliant. However, adapting to shaping latency in this fashion has implications for the buffer management performed by `QOSMGR`; a QoS connection may need to over-provision buffers to allow longer bursts of packets through compared to the connection's traffic specification. Coupled with an appropriate buffer management policy, the variations in the scheduling delay can also be accommodated.

As mentioned in Section 3.3, for legacy NICs with appropriate queueing and scheduling supported by the NDD, datalink-level shaping serves as an alternative to session-level shaping [13]. With datalink-level scheduling, the packet scheduler operates in a non work-conserving fashion, injecting compliant packets into the network either in the context of an executing application thread, or on receipt of a transmission-complete interrupt notification for the previous packet transmission. Since the packet scheduler is mostly invoked in interrupt context, the coupling between CPU scheduling and transmission of compliant packets is greatly reduced. Since threads are blocked only due to buffer shortages, the overheads incurred with datalink-level shaping are lower than those incurred in session-level shaping. While saving some thread-related overheads (block and wakeup, context switch), datalink-level shaping still incurs the overheads of timer operations.

Note that due to the need to support different service classes, traffic policing and buffer management must always be performed by `QOSMGR`, thereby necessitating the need for thread blocking and wakeup mechanisms regardless of the shaping mechanism employed. One significant drawback of datalink-level shaping is that it does not provide efficient mechanisms to give immediate notification to applications violating their traffic specification. Such mechanisms allow adaptive applications to react to such notifications by invoking appropriate traffic adaptation functions. Further, these mechanisms are also useful in controlling the allocation of CPU cycles to individual connections, as discussed in Section 7.6.2. Session-level shaping provides such efficient and immediate feedback to applications on a per-packet basis, allowing misbehaving applications to be notified or blocked at the earliest.

The scheduling delay associated with session-level shaping can be largely eliminated (or at least made predictable) by employing *QoS-sensitive* CPU scheduling policies, as discussed next.

7.6.2 QoS-Sensitive CPU Scheduling

Recently several QoS-sensitive scheduling policies such as stride scheduling [179], proportional share [165], and hierarchical scheduling [71] have been proposed. These policies ensure that the CPU is allocated to individual threads in the order of their associated QoS requirements and at the granularity of a certain quantum, i.e., each thread executes at the most for a quantum each time it is selected to run. Similarly, real-time operating systems typically provide support for periodic fixed-priority and dynamic priority CPU scheduling, such as rate monotonic (RM) and earliest deadline first (EDF), respectively [111].

While such QoS-sensitive CPU scheduling policies suffice for computation activities, the granularity of the CPU scheduling quantum may be too coarse for accurate scheduling and traffic shaping of application data exporter threads. The execution priority of a data exporter thread must be derived from the application's requested QoS and communication behavior on the corresponding connection. Consider a scenario where a higher-priority thread from one application waits for a lower-priority thread from another application to relinquish the CPU at the end of a quantum. However, the above situation implies that in the worst-case, the higher-priority thread must wait for an entire quantum before continuing to process and transmit compliant packets. The corresponding connection therefore loses credits and may even experience QoS violations. Further, since fragmentation is performed by the data exporter in the application's address space, the lower-priority thread may continue to send non-compliant network data, consuming shared best-effort buffers unfairly and even starving best-effort traffic.

This suggests that the data exporter threads, i.e., those performing fragmentation and other processing of network data, be scheduled for execution using fine-grain preemption.

Such fine-grain multiplexing of communication threads has been adopted and analyzed in [68, 69] for RM scheduling, and in [116, 117] for EDF scheduling, respectively. The architecture described in [116], which was presented in Chapter 3 decouples protocol processing priority from application priority, deriving the former from the traffic and QoS specification on each connection. Accurate traffic shaping is realized via EDF scheduling of protocol processing threads.

The data generator (i.e., computation) threads may still be scheduled via the quantum-based CPU scheduling policies mentioned above. Some strategies for integrated scheduling of data generator and data exporter threads are outlined in Chapter 8. The above scenario highlights another advantage with session-level shaping: a mechanism to introduce a scheduling point by blocking application threads until the traffic is compliant. This is not possible with datalink-level shaping, in which packet generation and hence buffer consumption becomes work-conserving under the constraint of availability of buffers.

7.7 Summary and Future Work

In this paper we have studied the performance impact of supporting QoS communication in TCP/IP protocol stacks. This study was conducted on a prototype implementation of a new QoS architecture for an RSVP-based integrated services Internet. We believe this is the first study that quantifies the performance impact of QoS support in TCP/IP protocol stacks. It also complements other recent efforts to understand and improve protocol processing and data transfer performance for best-effort traffic in high-speed networks. The significance of our study stems from the expected large-scale deployment of the new service model being defined by the IETF for an integrated services Internet.

Our main conclusions can be summarized as follows. Of the different QoS overheads considered, traffic shaping presents the most challenges due to its interaction with the OS CPU scheduler. Traffic policing and shaping overheads are partially offset by savings due to pre-allocation of per-session buffers. Further savings can be obtained for ATM networks due to a faster path through the network interface layer. While policing overheads can be further optimized in a straightforward manner, reducing traffic shaping overheads would require improvements in OS timer and thread operations. Potential load-dependent variations in the actual shaping latency can be accommodated via appropriate adaptation and buffer management, or largely eliminated via integration with QoS-sensitive CPU scheduling policies.

Our work complements recent work on QoS-sensitive CPU scheduling of applications [71, 165, 179] and protocol processing [68, 69, 116, 117] at end hosts. While these efforts focus on CPU scheduling, our primary focus is on the QoS support architecture exported to sockets based applications. With appropriate CPU scheduling support, our QoS architecture enables new and legacy applications to utilize end-to-end QoS on communication.

For future work, this study can be extended to explore the performance impact in the presence of multiple simultaneous QoS connections. This might shed additional light on the scalability of OS timer and thread operations, including synchronization and mutual exclusion overheads. We are also interested in examining the issues involved in QoS overheads imposed on the data reception path in TCP/IP protocol stacks.

Acknowledgement

We gratefully acknowledge the contributions of Tsipora Barzilai, John Chu and Isabella Chang at the IBM T. J. Watson Research Center, and Satya Sharma, Steve Wise, William Hymas, and Dan Badt at IBM Austin.

CHAPTER 8

INTEGRATION WITH HOST OPERATING SYSTEM

In this chapter we identify the issues involved in, and outline strategies for, integrating a QoS-sensitive communication subsystem within a QoS-sensitive operating system for application-level QoS guarantees.

We believe that application-level QoS guarantees can be realized by appropriately integrating the architecture and admission control extensions proposed in this dissertation with QoS-sensitive application scheduling policies. Furthermore, the architectural mechanisms, admission control extensions, and guaranteed-QoS service developed in Chapters 3, 4, and 5 respectively, remain applicable regardless of the protocol processing architecture and location, i.e., in the kernel or at user level. Many aspects of the proposed architecture, such as cooperative preemption and exploitation of overlap between CPU processing and link transmission/reception can and must be preserved when integrating with QoS-sensitive application scheduling policies. Our approach to QoS-sensitive communication subsystem design greatly facilitates such an integration.

We first compare and contrast the different protocol processing architectures proposed in the literature, primarily from the perspective of provision of QoS guarantees on communication. Subsequently we discuss some approaches towards realization of application-level QoS guarantees.

8.1 Protocol Processing Architectures

In this section we discuss how our proposed architecture compares to other protocol processing architectures, and argue that extending these architectures to support QoS guarantees

Type	Architecture	Protocol processing
I	Traditional in-kernel (monolithic)	system call or interrupt driven
II	User-level trusted server (microkernel)	IPC driven
III	User-level application libraries	system call and upcall driven
IV	Lazy receiver processing	system call or interrupt driven; process when application receives

Table 8.1: Protocol processing architectures.

would necessitate features similar in nature to the ones proposed in our architecture. Furthermore, regardless of the protocol processing architecture employed, realization of QoS guarantees in practice necessitates admission control extensions along the lines of the ones considered in this dissertation.

Architectural Overview

Table 8.1 lists the protocol processing architectures proposed in the literature.

Type I: corresponds to traditional in-kernel communication subsystem present in BSD Unix [108] and its derivatives. Protocol processing in this architecture is performed at system call time or subsequently in response to a network device or timer interrupt. The architectural enhancement described in Chapter 7 were applied to a Type I architecture. As mentioned, these enhancements are geared towards fragmentation for large data transfers being performed in the application's address space by one or more data exporter (or communication) threads.

Type II: corresponds to a contemporary microkernel configuration in which a trusted user-level server implements the communication subsystem, as exemplified by the guaranteed-QoS communication service described in Chapter 5. Protocol processing is thus triggered via IPC between the application and the trusted server on the one hand, and between the server and the kernel on the other.

Type III: also corresponds to a user-level protocol processing configuration, on a microkernel operating system [112, 168] or otherwise [55], with the difference that data transmission and reception is performed by threads executing in communication subsystem libraries linked with the application; control operations such as setting up connections are performed by the user-level trusted server or the kernel. Protocol processing for transmission is performed at system call time, while protocol processing for reception is triggered via an upcall into the application's address space.

Type IV: Lazy receiver processing (LRP) [53] is similar to the traditional in-kernel configuration, except that the processing of receiving packets is performed mostly in the context of, and at the priority of, the receiving application when it actually receives the data. Under some scenarios for TCP, a dedicated kernel process performs asynchronous protocol processing for incoming TCP packets. Arriving packets are directly demultiplexed onto the corresponding socket queue and excess traffic is discarded early.

Provision of QoS Guarantees on Communication

For provision of QoS guarantees on communication, each of the above architectures requires appropriate support to multiplex communication resources between connections. The nature of such support depends on the location of the communication subsystem, i.e., in the kernel, in a user-level server, or in application libraries. The location of the communication subsystem is important because it partly determines the manner in which host communication resources are shared between connections belonging to different applications as well as connections associated with the same application.

If the communication subsystem resides in a single address space, such as in the kernel (Type I) or in a trusted server (Type II), centralized control can be exercised over allocation of communication resources to connections, irrespective of the associated application. Our architecture is naturally geared towards such a structure, which also facilitates a clean separation between the computation subsystem and the communication subsystem. In our architecture and analysis we have not made any specific assumptions about the location of the protocol stack, which could reside in the kernel or in user space.

On the other hand, if the communication subsystem is split between address spaces, e.g., between the application and the kernel (Type III), centralized management of communication resources becomes difficult, if not impossible. Note that such a split architecture can also be realized between the application and the kernel (Type I) if the communication subsystem in the kernel does not provide fragmentation services, or application libraries are utilized to extend the functionality of the communication protocol stack.

Two observations can be made regarding provision of QoS guarantees in this case. One, the kernel must still provide architectural support to manage shared communication resources, such as CPU cycles, communication buffers, and link bandwidth, across different applications. Two, since a substantial amount of protocol processing is performed via threads in application-level libraries, appropriate architectural mechanisms are required to multiplex these threads on the CPU as per the QoS requirements and traffic specification on the associated connections. Our architecture can be directly utilized for such fine-grain multiplexing of CPU cycles allocated to the application; cooperative preemption can be used safely since all threads execute in the same address space and protection domain. This is especially true for an application that is a source or sink for multiple QoS connections.

Similarly, the proposed admission control extensions are also applicable to such architectures. This is because since protocol processing is performed, and data transmission initiated, during the application's currently allocated quantum, the overlap between protocol processing and link transmission still exists and can be exploited. Further, on data reception, the communication threads in the application library are immediately scheduled for execution [107]; thus, the overlap between data reception and protocol processing can be exploited as well with appropriate scheduling of these communication threads. With user-level protocol processing via application-level libraries, however, allocation of communication resources is coupled closely with allocation of computation resources, and as such the admission control must be more comprehensive.

Similar observations apply to a Type IV architecture (i.e., LRP) for data transmission and data reception, with data reception presenting additional challenges. Delaying the processing of received data till the application actually does a receive has significant implications for provision of QoS guarantees on communication. Not only does this tie protocol processing priority to the priority of the application, it is also unable to exploit the overlap between link reception and CPU processing. In contrast, our architecture utilizes early demultiplexing to immediately schedule protocol processing, deriving protocol processing priority from the QoS and traffic specification of the application while exploiting CPU and link processing overlap. Even with LRP, the architectural mechanisms provided by our architecture would be required to determine the order in which the application processes data received on multiple sockets.

Note that a Type III protocol processing architecture can be used in conjunction with processor capacity reserves [123] to realize predictable protocol processing [107]. Unlike our approach, the approach outlined in [107] does not derive the protocol processing priority from a connection's QoS and traffic specifications, nor does it exploit the overlap between protocol processing and link transmission/reception. Moreover, only a *single* communication library thread is associated with an application; data arriving on multiple connections associated with the application are all processed by this thread. Besides processing incoming packets strictly in FIFO order, this makes it harder to perform service differentiation, i.e., handle packets from different connections according to the respective QoS guarantees.

8.2 Realizing Application-Level QoS Guarantees

In the previous section we argued that for provision of QoS guarantees on communication, our architecture and admission control extensions are applicable to a variety of protocol processing architectures. In this section we discuss integration of the architecture and extensions with QoS-sensitive application scheduling policies for realization of application-level QoS guarantees.

We believe that an integration of a QoS-sensitive communication subsystem within a QoS-sensitive operating system should satisfy the following requirements:

- The protocol processing priority of a connection should be decoupled from execution priority of the associated application; instead, it should be derived from the QoS requirements, traffic specification, and observed communication behavior of the connection.
- Mechanisms such as cooperative preemption should be employed to multiplex CPU cycles for protocol processing between connections.
- It should be possible to exploit the overlap between CPU protocol processing and link transmission/reception, as this maximizes the number of connections that can be admitted for service.

We outline next one possible approach towards such an integration and discuss the issues that arise in that context.

Figure 8.1 illustrates one possible integration scenario. As depicted, we distinguish between the *computation subsystem* and the *communication subsystem*. The communication subsystem comprises all threads that participate in transmitting data and processing received data from the network. The computation subsystem, on the other hand, comprises all application processes and threads that perform activities other than communication

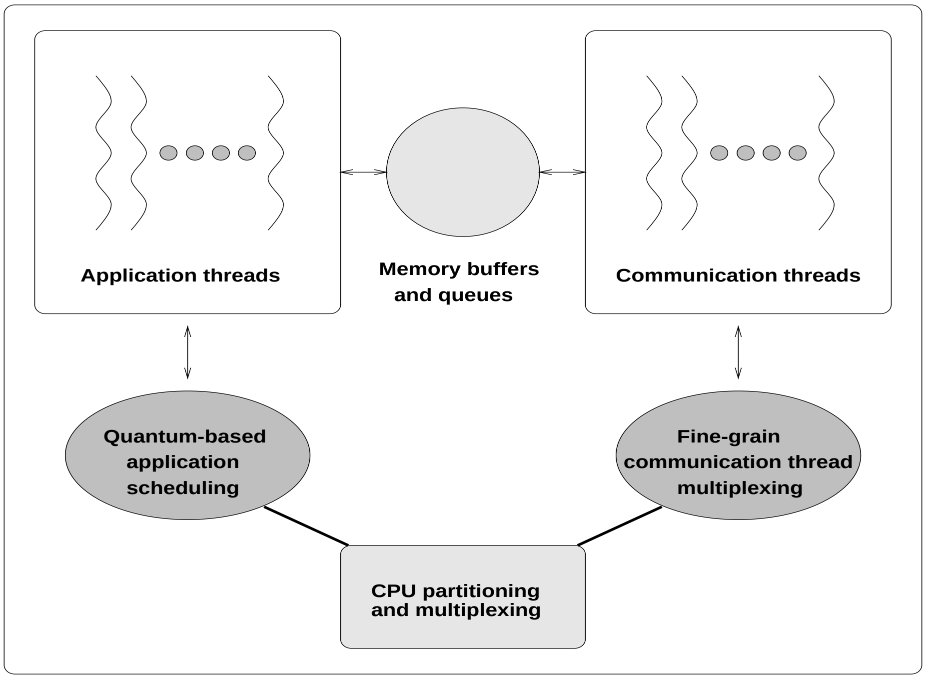


Figure 8.1: Integrated QoS-sensitive computation and communication subsystems.

processing. The two subsystems exchange network data through appropriate buffers and queues in memory. The available CPU resources are shared between the two subsystems via appropriate CPU partitioning and multiplexing, as discussed later.

We assume that each communication thread is associated with a single connection. As we have demonstrated, associating a communication thread with each application connection provides a natural mechanism to allocate CPU resources to that connection. Per-connection threads also permit traffic enforcement and allow traffic flow on the connection to be controlled via appropriate CPU scheduling policies. While the communication threads are associated with the communication subsystem, they need not be associated with a single address space and protection domain. As per the discussion in Section 8.1, these communication threads could reside in the kernel, in a trusted user-level server, or in application libraries.

Note that the key requirement is that the communication threads be *associated* with the communication subsystem. The exact location of communication threads may affect data transfer performance, but does not matter as long as the threads are scheduled by the communication subsystem. If all communication threads reside in a user-level server, then these threads are by default associated with the server protection domain. Similarly, if all communication threads reside in the kernel, then these kernel threads are by default associated with the kernel protection domain. Thus, we are effectively suggesting that regardless of their default protection domain, communication threads be associated with the *communication resource management domain*.

With the above framework, all communication threads are multiplexed on the CPU under control of the communication subsystem. The communication subsystem in turn must share CPU resources with the computation subsystem. For QoS guarantees, the communication subsystem must be allocated CPU capacity so as to limit (or bound) the interference from the computation subsystem; this capacity in turn determines the realization of application-level QoS guarantees on communication. There are two approaches to capacity allocation between the two subsystems.

In one approach, a portion of the host processing capacity can be reserved for the entire communication subsystem via capacity reserves [123]. Note that the reserves model proposed in [123] associates reserves with individual threads and allows multiple threads to subscribe to the same reserve. However, as such this model may need to be extended to support a capacity reserve for the entire communication subsystem; for a server-based incarnation of the communication subsystem, this would correspond to a reserve for the server's execution.

A similar approach that partitions the CPU to support co-resident general-purpose and real-time operating systems has been proposed and analyzed in [19]. In this approach, a certain capacity is set aside for execution of the real-time operating system and the remaining capacity is allocated to the general-purpose operating system. Schedulability tests for real-time tasks are suitably modified to consider only the available real-time capacity. This scheme can be extended to partition the CPU between the computation and communication subsystems by applying suitable modifications to the admission control procedure (`D_order`). Instead of just considering the wait times due to higher and lower priority channels, `D_order` must now consider the "wait time" due to the computation subsystem.

Another approach is to realize the entire communication subsystem as a class with an appropriate share in the recently proposed QoS-sensitive scheduling policies [71, 139, 165, 179]. These policies ensure that threads execute on the CPU in the order of their associated

QoS requirements and at the granularity of a certain quantum. Each thread executes for at most a quantum each time it is selected to run. An appropriate choice of a class and/or share would guarantee the necessary CPU capacity to the communication subsystem.

The capacity thus allocated to the communication subsystem can be further allocated to individual connection threads as per the mechanisms described in this dissertation. Our analysis and admission control extensions are directly applicable as long as `D_order` (or any other admission control procedure employed) accounts for the portion of CPU capacity allocated to the computation subsystem. Note that the CPU and link processing overlap captured in the message service time computations of Chapter 4 continues to apply even when the communication subsystem is allocated a fraction of the CPU capacity. This implies that in addition to cooperative preemption to other communication threads, a communication thread is also preempted by "service outages" during which one or more application threads occupy the CPU. However, these outages are guaranteed to be bounded by the CPU partitioning and multiplexing mechanism (Figure 8.1), and are accounted for by `D_order` when admitting QoS connections.

8.3 Summary

The above scenario only outlines one possible way to integrate a QoS-sensitive communication subsystem within a QoS-sensitive operating system. Numerous architectural and implementation challenges must be overcome when realizing this integration in practice. While we have highlighted sharing of CPU capacity between the two subsystems, other resources such as memory buffers and queues must also be sized and managed appropriately to facilitate such an integration while ensuring high performance.

Further exploration of the challenges involved in integrated resource management for application-level QoS guarantees is beyond the scope of this dissertation. As we have argued, our approach to QoS-sensitive communication subsystem design greatly facilitates this integration. The issues highlighted and practical design tradeoffs considered in this dissertation represented an important step towards practical realization of application-level QoS guarantees on communication.

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

This dissertation focuses on provision of QoS guarantees at the interface between the applications and the network, namely, the host communication subsystem. In particular, we have examined various issues involved in structuring host communication software to provide per-connection QoS guarantees. Our research approach centers around the implementation of a given service discipline, in our case real-time channels, on real computer systems. Accordingly, this dissertation makes several key contributions towards the practical realization of QoS-sensitive communication subsystems. All the architectural components, mechanisms, and extensions proposed in this dissertation for QoS-sensitive communication subsystem design have been prototyped and evaluated experimentally. We believe that the contributions made by this dissertation advance the state of the art in the provision of host resources for application-level QoS guarantees.

In the next section we summarize the primary contributions made by this dissertation. We conclude with a discussion of the various avenues possible for future research, and highlight some of the key problems that must be addressed for each.

9.1 Primary Contributions

In this dissertation we have made the following primary contributions:

- We have designed, implemented, and evaluated a QoS-sensitive communication subsystem architecture that manages communication resources on the basis of three design principles: maintenance of per-connection QoS guarantees, overload protection via per-connection traffic enforcement, and fairness to best-effort traffic. This architecture strives to maximize useful resource capacity by reducing preemption and other implementation overheads, while isolating each QoS connection to the maximal extent possible. It also supports a number of policies for overload protection, the best policy being largely application-dependent. While designed primarily for provision of deterministic QoS guarantees, it can be readily extended to support more relaxed forms of QoS guarantees and adaptive applications.

- In the context of the above architecture, we have proposed and implemented admission control extensions to bridge the gap between the theory and practice of communication resource management. This gap arises because theoretical resource management policies are typically formulated using idealized resource models; the assumptions thus made may get violated in practice due to the non-ideal performance characteristics of real hardware and software components, or the cost-performance tradeoffs often encountered during implementation. We extend such policies by identifying and accounting for a number of factors that impact communication subsystem performance significantly at sending and receiving hosts. Without these extensions, it would be impossible to use theoritical resource management policies in practice. While developed specifically for real-time channels, these extensions are general in nature and can be similarly developed for other service disciplines.

- Much additional insight can be gained in the complexity of QoS-sensitive communication subsystem design by exploring and resolving the challenges posed by contemporary operating systems. Accordingly, we have realized a complete guaranteed-QoS communication service on a microkernel operating system. This service further enhances the architecture and extensions mentioned above to realize a new integrated service architecture comprising a QoS-aware API, signalling and resource reservation services, and generic support for QoS-sensitive data transfer. We further enhanced the resource management policies to account for additional overheads imposed by the API and the implementation environment.

- One of the primary difficulties in QoS-sensitive communication subsystem design is accurate profiling and parameterization of the communication subsystem. We address this concern by proposing self-parameterizing protocol stacks to enhance the portability of QoS-sensitive communication subsystems. Self-parameterizing protocol stacks perform on-line profiling to construct a database of system parameters that form the abstraction of the underlying communication subsystem and platform. We believe that extending traditional protocol stacks with efficient profiling mechanisms and system parameter databases is a promising way to realize portable QoS-sensitive communication subsystems.

• Internet hosts represent an important class of end systems that will increasingly utilize the QoS support offered by an integrated services Internet. Of immediate relevance to these hosts is the enhancement of the existing sockets based communication subsystem to request and obtain QoS across the Internet. Any such support within the communication subsystem must not only maintain the performance of the default best-effort path, it must also not incur excessive overhead in order for integrated services to be widely utilized. Using the RSVP-based QoS architecture developed in collaboration with researchers at the IBM T. J. Watson Research Center, we explored and assessed the performance impact of QoS support in TCP/IP protocol stacks.

9.2 Future Research Avenues

Our work can be extended in several interesting directions which are briefly discussed below.

◦ ***Integration of QoS-sensitive subsystems:*** While Chapter 8 presented strategies for integrating a QoS-sensitive communication subsystem with QoS-sensitive application scheduling, many issues remain in the actual realization of such an integration. Several subsystems within the host operating system, such as the file system, I/O devices, etc., must be integrated appropriately and efficiently for true end-to-end application QoS provisioning. For example, contrary to current trends in high-performance communication to transfer (transmit and receive) data as fast as possible, a QoS-sensitive communication subsystem must transfer data *only as fast as appropriate* as determined by the connection's traffic contract and desired QoS.

◦ ***Interplay between privacy, authentication and QoS:*** In this dissertation we have focused exclusively on provision of QoS guarantees, ignoring issues in privacy and authentication. However, support for privacy and authentication must be integrated with provision of QoS guarantees in order for multimedia communication across the Internet to be a common feature. This is because of the innate human need to have privacy in all forms of meaningful communication in which she is an active participant, and the desire to validate the identity and intentions of the other participants. One of the concerns here is the impact of encryption and associated processing requirements on communication subsystem performance at end hosts, and another is the ability of network routers to identify and classify flows.

◦ ***Communication support for Web-based multimedia applications:*** There are two related aspects to communication support for Web-based multimedia applications: the paradigms adopted for Web-based Internet "computing", and the elements of that computation, i.e., multimedia data (such as audio and video) and its associated temporal constraints. Examples of emerging paradigms include interactive multi-way communication via virtual reality engines (exemplified by Internet gaming) and video computation. The nature of QoS support in communication subsystems at Web clients and servers that would suffice for such Web-based applications is an open question.

◦ ***Adaptive resource management:*** As mentioned in Chapter 2, a large class of multimedia and other soft real-time applications are adaptive in nature, i.e., they can be designed to adapt to fluctuations in the delivered QoS on communication, especially if the set of communicating participants or the traffic load changes dynamically. For these applications, reserving resources based on a traffic specification and desired QoS may not be the appropriate service model. It would be interesting to explore algorithms and policies for resource management that can intelligently adapt to traffic fluctuations transparently while still providing acceptable QoS to individual connections.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] M. B. Abbott and L. L. Peterson, "Increasing network throughput by integrating protocol layers," *IEEE/ACM Trans. Networking*, vol. 5, no. 1, pp. 600–610, October 1993.

[2] T. Abdelzaher, E. Atkins, and K. Shin, "QoS negotiation in real-time systems and its application to automated flight control," in *Proc. Real-Time Technology and Applications Symposium*, pp. 228–238, June 1997.

[3] R. Ahuja, S. Keshav, and H. Saran, "Design, implementation, and performance of a native mode ATM transport layer," in *Proc. IEEE INFOCOM*, pp. 206–214, March 1996.

[4] D. P. Anderson, S. Y. Tzou, R. Wahbe, R. Govindan, and M. Andrews, "Support for continuous media in the DASH system," in *Proc. Int'l Conf. on Distributed Computing Systems*, pp. 54–61, May 1990.

[5] D. P. Anderson, "Metascheduling for continuous media," *ACM Trans. Computer Systems*, vol. 11, no. 3, pp. 226–252, August 1993.

[6] D. P. Anderson, L. Delgrossi, and R. G. Herrtwich, "Structure and scheduling in real-time protocol implementations," Technical Report TR–90–021, International Computer Science Institute, Berkeley, June 1990.

[7] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, "Scheduler activations: Effective kernel support for the user-level management of parallelism," *ACM Trans. Computer Systems*, vol. 10, no. 1, pp. 53–79, February 1992.

[8] C. M. Aras, J. F. Kurose, D. S. Reeves, and H. Schulzrinne, "Real-time communication in packet-switched networks," *Proceedings of the IEEE*, vol. 82, no. 1, pp. 122–139, January 1994.

[9] ARMADA Homepage. http://www.eecs.umich.edu/RTCL/armada/.

[10] E. A. Arnould, F. J. Bitz, E. C. Cooper, H. T. Kung, R. D. Sansom, and P. A. Steenkiste, "The design of Nectar: A network backplane for heterogeneous multicomputers," in *Proc. Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 205–216, April 1989.

[11] M. L. Bailey, B. Gopal, M. A. Pagels, L. L. Peterson, and P. Sarkar, "PATHFINDER: A pattern-based packet classifier," in *Proc. of ACM SIGCOMM*, pp. 115–123, August 1994.

[12] A. Banerjea, D. Ferrari, B. Mah, M. Moran, D. C. Verma, and H. Zhang, "The Tenet real-time protocol suite: Design, implementation, and experiences," *IEEE/ACM Trans. Networking*, vol. 4, no. 1, pp. 1–11, February 1996.

[13] T. Barzilai, D. Kandlur, A. Mehra, D. Saha, and S. Wise, "Design and implementation of an RSVP-based quality of service architecture for integrated services Internet," in *Proc. Int'l Conf. on Distributed Computing Systems*, May 1997.

[14] M. Bjorkman and P. Gunningberg, "Locking effects in multiprocessor implementations of protocols," in *Proc. of ACM SIGCOMM*, pp. 74–83, September 1993.

[15] D. L. Black, R. D. Smith, S. J. Sears, and R. W. Dean, "FLIPC: A low latency messaging system for distributed real-time environments," in *Proc. USENIX Winter Conference*, January 1996.

[16] T. Blackwell, "Speeding up protocols for small messages," in *Proc. of ACM SIGCOMM*, pp. 85–95, October 1996.

[17] G. Blair, A. Campbell, G. Coulson, F. Garcia, D. Hutchison, A. Scott, and D. Shepherd, "A network interface unit to support continuous media," *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 2, pp. 264–275, February 1993.

[18] M. A. Blumrich, C. Dubnicki, E. W. Felten, and K. Li, "Protected, user-level DMA for the SHRIMP network interface," in *Proc. International Symposium on High-Performance Computer Architecture*, February 1996.

[19] G. Bollella and K. Jeffay, "Supporting co-resident operating systems," in *Proc. Real-Time Technology and Applications Symposium*, pp. 4–14, June 1995.

[20] M. Borden, E. Crawley, B. Davie, and S. Batsell, "Integration of real-time services in an IP-ATM network architecture," *Request for Comments RFC 1821*, August 1995. Bay Networks, Bellcore, NRL.

[21] A. Braccini, A. D. Bimbo, and E. Vicario, "Interprocess communication dependency on network load," *IEEE Trans. Software Engineering*, vol. 17, no. 4, pp. 357–369, April 1991.

[22] R. Braden, D. Clark, and S. Shenker, "Integrated services in the Internet architecture: An overview," *Request for Comments RFC 1633*, July 1994. Xerox PARC.

[23] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin, "Resource ReSerVation Protocol (RSVP) - version 1 functional specification," *Internet Draft draft-ietf-rsvp-spec-12.txt*, May 1996. ISI/PARC/USC.

[24] A. Brown and M. Seltzer, "Operating system benchmarking in the wake of lmbench: A case study of the performance of NetBSD on the Intel x86 architecture," in *Proc. of ACM SIGMETRICS*, pp. 214–224, June 1997.

[25] J. C. Brustoloni and P. Steenkiste, "Effects of buffering semantics on I/O performance," in *Proc. USENIX Symp. on Operating Systems Design and Implementation*, pp. 277–291, October 1996.

[26] J. C. Brustoloni and P. Steenkiste, "Copy emulation in checksummed, multiple-packet communication," in *Proc. IEEE INFOCOM*, November 1997.

[27] J. C. Brustoloni and P. Steenkiste, "Evaluation of data passing and scheduling avoidance," in *Proc. Intl. Workshop on Network and Operating System Support for Digital Audio and Video*, May 1997.

[28] V. Buch, T. von Eicken, A. Basu, and W. Vogels, "U-Net: A user-level network interface for parallel and distributed computing," in *Proc. ACM Symp. on Operating Systems Principles*, pp. 40–53, December 1995.

[29] A. Burns, K. Tindell, and A. Wellings, "Effective analysis for engineering real-time fixed priority schedulers," *IEEE Trans. Software Engineering*, vol. 21, no. 5, pp. 475–480, May 1995.

[30] G. Buzzard, D. Jacobson, M. Mackey, S. Marovich, and J. Wilkes, "An implementation of the Hamlyn sender-managed interface architecture," in *Proc. USENIX Symp. on Operating Systems Design and Implementation*, pp. 245–259, October 1996.

[31] L. F. Cabrera, E. Hunter, M. J. Karels, and D. A. Mosher, "User-process communication performance in networks of computers," *IEEE Trans. Software Engineering*, vol. 14, no. 1, pp. 38–53, January 1988.

[32] A. T. Campbell, C. Aurrecoechea, and L. Hauw, "A review of QoS architectures," *Multimedia Systems Journal*, 1996.

[33] A. T. Campbell and G. Coulson, "QoS adaptive transports: Delivering scalable media to the desktop," *IEEE Network Magazine*, pp. 18–27, March/April 1997.

[34] A. T. Campbell, G. Coulson, and D. Hutchison, "A quality of service architecture," *Computer Communication Review*, April 1994.

[35] C. E. Catlett, "In Search of Gigabit Applications," *IEEE Communication Magazine*, pp. 42–51, April 1992.

[36] D. R. Cheriton and C. L. Williamson, "VMTP as the transport layer for high-performance distributed systems," *IEEE Communication Magazine*, pp. 37–44, June 1989.

[37] G. Chesson, "XTP/PE overview," in *Proc. Conference on Local Computer Networks*, October 1988.

[38] D. D. Clark, "The structuring of systems using upcalls," in *Proc. ACM Symp. on Operating Systems Principles*, pp. 171–180, 1985.

[39] D. D. Clark, S. Shenker, and L. Zhang, "Supporting real-time applications in an integrated services packet network: Architecture and mechanism," in *Proc. of ACM SIGCOMM*, pp. 14–26, August 1992.

[40] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen, "An analysis of TCP processing overhead," *IEEE Trans. Communications*, pp. 23–29, June 1989.

[41] D. D. Clark and D. L.Tennenhouse, "Architectural considerations for a new generation of communication protocols," in *Proc. of ACM SIGCOMM*, pp. 200–208, September 1990.

[42] G. Coulson, G. S. Blair, and P. Robin, "Micro-kernel support for continuous media in distributed systems," *Computer Networks and ISDN Systems*, vol. 26, pp. 1323–1341, 1994.

[43] G. Coulson, A. Campbell, P. Robin, G. S. Blair, M. Papathomous, and D. Shepherd, "The design of a QoS-controlled ATM-based communications system in Chorus," *IEEE Journal on Selected Areas in Communications*, vol. 13, no. 4, pp. 686–699, May 1995.

[44] R. L. Cruz, *A Calculus for Network Delay and a Note on Topologies of Interconnection Networks*, PhD thesis, University of Illinois at Urbana-Champaign, July 1987. available as technical report UILU–ENG–87–2246.

[45] H. Custer, *Inside Windows NT*, Microsoft Press, One Microsoft Way, Redmond, Washington 98052-6399, 1993.

[46] C. Dalton, G. Watson, D. Banks, C. Calamvokis, A. Edwards, and J. Lumley, "Afterburner," *IEEE Network Magazine*, pp. 36–43, July 1993.

[47] P. B. Danzig, "An analytical model of operating system protocol processing including effects of multiprogramming," in *Proc. of ACM SIGMETRICS*, pp. 11–20, May 1991.

[48] B. Davie, "The architecture and implementation of a high-speed host interface," *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 2, pp. 228–239, February 1993.

[49] L. Delgrossi and L. Berger, "Internet stream protocol version 2 (ST-2) protocol specification - version ST2+," *Request for Comments RFC 1819*, August 1995. ST2 Working Group.

[50] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorthm," *Proc. of ACM SIGCOMM*, pp. 3–12, September 1989.

[51] J. Dolter, S. Daniel, A. Mehra, J. Rexford, W. Feng, and K. Shin, "SPIDER: Flexible and efficient communication support for point-to-point distributed systems," in *Proc. Int'l Conf. on Distributed Computing Systems*, pp. 574–580, June 1994.

[52] P. Druschel, M. B. Abbott, M. Pagels, and L. L. Peterson, "Network subsystem design: A case for an integrated data path," *IEEE Network Magazine*, pp. 8–17, July 1993.

[53] P. Druschel and G. Banga, "Lazy receiver processing (LRP): A network subsystem architecture for server systems," in *Proc. USENIX Symp. on Operating Systems Design and Implementation*, pp. 261–275, October 1996.

[54] P. Druschel and L. L. Peterson, "Fbufs: A high-bandwidth cross-domain transfer facility," in *Proc. ACM Symp. on Operating Systems Principles*, pp. 189–202, December 1993.

[55] P. Druschel, L. L. Peterson, and B. S. Davie, "Experiences with a high-speed network adaptor: A software perspective," in *Proc. of ACM SIGCOMM*, pp. 2–13, August 1994.

[56] C. Dubnicki, L. Iftode, E. W. Felten, and K. Li, "Software support for virtual memory-mapped communication," in *Proc. International Conference on Parallel Processing*, April 1996.

[57] A. Edwards, G. Watson, J. Lumley, D. Banks, C. Clamvokis, and C. Dalton, "User-space protocols deliver high performance to applications on a low-cost Gb/s LAN," in *Proc. of ACM SIGCOMM*, pp. 14–24, London, UK, August 1994.

[58] Edwin F. Menze III and F. Travostino, *The CORDS Book*, OSF Research Institute, September 1996.

[59] D. Engler and M. F. Kaashoek, "DPF: Fast, flexible message demultiplexing using dynamic code generation," in *Proc. of ACM SIGCOMM*, pp. 53–59, October 1996.

[60] K. Fall and J. Pasquale, "Exploiting in-kernel data paths to improve I/O throughput and CPU availability," in *Proc. USENIX Winter Conference*, January 1993.

[61] D. C. Feldmeier, "A survey of high performance protocol implementation techniques," in *High Performance Networks: Technology and Protocols*, A. N. Tantawy, editor, pp. 29–50, Kluwer Academic Publishers, 1994.

[62] D. Ferrari, "Client requirements for real-time communication services," *IEEE Communication Magazine*, pp. 65–72, November 1990.

[63] D. Ferrari and D. C. Verma, "A scheme for real-time channel establishment in wide-area networks," *IEEE Journal on Selected Areas in Communications*, vol. 8, no. 3, pp. 368–379, April 1990.

[64] M. E. Fiuczynski and B. N. Bershad, "An extensible protocol architecture for application-specific networking," in *Proc. USENIX Winter Conference*, January 1996.

[65] S. Floyd and V. Jacobson, "Link-sharing and resource management models for packet networks," *IEEE/ACM Trans. Networking*, vol. 3, no. 4, pp. 365–386, August 1995.

[66] B. Ford and S. Susarla, "CPU inheritance scheduling," in *Proc. USENIX Symp. on Operating Systems Design and Implementation*, pp. 91–105, October 1996.

[67] A. Garg, "Parallel STREAMS: A multiprocessor implementation," in *Winter 1990 USENIX Conference*, pp. 163–176, January 1990.

[68] R. Gopalakrishnan and G. M. Parulkar, "A real-time upcall facility for protocol processing with QoS guarantees," in *Proc. ACM Symp. on Operating Systems Principles*, p. 231, December 1995.

[69] R. Gopalakrishnan and G. M. Parulkar, "Bringing real-time scheduling theory and practice closer for multimedia computing," in *Proc. of ACM SIGMETRICS*, pp. 1–12, May 1996.

[70] R. Govindan and D. P. Anderson, "Scheduling and IPC mechanisms for continuous media," in *Proc. ACM Symp. on Operating Systems Principles*, pp. 68–80, October 1991.

[71] P. Goyal, X. Guo, and H. M. Vin, "A hierarchical CPU scheduler for multimedia operating systems," in *Proc. USENIX Symp. on Operating Systems Design and Implementation*, pp. 107–121, October 1996.

[72] P. Gunningberg, M. Bjorkman, E. Nordmark, S. Pink, P. Sjodin, and J.-E. Stromquist, "Application protocols and performance benchmarks," *IEEE Communication Magazine*, pp. 30–36, June 1989.

[73] O. Hagsand and P. Sjodin, "Workstation support for real-time multimedia communication," in *Winter USENIX Conference*, pp. 133–142, January 1994.

[74] T.-Y. Huang, J. W. Liu, and D. Hull, "A method for bounding the effect of DMA I/O interference on program execution time," in *Proc. 17th Real-Time Systems Symposium*, pp. 275–285, December 1996.

[75] J.-F. Huard, "kStack: A user space native-mode ATM transport layer with QoS support," Technical Report CU/CTR TR 463-96-29, Center for Telecommunications Research, Columbia University, October 1996.

[76] N. C. Hutchinson and L. L. Peterson, "The x-Kernel: An architecture for implementing network protocols," *IEEE Trans. Software Engineering*, vol. 17, no. 1, pp. 1–13, January 1991.

[77] A. Indiresan, A. Mehra, and K. Shin, "Design tradeoffs in implementing real-time channels on bus-based multiprocessor hosts," Technical Report CSE-TR-238-95, University of Michigan, April 1995.

[78] A. Indiresan, A. Mehra, and K. Shin, "The *END*: An *E*mulated *N*etwork *D*evice for evaluating adapter design," in *Proc. 3rd Intl. Workshop on Performability Modeling of Computer and Communication Systems (PMCCS3)*, pp. 90–94, September 1996.

[79] A. Indiresan, A. Mehra, and K. G. Shin, "The END: A network adapter design tool," RTCL Technical Report, University of Michigan, June 1997.

[80] A. Indiresan, A. Mehra, and K. G. Shin, "Exploring QoS support in adapters via an Emulated Network Device," Submitted for publication, May 1997.

[81] A. Indiresan, A. Mehra, and K. G. Shin, "Receive livelock elimination via dynamic interrupt rate control," RTCL Technical Report, University of Michigan, June 1997.

[82] M. R. Ito, L. Takeuchi, and G. Neufeld, "A multiprocessor approach for meeting the processing requirements for OSI," *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 2, pp. 220–227, February 1993.

[83] N. Jain, M. Schwartz, and T. R. Bashkow, "Transport protocol processing at GBPS rates," in *Proc. of ACM SIGCOMM*, pp. 188–199, September 1990.

[84] R. Jain, "Congestion control in computer networks: Issues and trends," *IEEE Network Magazine*, pp. 24–30, May 1990.

[85] S. Jamin, P. Danzig, S. Shenker, and L. Zhang, “A measurement-based admission control algorithm for integrated services packet networks,” in *Proc. of ACM SIGCOMM*, pp. 2–13, August 1995.

[86] K. Jeffay, D. L. Stone, T. Talley, and F. D. Smith, “Adaptive best-effort delivery of digital audio and video across packet-switched networks,” in *Lecture Notes in Computer Science*, volume 712, pp. 3–14, Springer-Verlag, 1993.

[87] M. Jones, P. Leach, Joseph Barrera III, and R. Draves, “Support for user-centric modular real-time resource management in the Rialto operating system,” in *Proc. Intl. Workshop on Network and Operating System Support for Digital Audio and Video*, pp. 55–65, April 1995.

[88] M. B. Jones, Joseph S. Barrera III, A. Forin, P. J. Leach, D. la Rosu, and M.-C. Rosu, “An overview of the Rialto real-time architecture,” in *ACM SIGOPS European Workshop on System Support for Worldwide Applications*, September 1996.

[89] H. Kanakia, “Host interface architecture: Key principles,” in *Proc. of the IFIP TC6 Int'l Conf. on Local Area Networks (INDOLAN)*, pp. 79–97, January 1990.

[90] H. Kanakia and D. R. Cheriton, “The VMP network adapter board (NAB): high-performance network communication for multiprocessors,” *Proc. of ACM SIGCOMM*, pp. 175–187, August 1988.

[91] H. Kanakia, P. P. Mishra, and A. Reibman, “An adaptive congestion control scheme for real-time packet video transport,” in *Proc. of ACM SIGCOMM*, pp. 20–31, September 1993.

[92] D. D. Kandlur, K. G. Shin, and D. Ferrari, “Real-time communication in multi-hop networks,” *IEEE Trans. on Parallel and Distributed Systems*, vol. 5, no. 10, pp. 1044–1056, October 1994.

[93] D. D. Kandlur, D. Saha, and M. Willebeek-LeMair, “Protocol architecture for multimedia applications over ATM networks,” *IEEE Journal on Selected Areas in Communications*, vol. 14, no. 7, pp. 1349–1359, September 1996.

[94] H. Kaneko, J. A. Stankovic, S. Sen, and K. Ramamritham, “Integrated scheduling of multimedia and hard real-time tasks,” in *Proc. 17th Real-Time Systems Symposium*, pp. 206–217, December 1996.

[95] D. Katcher, H. Arakawa, and J. K. Strosnider, “Engineering and analysis of fixed priority schedulers,” *IEEE Trans. Software Engineering*, vol. 19, no. 9, pp. 920–934, September 1993.

[96] J. Kay and J. Pasquale, “The importance of non-data touching processing overheads in TCP/IP,” in *Proc. of ACM SIGCOMM*, pp. 259–268, September 1993.

[97] J. Kay and J. Pasquale, “Measurement, analysis, and improvement of UDP/IP throughput for the DECstation 5000,” in *Proc. USENIX Winter Conference*, pp. 249–258, January 1993.

[98] K. A. Kettler, D. I. Katcher, and J. K. Strosnider, "A modeling methodology for real-time/multimedia operating systems," in *Proc. of the Real-Time Technology and Applications Symposium*, pp. 15–26, May 1995.

[99] S. Khanna, M. Sebree, and J. Zolnowsky, "Realtime scheduling in SunOS 5.0," in *Winter USENIX Conference*, pp. 375–390, January 1992.

[100] D. G. Korn, "Porting UNIX to windows NT," in *Proc. USENIX Winter Conference*, January 1997.

[101] C. Kosak, D. Eckhardt, T. Mummert, P. Steenkiste, and A. Fisher, "Buffer management and flow control in the Credit Net ATM host interface," in *Proc. Conference on Local Computer Networks*, pp. 370–378, October 1995.

[102] I. Kouvelas and V. Hardman, "Overcoming workstation scheduling problems in a real-time audio tool," in *Proc. USENIX Winter Conference*, January 1997.

[103] A. Krishnakumar and K. Sabnani, "VLSI implementations of communication protocols – a survey," *IEEE Journal on Selected Areas in Communications*, vol. 7, no. 7, pp. 1082–1090, September 1989.

[104] L. Krishnamurthy, *AQUA: An Adaptive Quality of Service Architecture for Distributed Multimedia Applications*, PhD thesis, University of Kentucky, 1997.

[105] C.-G. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim, "Analysis of cache-related preemption delay in fixed-priority preemptive scheduling," in *Proc. 17th Real-Time Systems Symposium*, pp. 264–274, December 1996.

[106] C. Lee, R. Rajkumar, and C. Mercer, "Experiences with processor reservation and dynamic QOS in Real-Time Mach," in *Proc. of Multimedia Japan*, March 1996.

[107] C. Lee, K. Yoshida, C. Mercer, and R. Rajkumar, "Predictable communication protocol processing in Real-Time Mach," in *Proc. Real-Time Technology and Applications Symposium*, pp. 220–229, June 1996.

[108] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.3BSD Unix Operating System*, Addison Wesley, May 1989.

[109] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Faribairns, and E. Hyden, "The design and implementation of an operating system to support distributed multimedia applications," *IEEE Journal on Selected Areas in Communications*, vol. 14, no. 7, pp. 1280–1297, September 1996.

[110] J. Liedtke, H. Hartig, and M. Hohmuth, "OS-controlled cache predictability for real-time systems," in *Proc. Real-Time Technology and Applications Symposium*, pp. 213–223, June 1997.

[111] C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in hard real-time environment," *Journal of the ACM*, vol. 1, no. 20, pp. 46–61, January 1973.

[112] C. Maeda and B. N. Bershad, "Protocol service decomposition for high-performance networking," in *Proc. ACM Symp. on Operating Systems Principles*, pp. 244–255, December 1993.

[113] B. D. Marsh, T. J. LeBlanc, M. L. Scott, and E. P. Markatos, "First-class user-level threads," in *Proc. ACM Symp. on Operating Systems Principles*, pp. 110–121, October 1991.

[114] S. McCanne and V. Jacobson, "The BSD packet filter: A new architecture for user-level packet capture," in *Proc. USENIX Winter Conference*, pp. 259–269, January 1993.

[115] L. McVoy and C. Staelin, "lmbench: Portable tools for performance analysis," in *Proc. USENIX Winter Conference*, pp. 279–295, January 1996.

[116] A. Mehra, A. Indiresan, and K. Shin, "Resource management for real-time communication: Making theory meet practice," in *Proc. of 2nd Real-Time Technology and Applications Symposium*, pp. 130–138, June 1996.

[117] A. Mehra, A. Indiresan, and K. Shin, "Structuring communication software for quality of service guarantees," in *Proc. 17th Real-Time Systems Symposium*, pp. 144–154, December 1996.

[118] A. Mehra, J. Rexford, H.-S. Ang, and F. Jahanian, "Design and implementation of a window-consistent replication service," in *Proc. Real-Time Technology and Applications Symposium*, pp. 182–191, June 1995.

[119] A. Mehra and K. Shin, "QoS-sensitive protocol processing in shared-memory multiprocessor multimedia servers," in *Proc. of 3rd IEEE Workshop on Architecture and Implementation of High-Performance Communication Subsystems*, pp. 163–169, August 1995.

[120] O. Menzilcioglu and S. Schlick, "Nectar CAB: A high-speed network processor," in *Proc. Int'l Conf. on Distributed Computing Systems*, pp. 508–515, May 1991.

[121] C. W. Mercer and H. Tokuda, "Preemptibility in real-time operating systems," in *Proc. Real-Time Systems Symposium*, December 1992.

[122] C. W. Mercer and R. Rajkumar, "An interactive interface and RT-Mach support for monitoring and controlling resource management," in *Proc. Real-Time Technology and Applications Symposium*, May 1995.

[123] C. W. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves for multimedia operating systems," in *Proc. of the IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[124] J. Mogul and A. Borg, "The effect of context switches on cache performance," in *Proc. Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 75–85, April 1991.

[125] J. Mogul and K. K. Ramakrishnan, "Eliminating receive livelock in an interrupt-driven kernel," in *Winter USENIX Conference*, January 1996.

[126] J. C. Mogul, "Network locality at the scale of processes," *ACM Trans. Computer Systems*, vol. 10, no. 2, pp. 81–109, May 1992.

[127] J. C. Mogul, R. F. Rashid, and M. J. Accetta, "The packet filter: An efficient mechanism for user-level network code," in *Proc. ACM Symp. on Operating Systems Principles*, pp. 39–51, November 1987.

[128] D. Mosberger and L. L. Peterson, "Making paths explicit in the Scout operating system," in *Proc. USENIX Symp. on Operating Systems Design and Implementation*, pp. 153–167, October 1996.

[129] D. Mosberger, L. L. Peterson, P. G. Bridges, and S. O'Malley, "Analysis of techniques to improve protocol processing latency," in *Proc. of ACM SIGCOMM*, pp. 73–84, October 1996.

[130] B. Murphy, S. Zeadally, and C. J. Adams, "An analysis of process and memory models to support high-speed networking in a UNIX environment," in *Proc. USENIX Winter Conference*, January 1996.

[131] K. Nahrstedt and J. M. Smith, "The QoS broker," *IEEE Multimedia*, vol. 2, no. 1, pp. 53–67, Spring 1995.

[132] K. Nahrstedt and J. M. Smith, "Design, implementation and experiences of the OMEGA end-point architecture," *IEEE Journal on Selected Areas in Communications*, vol. 14, no. 7, pp. 1263–1279, September 1996.

[133] E. Nahum, D. Yates, J. Kurose, and D. Towsley, "Cache behavior of network protocols," in *Proc. of ACM SIGMETRICS*, pp. 169–180, June 1997.

[134] E. M. Nahum, D. J. Yates, J. F. Kurose, and D. Towsley, "Performance issues in parallelized network protocols," in *Proc. USENIX Symp. on Operating Systems Design and Implementation*, pp. 125–137, November 1994.

[135] T. Nakajima, "A dynamic QOS control based on optimistic processor reservation," in *Proc. Intl. Conf. on Multimedia Computing and Systems*, 1996.

[136] T. Nakajima and H. Tezuka, "Virtual memory management for interactive continuous media applications," in *Proc. Intl. Conf. on Multimedia Computing and Systems*, June 1997.

[137] Netperf Homepage. http://www.cup.hp.com/netperf/NetperfPage.html.

[138] A. N. Netravali, W. D. Roome, and K. Sabnani, "Design and implementation of a high-speed transport protocol," *IEEE Trans. Communications*, vol. 38, no. 11, pp. 2010–2024, November 1990.

[139] J. Nieh and M. S. Lam, "The design, implementation and evaluation of SMART: A scheduler for multimedia applications," in *Proc. ACM Symp. on Operating Systems Principles*, October 1997.

[140] E. Nordmark and D. R. Cheriton, "Experiences from VMTP: How to achieve low response time," in *Protocols for High-Speed Networks*, H. Rudin and R. Williamson, editors, pp. 43–54, North-Holland, 1989.

[141] S. W. O'Malley and L. L. Peterson, "A dynamic network architecture," *ACM Trans. Computer Systems*, vol. 10, no. 2, pp. 110–143, May 1992.

[142] J. K. Ousterhout, "Why aren't operating systems getting faster as fast as hardware?," in *Summer USENIX Conference*, pp. 1–10, June 1990.

[143] C. Papadopoulos and G. M. Parulkar, "Experimental evaluation of SUNOS IPC and TCP/IP protocol implementation," *IEEE/ACM Trans. Networking*, vol. 1, no. 2, pp. 199–216, April 1993.

[144] A. K. Parekh and R. G. Gallager, "A generalized processor sharing approach to flow control in integrated services networks – the single node case," in *Proc. IEEE INFOCOM*, pp. 915–924, May 1992.

[145] C. Partridge, *Gigabit Networking*, Addison Wesley, One Jacob Way, Reading, Massachusetts 01867, 1994.

[146] C. Partridge and S. Pink, "A faster UDP," *IEEE/ACM Trans. Networking*, vol. 1, no. 4, pp. 429–440, August 1993.

[147] J. Pasquale, E. Anderson, and P. Muller, "Container shipping: Operating system support for I/O-intensive applications," *IEEE Computer*, vol. 27, no. 3, pp. 84–93, March 1994.

[148] B. Pehrson, P. Gunningberg, and S. Pink, "Distributed multimedia applications on gigabit networks," *IEEE Network Magazine*, pp. 26–35, January 1992.

[149] M. Perez, F. Liaw, A. Mankin, E. Hoffman, D. Grossman, and A. Malis, "ATM signaling support for IP over ATM," *Request for Comments RFC 1755*, February 1995. ISI, Fore, Motoral Codex, Ascom Timeplex.

[150] T. F. L. Porta and M. Schwartz, "The MultiStream Protocol: A highly flexible high-speed transport protocol," *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 4, pp. 519–530, May 1993.

[151] K. K. Ramakrishnan, "Performance considerations in designing network interfaces," *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 2, pp. 203–219, February 1993.

[152] D. C. Schmidt and T. Suda, "Transport system architecture services for high-performance communications systems," *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 4, pp. 489–506, May 1993.

[153] D. C. Schmidt and T. Suda, "Measuring the performance of parallel message-based process architectures," in *Proc. IEEE INFOCOM*, pp. 624–633, April 1995.

[154] D. C. Schmidt and T. Suda, "The performance of alternative threading architectures for parallel communication subystems," *Journal of Parallel and Distributed Computing*, 1997. To appear.

[155] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "RTP: A transport protocol for real-time applications," *Request for Comments RFC 1889*, January 1996. GMD/Precept Software/PARC/LBNL.

[156] H. Schulzrinne, J. Kurose, and D. Towsley, "An evaluation of scheduling mechanisms for providing best-effort, real-time communication in wide-area networks," in *Proc. IEEE INFOCOM*, June 1994.

[157] M. Seltzer and C. Small, "Self-monitoring and self-adapting operating systems," in *Proc. Workshop on HoTOS*, May 1997.

[158] S. Shenker, D. Clark, and L. Zhang, "A scheduling service model and a scheduling architecture for an integrated services packet network," *Working Paper*, August 1993. Xerox PARC.

[159] S. Shenker, C. Partridge, and R. Guerin, "Specification of guaranteed quality of service," *Internet Draft draft-ietf-intserv-guaranteed-svc-05.txt*, June 1996. Xerox/BBN/IBM.

[160] P. Sjodin, P. Gunningberg, E. Nordmark, and S. Pink, "Towards protocol benchmarks," in *Protocols for High-Speed Networks*, H. Rudin and R. Williamson, editors, pp. 57–67, North-Holland, 1989.

[161] J. M. Smith and C. B. S. Traw, "Giving applications access to Gb/s networking," *IEEE Network Magazine*, pp. 44–52, July 1993.

[162] S. Sommer and J. Potter, "Operating system extensions for dynamic real-time applications," in *Proc. 17th Real-Time Systems Symposium*, pp. 45–50, December 1996.

[163] P. Steenkiste, "Analyzing communication latency using the Nectar communication processor," in *Proc. of ACM SIGCOMM*, pp. 199–209, August 1992.

[164] P. A. Steenkiste, "A systematic approach to host interface design for high-speed networks," *IEEE Computer*, pp. 47–57, March 1994.

[165] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. K. Baruah, J. E. Gehrke, and C. G. Plaxton, "A proportional share resource allocation algorithm for real-time time-shared systems," in *Proc. 17th Real-Time Systems Symposium*, pp. 288–299, December 1996.

[166] The Real-Time Group, *OSF RI MK7.2 Release Notes*, OSF Research Institute, October 1996.

[167] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. Lazowska, "Implementing network protocols at user level," in *Proc. of ACM SIGCOMM*, pp. 64–73, October 1993.

[168] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. Lazowska, "Implementing network protocols at user level," *IEEE/ACM Trans. Networking*, vol. 1, no. 5, pp. 554–565, October 1993.

[169] C. A. Thekkath, D. L. Eager, E. D. Lazowska, and H. M. Levy, "A performance analysis of network I/O in shared-memory multiprocessors," Computer Science and Engineering Technical Report 92-04-04, University of Washington, April 1992.

[170] C. A. Thekkath and H. M. Levy, "Limits to low-latency communication on high-speed networks," *ACM Trans. Computer Systems*, vol. 11, no. 2, pp. 179–203, May 1993.

[171] H. Tokuda et al., "Real-Time Mach: Towards a predictable real-time system," in *Proc. USENIX Mach Workshop*, pp. 73–82, 1993.

[172] F. Travostino, E. Menze, and F. Reynolds, "Paths: Programming with system resources in support of real-time distributed applications," in *Proc. IEEE Workshop on Object-Oriented Real-Time Dependable Systems*, February 1996.

[173] C. B. S. Traw and J. M. Smith, "Hardware/software organization of a high-performance ATM host interface," *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 2, pp. 240–253, February 1993.

[174] A. Tucker and A. Gupta, "Process control and scheduling issues for multiprogrammed shared-memory multiprocessors," in *Proc. ACM Symp. on Operating Systems Principles*, pp. 159–166, December 1989.

[175] S.-Y. Tzou and D. P. Anderson, "The performance of message-passing using restricted virtual memory remapping," *Software – Practice and Experience*, vol. 21, no. 3, pp. 251–267, March 1991.

[176] R. van Renesse, "Masking the overhead of protocol layering," in *Proc. of ACM SIGCOMM*, pp. 96–104, October 1996.

[177] C. Vogt, R. G. Herrtwich, and R. Nagarajan, "HeiRAT: The Heidelberg resource administration technique design philosophy and goals," Research Report 43.9213, IBM Research Division, IBM European Networking Center, Heidelberg, Germany, 1992.

[178] I. Wakeman, A. Ghosh, J. Crowcroft, V. Jacobson, and S. Floyd, "Implementing real-time packet forwarding policies using Streams," in *Proc. USENIX Winter Conference*, 1995.

[179] C. Waldspurger, *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*, PhD thesis, Technical Report, MIT/LCS/TR-667, Laboratory for CS, MIT, September 1995.

[180] D. Wallach, D. Engler, and M. F. Kaashoek, "ASHs: Application-specific handlers for high-performance messaging," in *Proc. of ACM SIGCOMM*, pp. 40–52, August 1996.

[181] R. W. Watson and S. A. Mamrak, "Gaining efficiency in transport services by appropriate design and implementation choices," *ACM Trans. Computer Systems*, vol. 5, no. 2, pp. 97–120, May 1987.

[182] C. M. Woodside and R. G. Franks, "Alternative software architectures for parallel protocol execution with synchronous IPC," *IEEE/ACM Trans. Networking*, vol. 1, no. 2, pp. 178–186, April 1993.

[183] J. Wroclawski, "Specification of controlled-load network element service," *Internet Draft draft-ietf-intserv-ctrl-load-svc-03.txt*, June 1996. MIT.

[184] D. K. Y. Yau and S. S. Lam, "Adaptive rate-controlled scheduling for multimedia applications," in *Proc. of ACM Multimedia*, November 1996.

[185] D. K. Y. Yau and S. S. Lam, "An architecture towards efficient OS support for distributed multimedia," in *Proc. IST/SPIE Multimedia Computing and Networking*, January 1996.

[186] M. Yuhara, B. N. Bershad, C. Maeda, and J. E. B. Moss, "Efficient packet demultiplexing for multiple endpoints and large messages," in *Proc. USENIX Winter Conference*, January 1994.

[187] H. Zhang and D. Ferrari, "Rate-controlled static-priority queueing," in *Proc. IEEE INFOCOM*, pp. 227–236, June 1993.

[188] H. Zhang, "Service disciplines for guaranteed performance service in packet-switching networks," *Proceedings of the IEEE*, vol. 83, no. 10, , October 1995.

[189] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala, "RSVP: A new resource ReSerVation Protocol," *IEEE Network*, pp. 8–18, September 1993.

[190] X. Zhang, Z. Wang, N. Gloy, J. B. Chen, and M. D. Smith, "Operating system support for automatic profiling and optimization," in *Proc. ACM Symp. on Operating Systems Principles*, October 1997.

[191] M. Zitterbart, B. Stiller, and A. N. Tantawy, "A model for flexible high-performance communication subsystems," *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 4, pp. 507–518, May 1993.