Active Networks: Architectures, Composition, and Applications

A Thesis
Presented to
The Academic Faculty

by

Samrat Bhattacharjee

In Partial Fulfillment
of the Requirements for the Degree of
Doctor of Philosophy in Computer Science

Georgia Institute of Technology
July 1999

Active Networks: Architectures, Composition, and Applications

Approved:

Dr. Kenneth L. Calvert (Co-advisor)
(University of Kentucky)

Dr. Ellen W. Zegura (Co-advisor)
(College of Computing)

Dr. M. H. Ammar
(College of Computing)

Dr. A. G. Greenberg (External Member)
(AT&T Labs)

Dr. K. Schwan
(College of Computing)

Date Approved

Acknowledgments

I acknowledge and thank my co-advisors, Ellen Zegura and Ken Calvert, for their support, patience, and guidance during my tenure at Georgia Tech. They have led by example, during the good times and the bad. Ultimately, Ken and Ellen have taught me much — about academia and beyond; I thank them for making my time at Tech literally the best years of my life. I have also been fortunate to work with Mostafa Ammar and Karsten Schwan and I thank them for their support over the years and for serving on my thesis committee. I thank Albert Greenberg for his guidance while I was at AT&T Labs and for serving on my thesis committee. I acknowledge the financial support I received from the Defense Advanced Research Project Agency, the National Science Foundation, and the State of Georgia.

Members of the CANEs research group at Georgia Tech —Shashidhar Merugu, Matt Sanders, Bill McKinnon, Youngsu Chae and Richard Liston— have had to endure me every day for various parts of the last two years. I thank them for their patience and for creating a wonderful research environment. I thank John Limb and Mani Subramanian for their help and for sharing their insights on active networking with me. In the broader research community, I acknowledge the support of Gísli Hjálmtýsson, Jennifer Rexford, and Anja Feldman at AT&T Labs, James Sterbenz at BBN, and Bob Braden at ISI.

Richard Clayton, Rajesh Talpade, Bobby Krupczak, Kevin Almeroth, Xue Li, Naser Adas, and Dolors Sala, all ex-members of the Networking and Telecom group, have helped me in innumerable ways over the years and I am grateful for their assistance and friendship. I also thank all present members, especially Fang Hao, Zongming Fei, Zhiruo Cao, Joe Dixon, Lenitra Clay, Mary Chamlee, Minaxi Gupta, and Tianji Jiang for making my time at the group both lively and interesting. Special thanks to George Riley, Viren Shah, Richard West, Ion Mandoiu, Sumeer Bhola and Kurt Eiselt at the College of Computing for their support and friendship. I thank Peter Wan and the CNS staff for providing an effective computing environment; I also thank the volunteer programmers of the Free Software Foundation for creating and maintaining invaluable tools such as `GNU/Linux`, `emacs`, `gcc` and LaTeX without which this work could not be completed.

Dave Hawkins and Craig Henry have been mentors and valuable friends; I thank them for their support over the last decade. I thank Dave Devries and Gerald Adkins at Georgia College for their guidance over the years. Anindo Paul, Anuj Agarwal, Bona Sen, Sumit Roy, Rajesh Mohanty, Pilar Araujo, Chris Davidson, Michael Stone, Nanda Negandhi, Pratibha Singh and Malati Singh have been my extended family over the years; I thank them for their friendship. I thank my brothers, Ashish and Devashish, my sister Samraggi, and my grandparents for their love and support over the years. This thesis could not have started without the vision and guidance of Sanjay Singh and I am grateful to him for that. I thank Susan and the Schmoyer family for making the last two years memorable. Lastly, I thank my parents without whose love and sacrifice I would not be here and to whom I owe everything.

Contents

Acknowledgments **iii**

List of Tables **x**

List of Figures **xi**

Summary **xvii**

1 Introduction **1**

1.1 Active Networking 5

1.1.1 What about End-to-End arguments? 6

1.2 Previous work 9

1.2.1 Commercial Efforts 9

1.2.2 Open Signaling: Control Plane Flexibility for Circuit-Switched Networks 10

1.2.3 The Intelligent and Advanced Intelligent Networks 10

1.3 Thesis Outline 12

1.4 Organization of Dissertation 13

2 Architectural Considerations for Active Networks **16**

2.1 Introduction 16

2.1.1 A Spectrum of Interfaces 17

2.1.2 Tradeoffs (or Which Network API should the Active Network Support?) 20

2.2 Architecture Overview 22

2.2.1 Execution Environments 23

2.2.2 Node Operating System 23
2.2.3 Active Network Encapsulation Protocol 25
2.2.4 Interfaces and Standardization 26
2.3 Implementation Study: Control on Demand 26
2.3.1 Control on Demand 27
2.3.2 CoD: Fundamental Concepts 28
2.3.3 The CoD Node Architecture 30
2.4 Related Work 37
2.4.1 Execution Environments 37
2.4.2 Node Operating Systems 40

3 Composition in Active Networks 43
3.1 Overview of Slots 44
3.1.1 Properties of Slots 47
3.2 The CANEs Execution Environment 49
3.2.1 Composing services in CANEs 50
3.3 CANEs Implementation in the Odyssey Active Network 53
3.3.1 Overview of Bowman 53
3.3.2 CANEs in Bowman 55
3.4 The CANEs API 56
3.4.1 The Slot Mechanism 58
3.4.2 Variable Sharing 59
3.4.3 Declaring and Exporting Shared Variables 61
3.4.4 Packet Transmission 62
3.4.5 Timers 63
3.4.6 Miscellaneous 64
3.4.7 Example Usage of the CANEs API 65
3.4.8 CANEs Signaling 68
3.5 Related Work 73

3.5.1 SwitchWare 73

3.5.2 NetScript 74

3.5.3 ANTS 75

3.5.4 SmartPackets 76

4 Application Specific Congestion Control 77

4.1 Introduction 77

4.1.1 Background 77

4.1.2 A Nomenclature for Congestion Control Mechanisms 81

4.1.3 Roadmap 84

4.2 Application-specific congestion control 84

4.2.1 Architectural Requirements for ASCC 85

4.3 ASCC applied to MPEG Video Streams 88

4.3.1 MPEG Discard Algorithms 89

4.3.2 Implementation 91

4.3.3 State and Computation Requirements 94

4.3.4 Router Information 95

4.3.5 Congestion Triggers 95

4.3.6 Evaluation Metrics for MPEG 95

4.4 Network support for unicast video 97

4.4.1 Experimental Methodology 97

4.4.2 Single Active Router 99

4.4.2.1 Changing Buffer Capacity 102

4.4.2.2 Bits vs. Information 104

4.4.2.3 End-to-end Feedback 105

4.4.2.4 Effect of Output Queue Manipulation 108

4.4.2.5 Signal-to-Noise Ratio 110

4.4.3 Multiple Active Nodes 111

4.4.4 Discussion 113

4.5 Network Support for Multicast Video . . . 115
4.5.1 ASCC for Multicast . . . 117
4.5.2 Receiver-based Adaptation . . . 117
4.5.3 Experimental Environment . . . 118
4.5.3.1 Source Models . . . 118
4.5.3.2 Topology . . . 119
4.5.4 Results . . . 120
4.5.4.1 Single Video Source . . . 121
4.5.4.2 Detail Evaluation of Received Stream . . . 127
4.5.5 Multiple Video Sources . . . 130
4.5.5.1 Multiple Equal Weight Sources . . . 130
4.5.5.2 Multiple Prioritized Sources . . . 132
4.5.5.3 Periodic High Priority Source . . . 134
4.6 Discussion . . . 136
4.6.1 Alternate Models of Network-based Adaptation . . . 137
4.6.2 Citizenship . . . 139
4.6.3 Alternate Definitions of Decodable . . . 141
4.7 Related Work . . . 144
4.7.1 Congestion Control Mechanisms . . . 144
4.7.2 Applications for Active Networks . . . 145
4.7.3 Transfer and Transcoding of Continuous Media . . . 145
4.8 Summary and Extensions . . . 146

5 Self-Organizing Wide-Area Network Caches **149**
5.1 Introduction . . . 149
5.2 Background . . . 150
5.2.1 Wide-area caching . . . 150
5.2.2 Related work . . . 151
5.3 Network Caching Schemes . . . 152

5.3.1 Self-Organizing Caches 153
5.3.2 Traditional Mechanisms 155
5.3.3 Discussion 156
5.4 Simulation Methodology 156
5.5 Simulation Results 160
5.5.1 Summary 160
5.5.2 Variation in Cache Size 160
5.5.3 Variation in Server Distribution 163
5.5.4 Variation in Topology 164
5.5.5 Spatial Access Patterns 165
5.6 Analysis of Cache Performance 166
5.6.1 Network Caching Performance Model 167
5.6.2 Optimal cache partitioning 168
5.7 Conclusions 173

6 Conclusions **174**

6.1 Summary of Contributions 174
6.1.1 Architecture 174
6.1.2 Composition 175
6.1.3 Applications 176
6.2 On Active Networking 177
6.2.1 Open Problems 179
6.3 Concluding Remarks 180

Bibliography **181**

Vita **190**

List of Tables

1 The CoD network API. The calls for reading and writing packets from channels is inherited from the Node OS and is not shown here. 35

2 CANEs API: Defined Data Types. Internally, each type is mapped onto a data type in C or to a type defined by Bowman. 58

3 Fields in the CANEs packet structure (inherited from Bowman). . . 63

4 Congestion Control Space 82

5 ASCC algorithms for MPEG transfer. The Frame drop and GOP drop algorithms are able to recognize application-level units (frames in the case of MPEG) while the Static Priority and Tail Drop algorithms operate on individual packets. 89

6 Loss details corresponding to 99% link load (average background source rate of 1260 Kbps). The 8K and 24 K are the sizes of the shared buffer at the router. Note that, without application-specific techniques, even a small fraction of lost bits manifests itself as far greater losses at the application layer. 104

7 Details of Simulated MPEG Streams 119

8 Simulated Topologies 157

9 Definitions and Notation 169

List of Figures

1 Active versus traditional networking. 5

2 Components of the Active Networking Architecture. Each EE implements one particular network API. Access to node resources are mediated by the Node OS. 22

3 Packet flow through an active node. Incoming packets are demultiplexed to EEs. Depending on the EE, packets may receive individualized treatment. Details of input and output channel processing is not shown in this figure. 25

4 CoD Architecture showing the CoD EE and the Node OS. Note that the Node OS maintains per-flow queues and the forwarding engine. . 31

5 CoD Flow Classification. Each CoD packet carries a flow identifier which is used to classify packets and demultiplex them into per-flow queues. 34

6 Implementation of the CoD EE using Linux as the Node OS. Parts of the M-controller, per-flow queuing and per-flow packet scheduling was added to the base Linux kernel. 36

7 Schematic of a CANEs Underlying Program (with comments guarded by semi-colons). For simplicity, this example assumes that slot processing does not mutate packets. 52

8 A high level view of processing at an active node running the CANEs EE over the Bowman Node OS. User flows consist of an underlying program (U in this figure) and a set of injected programs (A, B, and C). This figure also shows a nominal thread level structure of our implementation. The CANEs EE spawns a control a-flow that contains a signaling thread; each user flow has its own a-flow and set of threads. There are also a set of system threads that run within the Node OS. 57
9 Prototypes for CANEs slot control functions 59
10 Prototype for CANEs variable sharing functions and declarations. . . 61
11 Prototype of the CANEs data transmission functions 62
12 Prototype of CANEs initialization and support functions 65
13 Preamble and the initialization for the example CANEs underlying program. Comments are shown in italics. 66
14 The `default-fwd` function. This function is invoked when the user a-flow executes. (Calls to free the packet are not shown in this listing.) 67
15 An injected program designed to bind to the `DefaultRoute` slot of the `default-fwd` underlying program. Calls to resolve the routing table identifier (`rt_id`) are not shown. 68
16 Top-level structure of CUI messages 69
17 Computation part of CUI messages 70
18 IO part of CUI messages 71
19 An example CUI message that binds the program `route` to the slot `DefaultRoute`. CUI keywords are shown in bold face; comments are italicized and guarded by semi-colons. 72
20 Schematic of an output buffered router. Packets come in on input links and are routed through the switching engine to the output links. Packets routed to a busy output link are buffered (if possible). . . . 77

21 The sender adaptation model. Congestion is detected by the receiver in the form of lost packets. The receiver then sends a congestion notification signal to the sender. Upon receipt of this signal, the sender decreases the transmission rate. 79
22 A schematic diagram of the implementation steps for MPEG discard algorithms. The Tail Drop and Static Priority algorithms do not maintain any state. The queue eviction step is required only for algorithms that maintain priority (GOP Drop and Static Priority). . 91
23 Emulation topology for unicast video streams: The active router functionality is implemented within a SunOS 4.1.3 kernel. The end-hosts are Solaris 2.5.1. The second router is present only for multiple active router experiments; otherwise, the first active router is connected directly to the destination. 97
24 Fraction of I frames received when output link is congested. Note that the GOP discard scheme performs relatively better as congestion increases. 100
25 Fraction of received data discarded at receiver when output link is congested. The GOP discard mechanism does not transmit *any* useless data while all of the data transmitted by the Frame Drop scheme is useless. With a 8K buffer, the performance of the Static Priority and Tail Drop mechanisms are nearly identical. 101
26 Fraction of I frames received with 24 K bytes of output buffering. Note that the Static Priority and Frame Drop schemes perform better with larger buffers. 102
27 Fraction of received data discarded at receiver with a 24 K byte output buffer. Note that even with larger buffers, the Tail Drop, Static Priority and Frame Drop schemes transmit substantial amounts of useless data. 103

28 Fraction of I-frames received with a 8K buffer and end-to-end feedback active. Note that the performance with feedback is not significantly different from performance without feedback (Figure 24). This is because the background traffic does not back off during congestion. 106
29 Fraction of I-frames received with a 24 K buffer and destination feedback active. Note that under low link loads, Static Priority performs as well as GOP Drop. 107
30 Fraction of received data that is useless: 24K buffer and destination feedback enabled. As noted before, when a link is congested, a significant fraction of data transmitted by generic mechanisms is useless at the receiver. 108
31 Comparison of the different mechanisms that comprise the GOP Drop scheme. For each of 8K byte and 24K byte buffer sizes, we plot the fraction of I-frames received when the GOP drop scheme is implemented completely (GOP Drop in the figure) versus when only the Discard-on-Arrival mechanism is implemented (GOP Input Drop in the figure). Note that in these experiments, most of the benefit is derived with just the (relatively more efficient) GOP Input Drop scheme.109
32 Signal to noise ratio of the decoded MPEG stream for a segment of one particular run. End-to-end feedback is enabled. The points below the x-axis correspond to complete frames received under each scheme. In this segment, the GOP Drop scheme manages to receive all but one I-frame (in this MPEG stream, I-frames have frame index 0, 10, 20, ...). 110
33 Fraction of I-frames decoded at the receiver. There are two active nodes in this topology; the buffer size at each node is 8 K bytes. . . . 112

34 Fraction of received data that is useless at the receiver. Data is transmitted through two active nodes with 8K bytes buffer each. The fraction of useless data transmitted by generic mechanisms increases when the data traverses through multiple bottleneck nodes. 113
35 Fraction of I-frames received with end-to-end feedback active. Note that while traversing multiple nodes, Static Priority outperforms Tail Drop even with a 8K byte buffer at each node. 114
36 Fraction of received data useless; multiple active nodes with 8 K byte buffer each. 115
37 Experimental Topology 120
38 Single video source, varying high, low watermark 122
39 Single video source, varying high, low watermark 123
40 Single video source, varying buffer size, 1000 Kbps Link 124
41 Fraction of frames decoded at destination D1 with varying background traffic from B0 $\rightarrow$ DB. Note that when the congestion is not severe, Tail Drop can outperform Receiver-based schemes. 126
42 Single video source, decoded frames per second at destination D1 . . 127
43 Single video source, fraction of data wasted per second at destination D1 128
44 Decoded Frames over time at destination D1 129
45 Two equal priority video sources, varying buffer size, 1500 Kbps link . 131
46 Fraction of frames decoded at receiver D1 with prioritized video sources(Priority of source S0 > S1). The active and the receiver-based schemes both receive the same fraction of low priority frames; however, the active scheme delivers substantially more decodable high priority frames. 132
47 Periodic high priority source, varying buffer size, 1500 Kbps link . . . 135
48 Periodic high priority source, Received Frame Sequence 136

49 Single video source, varying buffer size, 1000 Kbps Link, Alternate Network-based adaptation schemes 138
50 Single video source, varying buffer size, 1000 Kbps Link, Single shared buffer at routers with backoff-sources 140
51 Single video source, varying buffer size, 1000 Kbps Link, Weighted Fair Queuing at routers with backoff-sources 141
52 Decoder efficiency 142
53 Latency with Low Access Correlation 161
54 Latency with High Access Correlation 162
55 Cache Hits with High Access Correlation 163
56 Latency with Varying Server Distributions 164
57 Topology Effect under Moderate Access Correlation 165
58 Preferred Node Servers 166
59 Comparison of analytic and simulation results — Lookaround Caches 171
60 Optimal partitioning of cache space for local objects and location information 172

Summary

Active networking is a paradigm for data networks in which the network is designed not merely to forward packets, but also to be dynamically programmed in order to support per-user services. An active network exports a "meta-level" interface that exposes the network-level resources, mechanisms, and policies to users. Using these mechanisms, users can install per-user state in the network and "program" the network to execute a distributed algorithm. Along with rapid deployment of network protocols, active networks provide a platform for implementing network services that leverage both application-level information (usually available only at the edges of the network) and information that is only available inside the network.

In this thesis we investigate architectures for active networking and evaluate these network architectures through applications that benefit from active networking. As a part of our work, we have derived a programming model for active networks that supports reasoning independently about the correctness of both the underlying active network platform and the algorithms injected into it. We present an implementation of our programming model as an execution environment for active networks.

The contributions of this thesis are examples of applications, including application-specific congestion control and self-organizing network caches, that benefit from the flexibility provided by active networks; a set of principles that can be used to form the base for designing active networking architectures that support a range of programming interfaces; and a programming model for active networks that supports the ability to extend the underlying service in a controlled manner.

Chapter 1

Introduction

The Internet started as a experimental network in the late 1960s to prove the viability of *packet switching* [54]. Packet-switched networks consist of end nodes (*hosts*) and intermediate nodes (*routers*) that provide connectivity between hosts. Typically, user applications execute on hosts and communicate via routers. In a packet-switched network, the set of communicating nodes transmit information in quanta called *packets*. Packets typically have a maximum size; messages too large to fit in a single packet are segmented and transmitted over multiple packets. Packets contain the addresses of their destinations and are *independently routed* through a packet-switched network. Thus, in a packet-switched network, multiple packets destined to the same address and belonging to the same conversation may take different paths to their destination. This experimental packet-switched network provided best-effort service — not all transmitted packets were guaranteed to be received at the destination. Some packets could be corrupted or dropped inside the network. This is in contrast to *circuit-switched networks* which were the norm when the Internet experiment was conceived. Circuit switched networks provide a dedicated channel of fixed bandwidth to its users and require a signaling phase before data transmission. In three decades of its existence, the Internet has grown from a network consisting of less than ten nodes (in 1969) to over 55 million hosts in 1999[1]. The number of users of the network has grown from a handful of researchers in 1969 to an estimated 200 million people worldwide in 1999 [2]. The Internet is now the single largest data

[1] See, `http://www.nw.com`, `http://www.netsizer.com`
[2] See `http://www.euromktg.com/globstats`

network in the world; it is fair to say that the 1960s experiment in packet switching has been successful.

Networks, such as the Internet, are defined by a family of *protocols*. Protocols describe how nodes communicate, how packet are formatted, how nodes react to different network conditions, etc. These network protocols are executed by each node —both the hosts and the routers— in the network. Traditionally, routers in packet-switched networks have been designed and built for efficient forwarding of user data and have included only the minimal processing necessary for correct forwarding of packets. This simplicity of design had been true of the routers and the protocols in the Internet as well. However, the constitution of packet-switched networks, specifically Internet routers and protocols, has changed. The changes can be traced to the following causes:

- The success of the Internet has altered the dynamics of the applications that run over packet-switched networks. When the Internet was conceived, the prevalent applications were file transfer, electronic mail and remote terminal log in. As late as 1990, file transfer, electronic mail and remote log in accounted for over 70% of the total traffic on the Internet (NSFnet) backbone[3]. The World Wide Web (WWW) did not exist and other network services, i.e. other TCP/UDP services such as name lookup, chat, packet video and audio etc. accounted for less than 15% of the total traffic on the backbone. By 1995, the last year for which backbone statistics are available, file transfer, electronic mail and interactive sessions accounted for about 38% of the traffic on the backbone — the share for other applications had grown to over 50% of the total traffic with the world wide web alone accounting for over 24% of the total traffic. Accurate data is not available for the traffic on the Internet backbone networks after 1995 (the last year NSFnet ran the entire Internet backbone); however, data collected on the MCInet backbone in 1998 [30] shows that the

[3]See `ftp://nic.merit.edu/statistics/nsfnet`

world wide web accounted for around 70% of the total traffic. Fundamentally, it is clear that current packet networks, including the Internet, continually transport more and different traffic types. Many of the new traffic types, e.g. continuous media traffic such as video and audio transmission, can benefit from a network service that is richer than just best-effort packet forwarding.

- The Internet has evolved from a research platform to a commercial network. Thus, issues that were not considered by the designers of the original network, such as network security, have to be incorporated into the existing framework.

- The longevity of the Internet has assured that the set of links types that constitute the network has diversified beyond the assumptions made by early protocol designers. Many of the new link types, such as wireless and satellite links in which losses are not necessarily caused by congestion, have characteristics that are different from the assumptions made by the designers of the Internet family of protocols.

- The price-performance ratio for computation has continued to decrease. In certain cases, it is economically viable to add extra processing in routers to improve a given metric (such as performance or security).

As services required of the network increase in both quantity and complexity while relative processing cost decreases, the recent trend has been to introduce more user-specific processing *inside* the network. (Note that in this discussion, *user* refers to any entity that requests network layer services and includes, but is not restricted, to traditional transport protocols, and higher-level abstractions such as flows and processes). Consider the following examples:

- For security purposes, it is now common to include *firewall* functionality in routers such that only pre-defined subsets of packets may traverse certain links.

- Interaction of the TCP protocol's slow-start behavior and wireless links have led to the addition of "ack spoofing" mechanisms in the network that can improve performance by as much as 200% [10].

- Components such as the *PacketShaper*[4], are being to the network infrastructure. The *PacketShaper* is a network component that intercepts and drops selected packets such that critical applications can get fixed or fair share of available bandwidth.

- Application-specific enhancements, such as media gateways that transcode continuous media (such as video) in real-time [4] have been added to the network infrastructure.

These examples show that services are being added —inside the network infrastructure— at the network, transport and application layers. The trend to incorporate processing inside the network looks to continue.

There is, however, a cause for concern. The traditional network design and infrastructure is being populated by *ad-hoc* enhancements. The fundamental problem can be traced back to the essentially *fixed* user-network interface supported by traditional networks such as the Internet. The Internet provides an uniform best-effort forwarding service — to all packets — regardless of end application, user, underlying link, etc. The network interface provides no architectural mechanism to support flexibility within the network. In this work, we investigate *active* networks: networks that provide a (potentially per-user) *programmable* user-network interface. Active networks provide a platform for introducing new services into the network; more importantly, active networks export an *uniform interface* for adding functionality to the network infrastructure.

[4]See `http://www.packeteer.com`

1.1 Active Networking

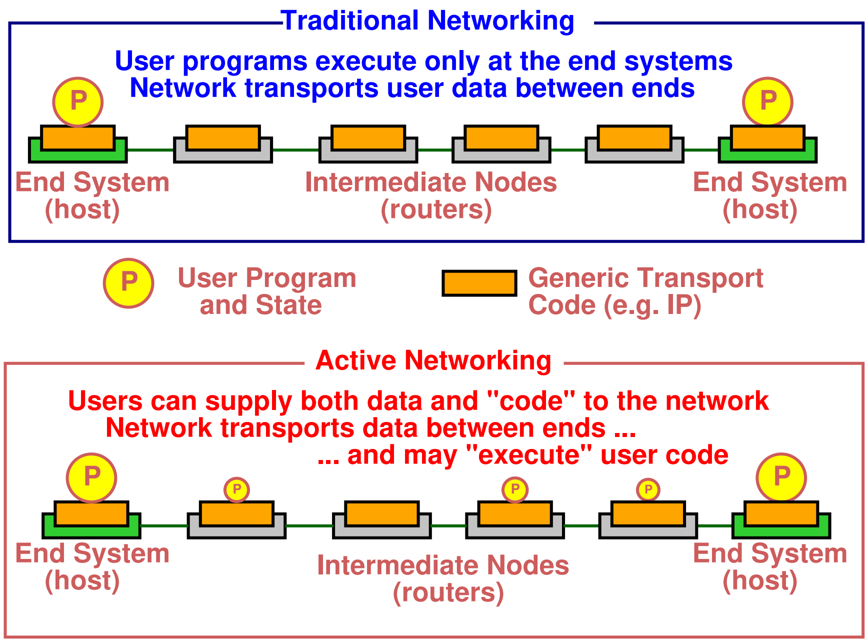

Figure 1: Active versus traditional networking.

Active networking is a new design paradigm in which the network is designed not merely to forward packets, but also to be *dynamically programmed* in order to support per-user services (Figure 1). Active networks export *open* "meta-level" interfaces that expose network-level resources, policies and mechanisms to its users. Using these uniform interfaces, users can dynamically modify the network's behavior and introduce new capabilities inside the network on-the-fly.

There are many possibilities for the programmability of networks. One possible approach is to provide a general computation engine at each node that can be programmed on a per-packet basis. The payloads of user packets carry both data and programs: the network transfers the data and executes programs carried in each packet. As part of this execution, the node state and the contents of the packet may be altered. It is possible to implement active networks without carrying executable code in each packet. Such approaches may download node behaviors, i.e.

programs, on-demand onto active nodes. Active networking approaches may differ in aspects other than the code load mechanism. These differences include (but are not limited to) the abstractions provided to the network programmer, the power of the virtual machine at each node, the degree of flexibility (i.e. whether code can be installed in the data path or the control path), languages used for programming the network nodes, policies for installation of code, state and resource management, etc. However, the fundamental unifying tenet of active networking is that *the network service can be tailored to the user's requirements via a specified and open interface.*

1.1.1 What about End-to-End arguments?

End-to-end arguments in system design, codified by Saltzer, Reed, and Clark [93], are architectural principles that are used to direct the placement of functionality in different components of (distributed) systems. End-to-end arguments are often construed to preclude the implementation of higher-level functions inside the communication subsystem, i.e., the network. As such, it might seem that active networking is the antithesis of the end-to-end argument. End-to-end arguments, however, do not necessarily dictate that all functionality belongs at the ends. (Likewise, active networking does not dictate that all functionality belongs in the network.)

End-to-end arguments "provide a rationale for moving a function upward in a layered system closer to the application that uses the function". According to the end-to-end arguments, the communication subsystem, which is shared by many applications with diverse needs, should not provide "higher layer" functionality that can be best implemented by the ends. Further, higher layer functionality provided within the communication subsystem is may not be useful for some applications. An example of such a "higher layer" function is reliable and error-free transfer: the networking infrastructure can, at considerable cost, guarantee reliable and error-free transfer of data. However, certain applications that require assured data delivery may still require application-level acknowledgments of the receipt of the data (as it is possible for errors to occur outside the network: in the host operating system for example). Further, some applications will not need the network-provided reliability, but incur its overhead. In summary, the end-to-end arguments state that most higher layer functionality can be implemented correctly *only* with the explicit involvement of the ends that require the functionality. As such, it is often not possible for lower layers to provide *generic* higher level functions. Implementations of such functionality at lower layers may impede performance and correctness. Further, applications that do not use particular functionality provided by the lower layers of a system should be isolated from the costs for providing the function.

End-to-End Arguments and Active Networking

We argue that end-to-end arguments do not preclude active networking. Proper application of the end-to-end arguments require the evaluation of the "engineering tradeoff" in the placement of functionality. There are cases in which functionality should, logically, be placed inside the network — hence, the network is the best layer to implement such functionality. Consider the example of reliable multicast: receivers in a multicast distribution tree provide acknowledgments of received data to the source. As the number of receivers increase, the probability of each packet getting lost on some link of the distribution tree (and subsequently having to be re-transmitted to a subset of the receivers) increases as well. (End-to-end solutions to reliable multicast do not scale as they lead to the "acknowledgment implosion" problem at the source: the source is overwhelmed by acknowledgments from receivers). Further, without network-level information, the re-transmissions traverse the entire multicast tree and bandwidth usage is poor. Scalable solutions to reliable multicast must, in some form, aggregate acknowledgments that are sent to the source and also re-transmit only on subtrees on which particular packets have been lost. Such a solution can be effectively implemented inside the network (as the knowledge of both the losses and the topology is available at the routers) [91, 18].

There are other cases in which implementations of functions can be optimized by the coupling of network-level information with application-specific information. Consider the example of application-specific congestion control (described in detail in Chapter 4). In this case, the location and time of congestion is known inside the network; the application, however, knows best how to reduce its data upon congestion. Combining the two leads to congestion control techniques that outperform generic congestion control mechanisms and optimize the use of network bandwidth. Thus, to optimize performance, it is often desirable to combine the application's and the network's information.

Classic applications of the end-to-end argument have viewed the network as a monolithic entity that provides a single type or quality of service to all users. Recent trends in the Internet [46, 25], however, have been to provide service differentiation for different flows. Active networking allows *per-user* customizable services inside the network; this view is different from a set of higher-level generic functions being implemented on behalf of all users inside the network. By providing a more flexible network *interface*, active networking increases the likelihood that the service provided by the network is useful to its users. Thus, active networking is not antithetical to an end-to-end argument: instead, end-to-end arguments require that the interface to active networks be carefully designed such that only applications that benefit from specific functionality in the network are subject to its costs.

Active networking provides network designers with more choices for function placement. Successful system design still requires making the correct choices for function placement, which still depends on the details of the particular problem being solved. The key task for active networking in terms of the end-to-end argument, then, is to identify the cases in which the performance gains and enhanced capabilities justify the cost incurred in deploying and using an active networking architecture.

1.2 Previous work

The idea of introducing computation inside the communication subsystem is not new: Jon Postel and Bob Braden have argued [5] that RFCs 1, 5, and 8 [35, 92, 40], in which the earliest Internet router —the Interface Message Processor(IMP)— console Decode Encode Language (DEL) is proposed and described contain the fundamental concepts of mobile code systems (such as Java) and active networking. The earliest example of an *active* network in which the network interface was explicitly programmable is probably the Softnet project [107] at the University of Linkoping, Sweden in the early 1980s. Softnet used a multi-threaded dialect of the FORTH programming language to construct a distributed packet radio network. All of the nodes of the network were programmable during normal operation — the programmability extended from low level protocols such as medium access schemes to higher level protocols such as flow-control and routing. The idea of code mobility to implement distributed algorithms is also prevalent in agent-based systems [83] — however, these efforts have not been applied for programming of the base-network infrastructure.

1.2.1 Commercial Efforts

Commercial efforts for network programmability include the Internet Operating System (IOS) router operating system from Cisco Systems [6]. IOS provides an API for protocol development on Cisco routers. An end-host protocol development tool is Microsoft Routing and Remote Service Access (RRAS)[7] that provides an API for protocol development on the Microsoft Windows NT platform. However, both the IOS and RRAS provide facilities for development specific protocols; they do not provide a standardized interface for dynamic incorporation of new protocols into the network.

[5] E-mail messages to **ActiveNets_Wire** list (Message-Ids: 199807020040.*AA13752@gra.isi.edu*, 199807031758.*KAA06214@boreas.isi.edu*)

[6] See `http://www.cisco.com/univercd/cc/td/doc/pcat/186.htm`

[7] See `http://www.microsoft.com/NTServer/commserv/deployment/planguides/MoreRRAS.asp`

1.2.2 Open Signaling: Control Plane Flexibility for Circuit-Switched Networks

The open signaling community[8] has been working on the standardization of a flexible interface to the control plane for data networks. The *xbind* [69], a broadband kernel developed at Columbia University, is a programmable platform that supports the creation, deployment, and management of control plane abstractions (such as virtual circuits, virtual paths, multicast). Using game-theoretic ideas, *xbind* further provides mechanisms for mediating and arbitrating access to resources at switching nodes. The Tempest [100] framework, developed at the University of Lancaster, provides a programmable network environment by allowing dynamic introduction and instantiation of control architectures. Tempest *switchlets* are alternative control architectures that can be introduced into a operational ATM network. Tempest allows finer granularity modifications to the control plane as well. Using *connection closures* — application-specific control plane modifications — flexible control can be introduced into the network on a per-connection basis. Both the *xbind* and the Tempest systems result in a flexible networking infrastructure: both free the network provider from having to define and choose a single control architecture for all future network services. However, the flexibility provided by open signaling is necessarily constrained to the control plane and targeted towards circuit-switched data networks.

1.2.3 The Intelligent and Advanced Intelligent Networks

The telephone network is an example of a circuit switched network. Prior to the mid-1960s, the service logic (communication protocols and services) for the telephone network was hardwired in switches. Programmable logic was introduced in telephone switches in the mid-1960s in form of Stored Program Control switches. Common Channel Interoffice Signaling (CCIS) [37] was introduced in the 1970s and created

[8] See `http://comet.ctr.columbia.edu`

separate all-digital channels for signaling inside the network. CCIS networks evolved into an adjunct packet-switched network that carried all signaling traffic. The CCIS network was replaced by the Common Channel Signaling Network; the latest set of protocols used in the adjunct signaling network is the Signaling System number 7(SS7) [64]. However, even in the SS7 network, the early service logic was in the form of monolithic switching code. The Intelligent Network (IN) was introduced in the mid-1980s and marked the first time the service logic was separated from the switching system. Under the IN system, service specific "hooks" were incorporated into the switching path. Processing on behalf of a call could be transferred to Service Control Points (SCPs): SCPs provided additional functionality (such as toll-free 800 number service). In the IN, each SCP could provide only one specific capability. The Advanced Intelligent Network (AIN) is an extension to the IN that allows the selection of a range of services via three-digit triggers (through the use of Service Control Points that are service independent.)

The goals of the Intelligent Networks are analogous to active networks: provide an open interface such that new services can be rapidly introduced into the telephony infrastructure. The approach for in-network flexibility taken by the AIN is to redirect calls that require specific services towards specific service control points *outside* the switching fabric. The IN concept has also been applied to wireless telephone in the form of the Wireless Intelligent Network (WIN). WIN uses the same basic concepts as IN: trigger points are identified in the wireless call model and the service is provided via Service Control Points. The IN scheme for providing value-added services corresponds to an active networking architectures in which functionality beyond forwarding is provided outside the router (possibly on a co-located server farm). Note that such a scheme precludes flexibility on the data path — like open signaling, the flexibility provided by the AIN is only on the control path. Even though the AIN provides an open interface for service creation, services in an AIN are of coarse granularity — i.e. services are not added on a per-call basis. Instead, service creation is a non-trivial activity, subject to policy of the switch provider.

1.3 Thesis Outline

Active networks are distinguished in that users of the network can program the network infrastructure. Three key aspects of active networking are investigated in this thesis:

- **Architecture.** Can an active network be built without violating well-known design principles? Can an inter-network be designed such that only some of the nodes are active? Can an active node be designed to preserve the *fast-path* for users who are interested in raw throughput only?

- **Composition.** What programming models should be supported, what should the available abstractions be? How can support for composing network services be provided at active nodes?

- **Applications.** Is there a class of applications that benefit from the flexibility provided by active networks? Do the gains to such applications outweigh the risks of exposing an open network interface? Can support for such applications be scalably implemented inside the network?

This thesis answers the questions listed above. Specifically, we present:

- A set of principles that can be used to form the base for designing active networking architectures that support a range of programming interfaces. We have developed an active networking architecture that is consistent with these design principles. Based on this architecture, we have implemented the Odyssey Active Networking Environment that includes the Bowman Active Node Operating System and the CANEs Execution Environment.

- The CANEs *execution environment* for active networking that is based on the *slot processing* programming model. We present the implementation of the CANEs environment, its application programmer interface, and the signaling protocol used by CANEs to establish state and computation at active nodes.

- Examples of applications that benefit from the flexibility provided by active networks.

  We have introduced *application-specific congestion control* in which the network is provided instructions on how to process user data upon congestion. We have compared this scheme to traditional congestion control mechanisms, and have detailed its applicability and performance. As an example, we have investigated, in detail, application-specific congestion control applied to video flows. We have generalized our schemes to support multicast video distribution (with multiple senders), and compared our results to traditional mechanisms for supporting multicast video. We have proposed and studied the benefits of *application-specific queuing* in which data with similar characteristics and processing requirements are processed in aggregate at *active* routers.

  A substantial fraction of all network traffic today comes from applications in which clients retrieve objects from servers. The *caching* of objects in locations "close" to clients is an important technique for reducing both network traffic and response time for such applications. We have introduced *self-organizing wide-area network caches* in which caches are located at each network node and use active management strategies for organizing cache content. We have evaluated caching techniques using both simulation and a general analytic model for network caching.

1.4 Organization of Dissertation

The remainder of this dissertation presents our work in architectures, composition and applications of active networks. In each chapter, we provide a survey of relevant prior work.

In Chapter 2, we consider viable architectures for active networking. This chapter includes a discussion about possible network interfaces for active networking and the tradeoffs inherent in the choice of any particular interface. We present one particular architecture that we developed in conjunction with the DARPA active networking community.This architecture embodies multiple user-network interfaces: some of these interfaces may be optimized for performance while other interfaces could provide flexibility. As the network does not mandate one particular interface, our architecture allows the user of the active network to trade performance for flexibility and vice versa. Next, we present an implementation study of the Control on Demand active networking platform. In related work, we provide a survey of current implementations of Node Operating Systems for active networks and execution environments.

In Chapter 3, we present CANEs Active Networking Execution Environment. We present a discussion about composing of network services in active networks and discuss a particular service composition scheme: the *slot* programming model and its implementation within CANEs. We present the CANEs application programmer interface (API) and the CANEs User Interface (CUI) that is used for signaling by the CANEs EE. In related work, we survey existing composition mechanisms for active networks and compare the features offered by each to the composition mechanism provided by CANEs.

In Chapter 4, we present an application of active networks: application-specific congestion control (ASCC). We present the motivation behind and the main ideas of ASCC and show how ASCC can be applied to best-effort MPEG video delivery. We present ASCC algorithms for MPEG and describe an unicast implementation. We then show results of experiments with ASCC and other traditional discard algorithms on our unicast testbed. We then describe how unicast ASCC schemes can be extended for multicast and present a set of multicast simulation results. We present a discussion about network citizenship of ASCC like techniques and consider tradeoffs in the efficient implementation of ASCC. In related work, we compare ASCC to traditional congestion control schemes, provide a survey of other applications for active networking and discuss other applications of active networking. We close with a summary of our results and a discussion ASCC extensions for scalability.

In Chapter 5, we present self-organizing wide-area network caches. We present the motivation behind network caches, describe traditional caching schemes, and show how active networks can be used to organize contents of network caches. We present a set of simulations results that compare active caching schemes to their traditional counterparts and develop a general analytic model for network caching. In related work, we provide a survey of existing network caching schemes. We close with a summary of our results and a discussion of future work.

We conclude the dissertation in Chapter 6 with a summary of results, contributions and avenues for future work.

Chapter 2

Architectural Considerations for Active Networks

2.1 Introduction

The architecture[1] of a network specifies the *form* of the user-network interface, which, in turn, defines the "range of expression" afforded to users by the network. Users control and customize network processing using the interface defined by the network architecture.

For example, in todays Internet, the Internet Protocol (IP) header specifies the user-network interface. The range of expression afforded to users of the Internet is defined and constrained by the values in the IP header. Typically, upon receiving a packet, routers consult their internal routing tables and choose an outgoing interface based upon the destination address field of the IP header. If the outgoing interface is free, the packet is forwarded on the chosen interface. If the interface is busy, the packet is queued in a buffer associated with the interface. If this buffer is full, the packet is dropped. Multicast destination addresses can resolve in a set of outgoing interfaces; in this case, the basic forwarding mechanism is repeated for each interface in the set. The protocol definition for IP (RFC 791[84]) does define other mechanisms that can be used to control the processing of IP packets. Principal among these are the IP Type of Service (ToS) field and IP options. The original intent of the ToS bits in the IP header was to discriminate priority on a per-packet

[1] By architecture, we mean the user-network interface supported by the active network, and not the actual hardware configuration of a particular router.

basis. IP options are optional extensions of the IP header. They define specific computations that routers must perform before forwarding the packet. The result of these computations are carried in the packets that carry the options. In practice, the type of service field in the IP header is routinely ignored by IP routers and the most prevalent use for IP options are for network diagnostics. The typical extant of user control over an IP network's behavior is limited to the range of values that can be placed in the destination address field in the IP header.

From an user's perspective, it is desirable to investigate network architectures that provide more control than IP. For example, users may want to choose the routing scheme that is used to forward their packets or choose how their packets are discarded when an output buffer is full. From a network provider's perspective, it may make sense to treat different classes of users differently, e.g., the provider may want to charge extra to "first-class" users whose packets receive higher priority at routers. The high priority could result in "first-class" user's packets experiencing lower delay and loss rates. In the next section, we discuss a range of interfaces such flexible network architectures may entail.

2.1.1 A Spectrum of Interfaces

To the end user, the network architecture (and the consequent network interface) manifests itself in the form of a *network application programming interface*. The network API provides users access to the set of node behaviors that can be modified by the ends. In effect, the network API defines the *virtual machine* that is exported to users at each network node. The network API for the Internet Protocol comprises the language defined by the syntax and semantics of the IP header and its effect on the routers in the network. In traditional networks the capabilities of the virtual machine is fixed, and the expressive power of the language is limited. Active networks, on the other hand, provide a *programmable* network API.

There are several plausible extensions to the essentially fixed network API offered by the Internet. In context of the network API, these extensions can be classified in the following dimensions:

- **Abstractions and Expressiveness.**

  Two major components of a programmable network API is the set of abstractions and resources exposed by the active node, and the power of expression it allows its users in accessing and modifying these resources. The abstractions are in the form of the virtual machine (as defined by the network API), the expressiveness is in the form of a language using which the network API is programmed. It is important to note that the virtual machine abstractions and the expressive power of the language to access these abstractions must be considered together. Powerful abstractions alone without an expressive language to access and modify the node resources does not provide much flexibility. Similarly, the ability to compute powerful functions without the ability to access the node's resources is not of much use either. The possible set of abstractions and the expressiveness of the network interface ranges over a wide spectrum:

  - The network API is in the form of a set of user-supplied scalars that select services pre-defined in the network (e.g. select a particular routing schema). A richer interface can support scalar arguments to pre-defined services within the network (e.g. provide threshold values when routes should be changed).
  - A more flexible API may provide the user a mechanism to "program" the control path of the data. In this case, the power afforded to user's is controlled both by the set of abstractions exposed by the node and the language used to program these abstractions. With a powerful programming interface, such an interface can preserve the performance of the (unexposed) data path while providing flexibility in the control path.

An example of such an interface is the Active RSVP [82, 24] work at ISI, in which the network resource reservation protocol can be "programmed" by users. A further example is the Control-on-Demand router described in Section 2.3.

– An even more flexible interface would require programmability on the data path (i.e. the router would potentially execute user code on a per-packet basis). A form of data path programmability is provided by a network interface that allows users to supply *behaviors* to be plugged into specific points in the data path. For example, the user could supply a packet discard algorithm that the router executes to drop the user's packets. In this case, the designer of the network can choose exactly which abstractions —in both the data and the control path— to expose to its users. The power of the network API can further be limited by the choice of the language in which the behaviors are specified.

– The limiting generalization is a Turing-complete interface: the network nodes define a Turing complete virtual machine that provides access to all of the node's resources. User packets are programs written for this virtual machine. The network executes programs supplied by the user (e.g. all packets contain programs in a pre-defined representation which are executed by nodes upon which these packets are incident).

The obvious advantage of less expressive interfaces is that it constrains the possible node behaviors and simplifies correctness analysis. It is also more likely to admit fast-path optimization, e.g. through special-purpose hardware.

- **Statefulness.**

Certain classes of user algorithms can be efficiently achieved if the network API allows users to install state at network nodes [17]. Therefore, an important component of the network API is the ability to install and refer to pre-installed state at network nodes. Traditional network APIs (e.g. IP) are stateless — all of the requisite state to process a packet is carried in the packet itself. Some active networking architectures allow the installation of state. Such architectures have to provide mechanisms for isolation of state that belong to different users. Further mechanisms that allow the node to age and discard state must also be provided.

- **Granularity.**

  The network API may allow the user to program the virtual machine exposed by the network node on a per-packet basis. In this case, each packet (called a capsule [98]) carries some code that is executed by the network nodes. For performance reasons, it is beneficial to consider other granularities of control, e.g. amortize the cost of code transportation, loading and context switch over the lifetime of a end-to-end TCP connection.

- **Scope.**

  The scope of the network API defines the set of packets that are affected by the processing triggered by a packet. Trivially, processing on behalf of a packet affects how the individual packet is treated in the network. Depending on the network API, such processing may install state and computations that affect other packets e.g. subsequent packets belonging to the same user. Some network APIs allow global updates in node behavior; processing on behalf of a packet changes the way the node processes all subsequent packets.

2.1.2 Tradeoffs (or Which Network API should the Active Network Support?)

Tradeoffs are introduced by each network API. In recent years, packet forwarding paths in routers have been optimized down to machine instructions [81, 9]. It is difficult to implement flexible network APIs (e.g. one that allows user code to be instantiated in the data path) that perform within even two orders of magnitude of such optimized packet forwarders [103, 55]. On the other hand, flexible network APIs can enable a whole new range of applications [70]. Eventually, the performance of a system must be evaluated in terms of user and application-dependent metrics. Some users and applications (e.g. file transfer) merely want the fastest possible network connection; others (e.g. on-line media transfer) may require the network to sacrifice raw bits per second performance for further processing upon congestion [19].

There is also a potential tradeoff between the safety of the network versus flexibility allowed of the network API. The node and network level safety of traditional networking protocols have been analyzed for decades [32, 97, 34, 101]. With active networking, however, network APIs can potentially introduce new protocols and code in the network on a per-user basis. If the EE and the node OS allow users unmitigated access to resources then flexible network APIs pose a definite threat to the safety of the active node and the entire network. However, unlike the tradeoff between performance and flexibility, the tradeoff between safety and flexibility is largely a function of policy, i.e. the tradeoff between safety and flexibility need not be inherent. It is possible provably implement safe but powerful interfaces [12, 80, 13]. In many cases, however, safe interfaces come at a performance cost [103, 55].

Ultimately, the choice between flexibility, safety and performance is an engineering decision. Active networking, possibly at a cost of safety and performance, provides a platform for in-network flexibility. The eventual selection of the *best* network API is still an user and application-dependent choice. It does not behoove a global active networking architecture to specify one particular interface or set of abstractions. In the following section, we describe an active network node architecture that, in its design, explicitly acknowledges the tradeoffs between flexibility, safety, and performance.

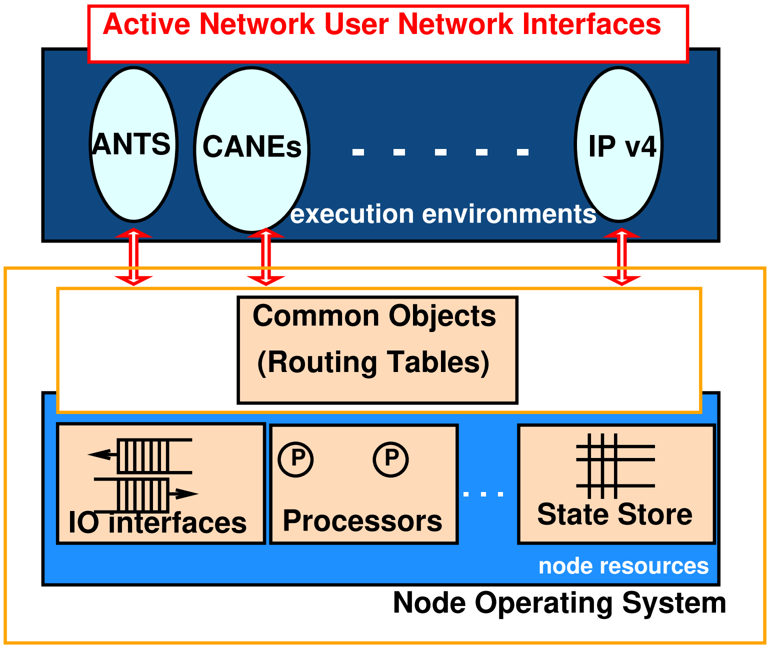

Figure 2: Components of the Active Networking Architecture. Each EE implements one particular network API. Access to node resources are mediated by the Node OS.

2.2 Architecture Overview

In this section we present an overview of the node architecture developed in the DARPA active networks program. The architecture defines the basic packet processing cycle at each active node. It does not decree a particular network API or the set of abstractions that must be supported by the active network API. Instead, the node architecture is explicitly designed to support multiple network APIs simultaneously. This allows different network APIs that offer different levels of performance and flexibility to co-exist in the active network. This approach also provides an evolutionary path for active networks as traditional network architectures (such as IP version 4 or IP version 6) can be accommodated as just another network API.

Functionality at the active network node is divided between the *Execution Environments* (EEs) and the *Node Operating System* (NodeOS). The general organization of these components is shown in Figure 2. In terms of the discussion in Section 2.1, each particular EE is responsible for implementing a particular network API. Each EE then maps its own exported set of abstractions to the set of local resources at the node. The NodeOS manages and mediates access to local node resources by EEs.

2.2.1 Execution Environments

An Execution Environments (EE) implements a specific user-network interface. Users of the active network must select one EE which is used to transport and process their packets. All user access to node resources is provided through an EE, i.e. user code executes at a node within the context of an EE. Accesses to resources at a node (like processor, memory, and transmission links) are through interfaces provided by EEs. The interface to the node resources provided by an EE can simply be a bijective map onto analogous interfaces provided by the Node OS. However, the EE-specific interfaces provides the EEs an opportunity to provide functionality (e.g. EE-specific resource bounding or credential checking) beyond what is provided by the node OS. The architecture does not specify limits on expressiveness or the abstractions provided by EEs. Thus, EEs may provide a very simple service that can be statelessly controlled through user-provided parameters, or it may implement an interpreter for a powerful, stateful programming language, or something in between.

2.2.2 Node Operating System

The node architecture allows multiple EEs to reside at a single node. All of the node resources — processor, link bandwidth and state — are accessed through EEs. The Node OS mediates access to these resources. The Node OS is also responsible for enforcement of node security policies.

The node OS provides the basic set of abstractions for communication, computation, and storage that are used by the EEs to implement individual network APIs. These abstractions are exported to EEs via the node OS Interface [42].

- Communication resources are abstracted as "channels" by the Node OS. Channels are equivalent to logical interfaces: they specify a physical interface at the node along with a protocol processing component. Thus, the Node OS can support channels various types, e.g. TCP/IP, IP/ATM, etc.

- Computation is modeled by "flows". A flow is analogous to a *process* in Unix. Flows can create specific channels, and can subscribe to subsets packets on specific channels. The set of packets that are processed by a flow is constrained by a "packet filter" specified by each flow. Packet filters consist of a set of predicates that are matched against headers of incoming packets. Note that per-user flows are created in context of EEs. Code executed within a flow by an user thus depends entirely on the particular network API exported by a specific EE.

- State is modeled via the "state store". A state store is analogous to a tuple space [79, 29]. The state store provides a mechanism to store and retrieve tagged data. Ephemerality of, access to, and resource bounds on the stored state is not currently specified by the architecture.

- The node OS may provide support for "common objects" that are likely to be useful to multiple EEs. Common objects include but are not limited to abstractions for routing (routing tables) and packet processing (packet rings).

The logical flow of packets through an active node is shown in Figure 3. When an active node receives a packet over a physical link, it classifies the packet based on the packet's contents (i.e. headers) to belong to a specific channel (i.e. logical interface); each packet is either assigned to an existing channel or discarded.

The mapping of incoming packets to flows is controlled by the filter specified by the user when it subscribes to the channel. In the typical case, an EE (on behalf of a user) requests subscription to a set of channels matching a certain pattern of headers, e.g. a certain Ethernet type or combination of IP protocol and TCP port

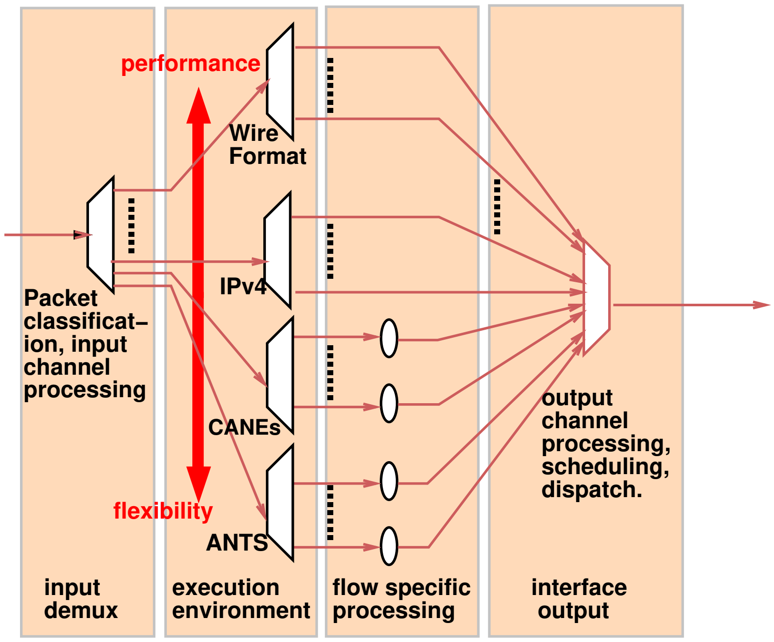

Figure 3: Packet flow through an active node. Incoming packets are demultiplexed to EEs. Depending on the EE, packets may receive individualized treatment. Details of input and output channel processing is not shown in this figure.

numbers. With sufficient privileges, EEs can create entirely new channels as well.

2.2.3 Active Network Encapsulation Protocol

The Active Network Encapsulation Protocol [45] provides another mechanism to route user packets to particular EEs. The ANEP header includes a "Type Identifier" field; well-known Type IDs are assigned to specific execution environments. (Presently this assignment is handled by the Active Network Assigned Number Authority.) If a particular EE is present at a node, packets containing a valid ANEP header with the appropriate Type ID (encapsulated in a supported protocol stack) will be routed to the appropriate EE. Further demultiplexing may occur inside the EE to route the packet to a specific user.

2.2.4 Interfaces and Standardization

The primary interfaces in this architecture are the user-EE interface (the network API) and the EE-NodeOS interface. As stated above, the architecture does not decree form of the network API exported by any EE. Thus network APIs can vary in abstractions, expressiveness, and over time. The EE-NodeOS interface need not be identical from node to node; all that is required is for each node to provide a standard set of basic services to EEs.

It is instructive to map the architecture and the interfaces to the structure of the UNIX operating system. The node OS interface maps to the UNIX system call interface, EEs to shells, and network APIs to shell languages. Just as the UNIX system call interface does not dictate a particular language be implemented by each shell, the node OS is indifferent to the network API exported by an EE. All that is required is that the EE provide mapping back to the abstractions provided by the node OS. Further, note that like the UNIX system call interface, the node OS interface need not be exactly the same everywhere. It is possible to re-code EEs for different interfaces as long as the basic abstractions remain the same. Thus, EEs further serve to isolate users from the vagaries of particular node OS' and interfaces that may be present at individual nodes in an active network. The UNIX analogy is useful in one more respect: just as there are only a few shells in UNIX, we expect only a few EEs to be present at an active node. Creation of an EE should be construed as a non-trivial effort; installation of EEs may require the active node be re-initialized.

2.3 Implementation Study: Control on Demand

In this section, we present the implementation of Control on Demand, a active router platform. The CoD platform conforms to architecture presented in the previous section. It is a software router platform based upon GNU/Linux as the Node

OS. Architecturally, CoD network API provides flexibility on the control path with limited ability to affect the data path.

2.3.1 Control on Demand

The common data forwarding path in current routers has been analyzed and optimized for performance for decades[81]. Packets that can be forwarded via this common, optimized path are said to follow a *fast path* through the router. Unfortunately, forwarding hardware and the fast path logic has been coupled to the extent that adding a instructions to the forwarding path is enough to dramatically reduce router performance. A major problem facing proposals for any *enhanced* packet processing in current routers is that, invariably, such processing will be done out of the fast path which will lead to orders of magnitude degradation in performance. Such a fate, for example, has befallen the IPv4 options and IP options are not used in general applications. In this section, we describe Control on Demand, an environment for active networking that is designed to preserve the fast path in routers while still providing facility for per-user processing.

Control on Demand (CoD) [57, 59] is an execution environment for active networking which was developed at AT&T Labs. The CoD network API allows users to inject specific *controllers* to customize processing. Controllers are code pieces that implement a specific functionality. In the implementation described, controllers are written in C, and are transported as native shared objects[2].

The CoD network API provides Turing complete access to control path abstractions. There is provision for limited access to data path abstractions as well; however, the fundamental tenet is not to introduce extra processing on the data forwarding path. Thus, the CoD network API is constrained such that only specific parts of the node state can be affected by a controller. The user-supplied controllers can compute arbitrary functions, but their ability to affect node state is constrained

[2]While this method provides the maximum software-based performance, the issues of protection are largely ignored.

via the CoD API.

A major problems with active networking approaches before CoD [103, 55] is that the "active" code was executed on a per-packet basis — i.e. the user-specific computation was scheduled each time a packet arrived. Thus, the user-code activations were irrevocably coupled with packet input. This approach does not scale, especially when the router is either processor or bandwidth congested. The per packet user code processing precluded previous approaches from maintaining a fast common path during the tenure of active packets at active nodes. The CoD scheme decouples scheduling of user-specific code and packet arrivals. The per-user code runs as a preemptive computation on the CoD node and processes its incoming queue each time it is scheduled. Most packets stay on the fast path and are not necessarily actively processed. (The user code may, however, specify a subset of packets be copied out of the forwarding path for detailed processing.) In the following section, we discuss the fundamental concepts CoD is based on. In Section 2.3.3 we describe the logical structure of the CoD platform. Details of our implementation of CoD over Linux are given in Section 2.3.3.

2.3.2 CoD: Fundamental Concepts

Due to efficiency considerations, the CoD design deviates from the "code per packet" or "capsule" model of active networking. Further, the CoD EE design specifically considers overheads incurred in fetching code and constrains on processor availability at routers. Thus, the fundamental concepts that CoD is based upon are:

- **Flow-based application of control.**

  Instead of a program being associated with each packet, the application-specific control is associated with a *end to end packet flow*[3] of packets under

[3]An end-to-end packet flow is a set of associated packets identified using an opaque identifier on an end-to-end basis

CoD. In an appeal to brevity, in the rest of this section we will use the abbreviation "flow" to signify a CoD end-to-end packet flow. This should not be confused with the computation abstraction "flow" provided by the node OS.

Thus *investment* in state and processing is made on a per- flow basis, and is amortized over the lifetime of the flow. Particular controllers required for each flow are fetched from a code server and cached at each node. The signaling required to trigger the code fetch can either be in-band or out of band. In our implementation, we use in-band signaling; packets contains a CoD header that identifies the particular controller that is required by the flow. If the controller is not already present at a CoD router where such a packet is received, the code is fetched and cached. Note that the packets are forwarded towards the destination while the code is being fetched.

- **Enhancement control.**

  By design, CoD controllers are performance *enhancements*. End to end application semantics are not modified by the presence/absence of the controller. In all cases, packets belonging to a flow are default forwarded towards their destination (even without the user-specific controller being present). Thus, at the very least, CoD nodes act as best effort routers.

  This paradigm has the following advantages: (1) controllers can be installed asynchronously, (2) controller do not have to be present during the entire lifetime of the flow and (3) it provides a deployment/upgrade path. Consider the following example: A flow requests a CoD controller that provides a specific packet-discard algorithm. If the requested controller is not present at the node, the flow's packets are default forwarded towards their destination and will be tail dropped if congested. After the controller has been fetched, it can be installed on behalf of the flow at the CoD router (after some packets of the flow have already traversed the CoD router). Further, packets in the flow can encounter non CoD routers in the path which will also default forward the packets towards the destination. Thus, CoD nodes can co-exist with traditional routers; CoD aware flows can be run on networks where only a subset of the nodes implement the enhancement control.

- **Asynchronous best effort Control**

  At a CoD node, packets are forwarded on a "data path" that does not necessarily involve the processor on which user controllers execute. User controllers have a *window of opportunity* to act on packets while they are enqueued at the CoD node waiting for transmission. In general, packets are not deviated out of their fast path and enqueued in a processor queue such that the user-specified control may be applied on a per-packet basis. Fundamentally, in CoD, packet scheduling is independent of the scheduling of the user controllers. Thus, at a CoD node, the processor on which the controllers execute can become a bottleneck but this would not affect the packet forwarding rate (assuming the packet forwarding engine is independent of the congested processor). Thus, even if the user controller is present, it is applied on a *best-effort* basis; i.e. the scheduling of the flow-specific controller is not guaranteed. Hence, even when installed at a node, the flow specific controller can *miss* packets. The interface does allow for a tagged subset of packets to be copied out of the data path and reliably enqueued in a flow-specific queue for subsequent action by the flow-specific controller.

2.3.3 The CoD Node Architecture

Figure 4 shows the overall architecture of an active networking node supporting the CoD EE. The major components of the CoD EE are a "meta-controller" and a set of flow-specific controllers. The CoD EE assumes that the Node OS provides access to a forwarding engine capable of per-flow queuing. Thus, there are two separate interfaces at the node: (1) the Node OS–CoD EE interface that provides access to the forwarding engine and other node resources and (2) the CoD network

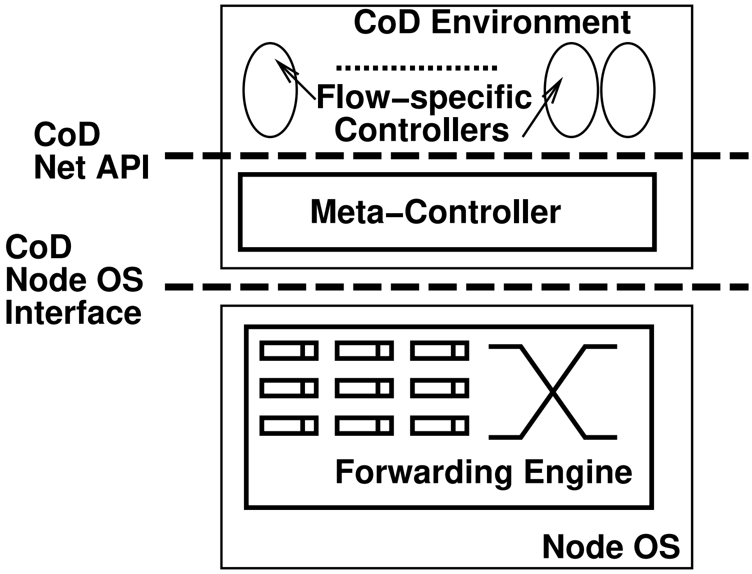

Figure 4: CoD Architecture showing the CoD EE and the Node OS. Note that the Node OS maintains per-flow queues and the forwarding engine.

API that the flow-specific controllers use to provide per-flow control.

CoD packets carry a CoD header that specifies a flow identifier and may request a specific controller for flow processing. Using the flow identifier, incoming packets are classified by the Node OS into a per-flow queue. A copy of each unclassified CoD packet is given to the meta-controller which then initiates processing on a new flow. The original packet is transmitted by the forwarding engine towards its destination. Decoupled from the forwarding engine, flow-specific controllers are scheduled and can act on packets queued in the per-flow queues.

In the rest of this section we present specifics of each component and each interface.

- **Forwarding Engine**

  The CoD architecture assumes a (hardware) forwarding engine that is capable of per-flow queuing. Thus, after a packet has been classified into a per-flow queue, the forwarding engine is used to transmit the packet out of the appropriate interface(s). Inherent to the per-flow queues are a packet scheduler, that can select packets to schedule out of each flow. CoD further assumes an interface to the packet scheduler, using which the flow-specific controllers can schedule their own flows.

- **The meta-controller**

  The meta-controller is a resident default controller that is responsible for fetching and initiating the various per-flow controllers. Upon detection of a new flow, the meta-controller is invoked, which parses the CoD header. Unless the flow specific controller is already cached at the node, the meta-controller downloads the requested controller and subsequently initiates the requested controller. The meta-controller is responsible for handling error conditions such as an unavailable or faulty flow-specific controller.

- **The flow-specific controllers**

  The flow-specific controllers consist of code downloaded and dynamically linked into the router. In general, these controllers may compute Turing complete functions; however, they access the node state through the CoD network API.

  CoD controllers have to implement an efficient mechanism for synchronizing themselves to the current state of the flow. To provide efficient flow-specific processing, we use the following mechanisms:

  - **Application Level Framing** The efficiency of the flow-specific controllers depend largely upon the processing required to synchronize the controller with the flow state, and to parse the individual objects that constitute the flow. We use application level framing [31, 49] to efficiently solve both these problems.

  - **Frame Peeking** Once the flow is properly labeled, we propose the *frame peeking* mechanism to efficiently expose the application level framing to the controllers. The flow-specific controllers specify a filter that selects a subset of all packets in its queue. Only the selected packets are then forwarded to the controller for processing. As most CoD controllers do not change the payload of packets, all the information needed for control processing is available in the packet headers. The frame peeking mechanism allows for specific headers to be forwarded to each controller.

A CoD router has the following interfaces:

- **The Node OS–CoD interface**

  This is an implementation-dependent interface that the meta-controller uses to communicate with the forwarding engine. A version of this interface is being standardized by the DARPA community [42]. In general, as the forwarding engine can be implemented either in software or in hardware (or partially in both), this interface may change with each CoD implementation. As such, we do not specify the particulars of this interface — though the semantics of this interface must allow the CoD network API to be implemented.

- **The CoD network API**

  The CoD network API is used by flow-specific code controllers to provide per-flow control. This API is specified (see Section 1) and allows the flow-specific controllers to be written and linked against the CoD API independent of the underlying Node OS implementation.

  The design of the CoD network API is an attempt at a balance in the performance versus flexibility tradeoff. The CoD API is the only mechanism by which the node resources are exposed to users, thus it should not be overly restrictive. On the other hand, an overly general interface is difficult to optimize, and analyze. Further, extremely powerful interfaces introduce strong and undesirable coupling between the controllers and the forwarding engine as they often must introduce user code in the forwarding path.

  The CoD network API is listed in Table 1. An opaque flow identifier is used to classify packets upon receipt. If a new CoD flow is detected, its flow state

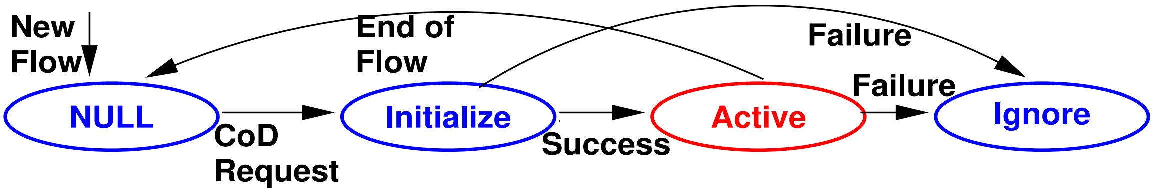

Figure 5: CoD Flow Classification. Each CoD packet carries a flow identifier which is used to classify packets and demultiplex them into per-flow queues.

is set to *Null*, a copy of the packet is sent to the meta-controller. The meta-controller then creates a new per-flow queue in the forwarding engine for this flow and using the CHANGE-FLOW-STATE primitive sets the flow state to *Initialize*. When the flow-specific controller for this new flow has been initialized, the flow state is set to *Active*. Upon failures to fetch or initialize the requisite controller, the flow state is set to *Ignore*. Regardless of the flow state, packets for this flow are enqueued in the per-flow queue and serviced by the output scheduler. If possible, the output interfaces for each flow is cached in a per-flow control block (maintained by the forwarding engine) — this obviates the need to repeat the routing computation for each packet in a flow.

Flow specific controllers can use the EXPOSE-QUEUE primitive to examine parts of specific packets enqueued in its per-flow queue. The SET-SCHEDULE primitive is used to schedule output. The rate at which a queue is served is set for $Byte_i$ bytes. The last specified rate remains in effect till the next SET-SCHEDULE call. Disposition of a set of packets can be set via the SET-PACKET-STATE primitive. This primitive is mainly used by controllers to delete matching sets of enqueued packets.

CoD Implementation

In this section, we summarize an implementation of the CoD architecture. We use IPv6 as the base transport layer, and a modified Linux kernel as the Node OS. The

Table 1: The CoD network API. The calls for reading and writing packets from channels is inherited from the Node OS and is not shown here.

Command	Parameters	Effect
`CHANGE-FLOW-STATE`	Flow, State	Set flow state
`EXPOSE-QUEUE`	Start Offset, Length, *Which*	`returns` {Queue Info., # of matches, {{matches}}
`SET-SCHEDULE`	$\{Byte_i, Rate_i\}$	After $Byte_i$ bytes served, output rate set to $Rate_i$
`SET-PACKET-STATE`	$\{Packet_i,\ State_i\}$	Set packet state (e.g. `Discard`)

salient features of our implementation are:

- **Extensions to the Linux IPv6 Kernel** Some extensions to the Linux kernel are due to the per-flow assumptions of CoD, and are useful in their own right (i.e. they add functionality to the base IPv6 subsystem). These include per-flow queuing, per-flow scheduling, and route caching for flows with the flow identifier set so that the output interface is only computed once. The current implementation uses a simple FIFO scheduler that exports the necessary interface for CoD, and may eventually be replaced by more complicated packet schedulers.

- CoD **Extensions to IPv6** The IPv6 incoming packet interrupt routine was modified to add one comparison to check for a valid CoD header. This is equivalent to a rudimentary ANEP demultiplex. If a valid CoD header is found, the packet is forwarded to a CoD demuliplexer that uses its flow classification mechanism to further process the packet.

- **The meta-controller** Under the Linux implementation, the meta-controller is segmented in two parts, an in-kernel part, and a user-space controller. The

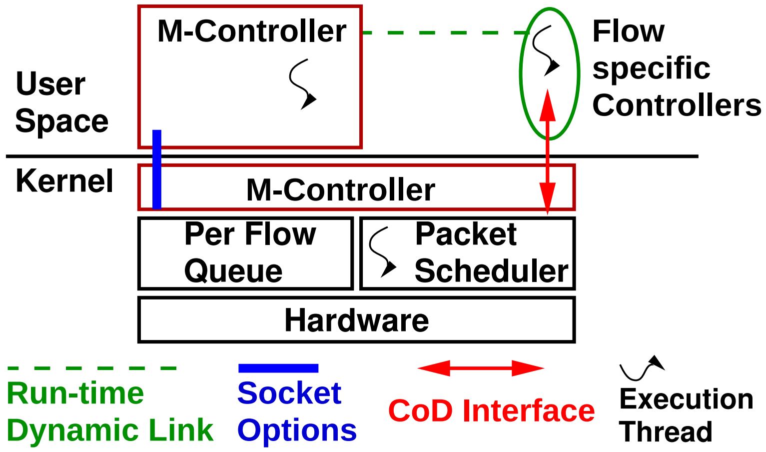

Figure 6: Implementation of the CoD EE using Linux as the Node OS. Parts of the M-controller, per-flow queuing and per-flow packet scheduling was added to the base Linux kernel.

modified IP interrupt routine passes CoD packets to the kernel-resident meta-controller. The meta-controller accesses the flow control block (indexed by the flow identifier), and possibly queues the packet in the flow-specific queue. If the packet is the beginning of a new CoD flow, it is copied and passed to the user-space meta-controller which then initiates processing on the new flow. The user-space meta-controller is responsible for downloading and initiating the flow-specific controllers. Upon detection of an error on the part of a flow-specific controller, the user meta-controller changes the state of the affected flow accordingly.

- **The flow-specific controllers** The flow specific controllers are compiled against a library that implements the CoD kernel-interface. Each flow-specific controller is dynamically linked in by the meta-controller during flow initialization. Each controller exports a well-known initialization routine that is run by the meta-controller to initialize the flow. The controllers are provided with processor scheduling primitives that they use for their own scheduling. We implemented flow-specific controllers for application-specific processing and smoothing of MPEG and multi-description encoded video streams.

2.4 Related Work

The related work in active networking architectures fall in two broad categories: different execution environments and different Node Operating Systems.

2.4.1 Execution Environments

There are several EEs that have been developed for active networks. They cover a broad range of network APIs.

CANEs

The Composable Active Networking Elements (CANEs) EE was developed at the Georgia Institute of Technology [27] as part of the CANEs active networking project. The CANEs EE is designed to create composite network services out of orthagonal building blocks. Computations within the CANEs EE consist of two parts: the underlying program and a set of injected programs. Underlying programs are supplied by the node provider and represent per-packet processing that is common to all packets in an user's set of packets. The underlying program exports a set of trigger points (called *slots*) where user-specific computations (in the form of injected programs) can be bound. As the service provided by the network is a composition of the service provided by the underlying programs and the set of bound injected programs, the power of the network API exported by CANEs can be controlled by the exported underlying program and its defined slots.

The CANEs EE is not dependent on a particular language. Instead it is based upon a formal model that supports independent reasoning about the correctness of both the underlying active network platform and the algorithms injected into it, in a manner that admits the full range of possible programming interfaces. The reasoning is based upon a specialized form of program composition that captures the interaction between the underlying platform and the user-injected program and allows properties of each to be preserved. The formal model and implementation details of the CANEs EE is described in Chapter 3.

ANTS

The Active Node Transport System (ANTS) toolkit [103] is an EE developed at Massachusetts Institute of Technology. ANTS is a toolkit for protocol and service deployment and is built atop the Java Virtual Machine (JVM). The ANTS Network API consists of the Java Virtual Machine augmented with the ANTS class, which implements methods permitting user code to be decoded and interpreted. User code is transmitted in packets (called *capsules*) in form of Java byte-code which are decoded and interpreted by the JVM. The node resources available to users within ANTS is defined by the JVM. Further, ANTS uses the JVM's sandbox execution model to provide safety at active nodes.

User code for the ANTS EE is written in the form of Java classes. For transferring code, ANTS provides a hop by hop "code load" mechanism that is used to demand pull user code into the ANTS EE. It also provides a secure hash based fingerprinting mechanism to identify particular capsules. Once the code for a particular capsule has been downloaded to a node, ANTS provides mechanisms to recover the transmitted Java classes and for executing them within the JVM.

ANTS represents an interesting extreme for active networking. Java provides a Turing complete network API. However, the abstractions provided by the EE are provided is essentially the ones provided for the JVM and do not the requisite abstractions for network programmers. For example, ANTS does not provide timers which are critical for protocol programming, nor does it have a concept of buffers or links that map to buffers of links that may be present at an active node. ANTS admirably leverages the mobile code facilities of Java to provide deployment of new protocols in an active network; however, it is also beset by the performance woes of

Java.

SwitchWare

The Switchware active node has been developed at the University of Pennsylvania [1], uses a language called PLAN (Packet Language for Active Networks) [55] for accessing active network resources. Switchware is based upon a three level programming model for active networks. At the base is a secure router infrastructure over which execute active extensions. At the top is the Switchware network API in the form of active packets programmed using PLAN.

The base layer secure router infrastructure is analogous to a secure node OS that is specific to Switchware. It provides mechanisms for authentication and secure bootstrapping. Active extensions named *switchlets* are heavy weight extensions of the router infrastructure and can (presumably) be implemented in any language. Switchlets are not mobile code; instead they are installed at an active using an unspecified out-of-band mechanism. Active packets form the highest (user-visible) layer of the Switchware architecture and carry PLAN code. PLAN is a language that is designed for "safe" access to the network's resources. By design, PLAN only supports straight-line programs, and does not support installation of any state at the node. Thus, both the computing and state usage of any PLAN program is bounded. PLAN allows the invocation and sequential composition of switchlets.

Architecturally, the Switchware network API is constrained by the PLAN language. However abstractions provided by the node (in form of Switchlets) that can be manipulated by PLAN programs are Turing complete. Switchlets are implemented in CaML which is a dialect of ML [2]. PLAN is currently implemented in Java (and thus inherits its mobile code facility and performance woes). An active network testbed named PLANet in which each node executes PLAN has been demonstrated [56].

Netscript

Netscript is an EE developed at Columbia University [106] that offers the Netscript language as the network API. Analogous to PLAN, Netscript is another language developed specifically for Active Networks. Netscript allows STREAMS-type [89] composition of mobile agents that are executed within the Netscript EE. Netscript programs can be distributed over an virtual (overlay) network connected via virtual links and may consist of multiple collaborating agents at each node. Architecturally, the Netscript net API is Turing complete; Netscript has been implemented within the JVM.

SmartPackets

The SmartPackets EE has been developed at BBN [95]. The network API is in the form of an active network specific language, Sprocket and the SmartPackets EE executes a Sprocket interpreter named Spanner. Sprocket is a language developed for writing compact programs that can fit in a few packets and is targeted for use in network management applications.

2.4.2 Node Operating Systems

A node OS interface is being standardized within the DARPA active networking community [42]. The primary abstractions consist of the flow for computation, channel for communication and the state store for memory.

Scout

The most prominent Node OS work is the adaptation of the Scout OS [78] for active networking. Because the standard JVM does not support access to transmission resources at a sufficiently low level, implementations of ANTS (or any other active network API based on Java) on standard platforms cannot support quality-of-service capabilities, and are limited to the basic network capabilities provided by Java.

In order to overcome the native resource access limitations of the JVM, a new implementation of the JVM called Joust [47], which supports lower-level abstractions for real-time scheduling, is being developed at Arizona.

Bowman

The Bowman OS has been developed by the CANEs group at Georgia Tech as a node OS for the CANEs EE. Strictly speaking, Bowman is not a OS but an OS support layer for active networking EEs. Bowman is designed to be layered over a host OS that will provide platform-specific low level functionality such as memory mapping, threads, and access to network interfaces. The Bowman interface [77] conforms to the active networking node OS interface [42]. The Bowman system call interface can be extended dynamically to add *extensions* that provide functionality beyond what is provided by Bowman. Bowman also supports a virtual network scheme [17] within the OS.

Bowman was originally implemented in the Solaris Operating Environment (SunOS version 5.x) and is natively multi-threaded and multi-processor safe and can leverage the real-time capabilities of Solaris. Versions have now been ported to the Linux (version 2.2.x) Operating System.

OS-Kit, Janos and the Exokernel

Developed as part of the Flux group at the University of Utah, the OSKit is a framework and set of modularized library code that aims facilitate creation of new OSs. Janos: a Java-based Active network OS is being developed at Utah using the OS kit[4]. The ANTS EE is being ported to Janos. There are efforts at TIS to port the Exokernel [43, 67] as a node OS [94].

[4] See `http://www.cs.utah.edu/projects/flux/janos/summary.html`

Acknowledgment

The active network architecture described in this chapter has been influenced by discussions with Bob Braden, John Guttag, Larry Peterson, James Sterbenz, and David Wetherall and has in spirit been adopted by the DARPA active networking community [41, 27]. CoD was developed at AT&T Labs, Florham Park, New Jersey in conjunction with Gísli Hjálmtýsson, Jennifer Rexford, and Albert Greenberg prior to the general active networking architecture presented here. Its design and implementation was a pre-cursor to and later influenced the design and understanding of the more general DARPA architecture.

Chapter 3

Composition in Active Networks

The goal of active networking is to ease the deployment of new network services.The active network API must provide a facility to install new services in the network; in order to support complex services, the API must also provide support for the process of service creation. An important support feature of a network API is the ability to *compose* services from building blocks (or *components*). Viable active network APIs must contain a *composition mechanism* that can be used to create a *composite service* from components. Composition of network services has the usual positive properties of modular design: services need not be built from scratch each time and robust components can be developed incrementally. Further, a composition mechanism may also be used to constrain the set of composite services that can be created, possibly making it easier to reason about the correctness of the overall service and interactions between individual components.

Active network APIs can provide varying degrees of support for composing services. For example, the composite service provided by the network API may only be the selection of specific services from a fixed set. On the other extreme, the network API may be in the form of a Turing complete programming language that does not impose any structure on user computations; in this case, an essentially infinite set of services may be composed by using the constructs provided by the network API programming language. However, the network API can also provide more structured approaches for composition, e.g. object-orientation, a software bus, or event-driven composition. Using object-oriented techniques, static composition can be achieved by class inheritance in which sub-classes are automatically generated using virtual methods provided by users[60, 8]. These virtual methods override their abstract counterparts in the base classes exported by the active node; this is essentially the composition mechanism supported by the environments that use Java (or some other object-oriented language) as their network API. The network API may, however, impose more structure on service composition by mandating how service components communicate. One example of this type composition is a software bus: an agent that encapsulates the interface details for communication between modules within an application. The network service would be composed using a set of code modules that communicated using the software bus. An example of a software bus system is Polylith [85]; composite services in Polylith are created by statically associating a set of code modules that communicate using a software bus. A network API may also support *dynamic* composition in which the service offered can change depending on the state of the node. One mechanism to achieve such dynamic composition is by using event-driven computations. Particular events are signaled (or raised) during specific times at run-time. Code modules associated with specific events are executed when each event is raised.

In this chapter, we describe the CANEs execution environment. The CANEs EE uses the *slot processing model* for composing of network services. In the next section, we present an overview of the slot processing model and its formal basis. The CANEs EE is introduced in Section 3.2 and its implementation details are given in Section 3.3. We present a survey of related work in composition of active network services in Section 3.5.

3.1 Overview of Slots

In this section, we describe the slot processing model, which is the basis for the CANEs EE. In this model, processing at the node comprises of two parts: a fixed part that represents the uniform processing applied to every packet, and a variable part that depends on the packet contents and the node state. The fixed part corresponds to the virtual machine that is exported by the network architecture; the variable part is encoded using the network API and is *input* to the node virtual machine. For an user, the network's behavior consists of the actions of the node virtual machine acting on the input carried in the user's packets. Depending on the facilities provided by the network API, node behavior can be tailored by providing suitable input to each node virtual machine on the network path.

We refer to the fixed part of the node behavior as the *underlying program* running at each node. User specific functionality that forms the variable part of packet processing is called the *injected program*. In our terminology, users *inject* programs into the network using the network API. Mechanisms to inject user code into the network must address the issues of the form and transport of the injected code to specific nodes in the active network. In order to reason about the injected code in context of an underlying program, we need to model the interaction between the injected and the underlying program. Note that users of the network are not allowed to install new underlying programs. Therefore, the form of the injected program and the interaction between the underlying and injected programs essentially defines the programming interface supported by the network. In order to reason about the composite service offered by the network, the programming interface must formally express the interaction between the underlying program and the injected program, i.e. the programming interface must define how control flows between the injected and the underlying programs, and how state is shared between the two programs? Two extreme points in the range of possible programming interfaces are as follows:

- The programming interface is Turing-complete, and the injected program completely replaces the existing node program. No properties of the underlying program are preserved by such injection, and reasoning about the composite is equivalent to the reasoning about the injected program. Reasoning about the network on the whole may not be so easy, as that would involve reasoning about the entire space of all combinations of all injected and underlying programs (as the "old" underlying program may be executing at some nodes, and the "new" injected program at others).

- The programming interface consists of identifying choices the underlying program should make, at well-defined points in its computation. In effect, the underlying program offers a *menu* from which the user can select to customize processing inside the network. Reasoning about the composite of the injected program and the underlying program is equivalent to reasoning only about the underlying program. Further, reasoning about the network only involves reasoning about the underlying program. However, the flexibility of the user programs is limited to choosing what is already present in the underlying program.

Our goal is to (1) support customization and programmability of network services and (2) deduce global formal properties of the service composed of the underlying and injected programs. A satisfactory solution must support composition of complex services from simpler ones; allow abstraction and modularity; and support dynamic injection of injected programs. In case of a Turing-complete programming interface, it is difficult to reason about the behavior of network services; in the second case, the network interface is not flexible enough. The slot processing model is a solution that considers cases in which both the injected program is interesting and the network maintains specified properties. Further, it supports reasoning —in the form of invariants and leads-to properties— that is intuitive and can be used to derive global properties about composite services.

Composition under the slot processing model achieved in two steps — first the user selects an "underlying program" that is executed on behalf of the user. The underlying program is offered by the node, and has provable properties. The user then provides (or selects) a set of "injected programs", which correspond to individual code modules, that can be used to customize the underlying program.

In effect, slots identify *specific points* in the underlying program where injected code may be executed. Slots are also the mechanism by which the underlying program asserts properties that hold at the node. We say that injected code is “bound” to specific processing slots, and becomes eligible for execution when appropriate slots are “raised”. Underlying programs define and raise specific slots, and composite services are formed by binding user-specified code to specific slots. More than one injected program may be bound to the same slot— the order of execution of instructions belonging to different injected programs bound to the same slot is nondeterministic. User-supplied code may also raise slots during their own execution. The slot mechanism is similar to superposition in UNITY; the main differences are (1) control is transferred explicitly to and from injected programs bound to slots, and (2) the injection process modifies the superposed program such that certain conditions (such as termination) are guaranteed to hold.

3.1.1 Properties of Slots

The model for processing slots is developed using the UNITY formalism. In this section, we outline a set of properties of the slot processing model. The slot processing model specifies:

- A set of underlying programs and their formal properties including slots raised and properties provable when each slot is raised and when slot processing completes.

- Restrictions on the form of the injected code, and formal specification of obligations of the injected code when it is bound to a specific slot.

- A transformation technique to form a composite program consisting of a finite number of injected programs and an underlying program

The restrictions are on the semantic properties of the injected code, and the implementation of restrictions is decoupled from its specification. For example, the underlying program may specify that injected programs must terminate after a bounded number of statements are executed. Such a restriction may be implemented in many ways—by specifying an injected program language that only admits finite "straight-line" programs, or augmenting each statement of injected programs with a program counter and terminating the injected program when its program counter reaches a maximum value It is possible for the node not to explicitly enforce restrictions, in this case, the termination condition. The node may simply *trust* the injected program to terminate (possibly if injected program is signed by a trusted authority). In general, the slot model does not specify means for implementing restrictions—however, in [13], we show that as long as the injected programs discharge their specified obligations, the safety and the progress properties of the underlying program, except those that involve the slot control mechanism, are preserved throughout the active network.

Injected programs that conform to the requisite restrictions and implement the required properties are called programs in "acceptable form". We show that program transformation used to derive the composite program preserves the progress properties of all injected programs in acceptable form. Further, we show that composing the underlying program with a composition of a finite number of programs in acceptable form is *equivalent* to successive composition of the underlying program with each of the programs in acceptable form. These are key properties from the user's perspective as they assert that interaction between a set injected programs is independent of the underlying program, and other injected programs.

The slot processing approach formally exposes the safety, progress properties of the active node in form of restrictions and obligations on injected programs, respectively. The underlying program and the restrictions on the injected programs define the space of properties provable about the active node. From the node provider's perspective, the *level of safety* of the node and the network can be chosen by offering appropriate underlying programs and mandating the form of injected programs.

3.2 The CANEs Execution Environment

In this section, we present the Composable Active Network Elements (CANEs) execution environment for active networking. The EE implements the slot processing programming model described in Section 3.1. The goals of the CANEs EE are as follows:

- To provide a basic set of customization options, including function selection with scalar parameters and non-scalar policies.
- To support path speed commensurate with the level of customization. A fast path should be possible for packets that request low-end (or no) customization.
- To allow customization on multiple time scales, including per-packet and per-flow.
- To provide additional flexibility beyond the basic customization options, by supporting composition.

The CANEs environment consists of a set of underlying programs that are available at each CANEs node. The supported set of underlying program at each node will depend on the node-provider. Further, depending on policy, the set of underlying programs can be either static or they may be dynamically loaded. Obviously, a static set of underlying programs provide opportunities for optimization, possibly via hardware implementation.

Underlying programs can typically be customized by scalar parameters; they also export a fixed set of "slots" to allow further customization. Each slot will has a default behavior that can be over-written by injected programs supplied by the user. Examples of underlying programs include forwarding of various flavors (unicast, small multicast, large multicast, etc.) and basic flow setup. Contingent upon node policy, the CANEs environment may also provide a *install new underlying program* program.

3.2.1 Composing services in CANEs

User computations within CANES are associated with packet flows: flows are identified using user-supplied patterns of bits carried in each use packet. Thus, within CANEs, there is a logical binding of a specific computation with sets of packets that belong to a specific flow. Packets and computations are associated during a signaling phase before the actual data transfer.

The computation associated with a flow specifies the service received by the user from the network. CANEs computations can be used to specify composite services — composition of services is specified in two steps. First, the user identifies one underlying program from amongst those offered by an active node. The underlying program provides a basic service (e.g., forwarding) and exports a set of processing slots to be used for customization (e.g., to replace the default forwarding table with a customized forwarding table). Each processing slot is associated with a specific execution point in the underlying program.

In the second step of service composition, the user selects or provides a set of *injected programs* used to customize the underlying program. The injected programs can already be available at the node, or can potentially be downloaded from the user or a third-party developer. Each injected program is bound to one or more processing slots. The injected program is eligible for execution when the appropriate slot is raised. More than one injected program may be bound to the same slot; the order of execution of instructions belonging to different injected programs bound to the same slot is non-deterministic. As shown in [13], this style of composition has advantages with respect to proving properties about the composite service based on properties of the underlying program and preservation of properties by the injection process. The current CANEs implementation, however, cannot be used to prove formal properties.

Example of Service Composition Using Slots

In this section, we present the schematic of a CANEs underlying program that supports processing slots. Services are composed within CANEs by binding injected programs to these processing slots. In the CANEs implementation, association of computation with packets and binding of injected programs to slots is achieved using the CANEs signaling protocol (Section 3.4.8). An implementation of a similar underlying program using the CANEs API is presented in Section 3.4.7.

Figure 7 illustrates a generalized forwarding function (GFF) underlying program for CANEs that supports forwarding to a small number of destination addresses. The GFF has four parameters. Two are required: a source address S, a list of addresses A. Two are optional: a forwarding table identifier R and a selection function M used to match addresses with forwarding table entries. If the optional parameters are not supplied, defaults will be used. The GFF also exports slots where the user may bind injected programs. Each slot has a default behavior, indicated in the square brackets and used if the user does not supply an alternative. For example, Slot 2 is reached if the list of output interfaces turns out to be null. By default, no error message is generated; the user may choose, for example, to send an error message to the source. The other slots provide opportunities to control action taken upon packet receipt and further processing (e.g., sending notification, incoming and outgoing topology information to the source, application of per-interface policy, flow-specific congestion control etc.).

The CANEs EE implementation is part of the Odyssey active networking environment which consists of the Bowman Operating System, the CANEs EE, and assorted support protocols. Specifics of the implementation is described next.

```
Parse packet to obtain S, A, [R, M]
                ; Required arguments: S (source) A (address list)
                ; Optional arguments: R (Routing Table), M (match function)
Check authorization for R, M (using S)
                ; Processing will be aborted if the auth. fails
⟨Slot 0:[null]⟩
                ; Packet Arrival Slot; Possible uses: route trace, caching
Outputlist := ⊥
                ; Initialize the outgoing set of interfaces to null
For each address a in A:
    Let interface i := Lookup(a, R, M)
                ; Look up the interface for each address a
    ⟨Slot 1:[Add i to Outputlist]⟩
                ; Code bound to this slot may implement policy
                ; to reject certain interfaces
if Outputlist = ()
                ; No output interfaces were found.
    then ⟨Slot 2:[null]⟩; abort;
                ; Code bound to slot 2 can send errors to source
⟨Slot 3:[null]⟩
                ; Code to send output list back to source can be bound to slot 3
For each unique i in Outputlist:
                ; Packet will be dispatched on each outgoing interface
    Create a copy D of the packet,
                with A' = {a : (i, a) ∈ Outputlist}
                ; The new destination address set contains only addresses
                ; that can be reached via interface i
    if i is congested
        then ⟨Slot 4:[discard]⟩
                ; Congestion control algorithms may be bound to slot 4...
    else ⟨Slot 5:[null]⟩
                ; ...alternate schedule policies to slot 5
        enqueue D for i.
```

Figure 7: Schematic of a CANEs Underlying Program (with comments guarded by semi-colons). For simplicity, this example assumes that slot processing does not mutate packets.

3.3 CANEs Implementation in the Odyssey Active Network

The Odyssey active networking platform, consisting of the CANEs EE and the Bowman Node OS layer, is implemented using the C programming language. (Underlying and injected programs in CANEs can be written in any language that can be compiled to produce dynamic load libraries). Both Bowman and the CANEs EE provide support for per-flow multiple concurrent contexts (multiple threaded programs) for per-flow. In the next section, we provide a brief overview of the Bowman node OS and how it relates to CANEs. Detail of the design and performance of Bowman is reported in [77].

3.3.1 Overview of Bowman

Bowman is a single-address space active network node operating system built around three key abstractions: *channels*, *a-flows*, and *state stores*.

- **Channels.** A channel is the primary communication abstraction in Bowman. Channels are communication end-points and consist of a set of protocols using which data can be sent and received at Bowman nodes. In Bowman, channels are end-points of links to other Bowman nodes; thus, unlike sockets, channels are a node-level construct and do not belong a single user.

  The channel implementation in Bowman encapsulates the protocol processing required to implement individual communication protocols. Along with input and output protocol processing, Bowman channels allows other forms of processing such compression and forward error correction, to be defined on specific links as well. In effect, channels decouple user computations from the underlying implementation of particular communication protocols. Bowman exports interfaces to create, destroy, query and communicate over channels.

- **A-Flows.** The a-flow[1] is the primary abstraction for computation in Bowman. An a-flow encapsulates a set of processing contexts and user state. Each a-flow consists of at least one thread and executes on behalf of an identified principal (user of the system). Bowman provides interfaces to create, destroy and run a-flows. Bowman also provides interfaces to create and destroy threads within an a-flow.

- **State-store.** The state-store is an associative memory that allows (fixed-size) pieces of information to be stored and retrieved via (fixed-size) *keys*. The Bowman state-store interface provides for creation of a binding between a key and a data item, and for retrieving the data item associated with a given key.

The *Host* OS

In order to implement the channel, a-flow, and state-store abstractions Bowman requires interfaces to lower level hardware-specific mechanisms such as memory management, thread creation and scheduling, synchronization primitives, device input and output. The Bowman implementation is layered on top of a *host* operating system that provides these lower level services. The host operating system implements hardware-specific routines for device and memory access, synchronization etc. Bowman can be ported to different host operating systems. The current Bowman (and CANEs) implementation has been ported to SunOS (versions 5.5 and higher) and Linux (kernel versions 2.2.0 and higher).

Bowman Extensions

Bowman provides an *extension* mechanism that is analogous to loadable modules in traditional operating systems. Using extensions, the Bowman interface can be extended, dynamically, to provide support for additional abstractions such as queuing

[1]The term "flow" is often used to describe per-user computation state at an active node [42]. As these flows are, in a manner, *active*—i.e. they refer to both a set of packets and some processing, we adopt the use of the term "a-flow".

mechanisms, routing protocols, user-specific protocols and other network services. Bowman extensions are written using Bowman system calls and calls exported by other extensions. In the latter case, all required extensions must already be loaded into the system.

EEs on Bowman

There are two different approaches for implementing EEs over Bowman: the monolithic approach and the multi a-flow approach. The code for an EE can be loaded as an extension to Bowmanand typically contains the control code for the EE and a set of routines that implement the user-network interface supported by the EE. In order for the EE to process packets, the control code must spawn at least one a-flow. In the monolithic approach, the EE creates only one a-flow and does not expose its internal processing structure to Bowman. Instead the EE subscribes to all packets requested for processing on behalf of all its users and can demultiplex incoming packets to different threads within its single a-flow. In the multi a-flow approach, the EE code spawn one control a-flow that can used for the EE signaling and housekeeping. The control a-flow starts a different a-flow for processing each user's packets. Note that in each case, third-party code, loaded on behalf of users of the network, is linked against the API that is exported by the EE; however, in the multi a-flow approach, the user's computation is scheduled by Bowman.

3.3.2 CANEs in Bowman

The CANEs EE is implemented on Bowman using the multi a-flow approach. A CANEs user flow executes in its own a-flow — the code within the user a-flow is linked against the CANEs API (described in Section 3.4). User computations are set up using an explicit signaling phase (discussed in detail in Section 3.4.8). During the signaling phase, both the computation and its inputs, outputs are specified. During execution of the flow, all packets that match a specified pattern are input to the computation. The signaling phase can also identify a routing scheme for routing outgoing packets. Different components of an active node running the CANEs EE is shown in Figure 8. The control component of the CANEs EE is implemented using the node OS interface exported by Bowman.

Figure 8 shows a thread-level abstraction of our CANEs implementation over Bowman. Upon startup, the CANEs EE spawns the control a-flow that contains a thread for handling signaling messages (the CANEs signaling interface is described in Section 3.4.8). Underneath the EE, several system threads provide support for packet input, output, timers and other Node OS routines. Each user's flow is executed in its own a-flow and can spawn multiple independent threads that share an address space. The CANEs EE itself provides a library that implements the CANEs API and resides in the system as a single EE a-flow that contains a set of threads to handle housekeeping chores (like signaling and timers).

In our implementation, both the underlying and injected programs are demand loaded, dynamically using the Bowman extension mechanism. The slot processing model described in this chapter is implemented; both underlying and injected programs can export slots, to which multiple sub-injected programs may be bound. The implementation provides mechanisms for variable sharing between programs exporting and sharing slots. In the next, we describe, in detail, the CANEs API. This API along with the signaling interface is the network API exported by the CANEs EE.

3.4 The CANEs API

As described in Section 3.2, composition in the CANEs EE is achieved in two steps — the user selects an "underlying program" and provides a set of "injected programs". Injected programs are bound to slots in the underlying program and are used to customize the underlying program on a per-user basis. Communication between the underlying and the injected program is via shared variables. The CANEs

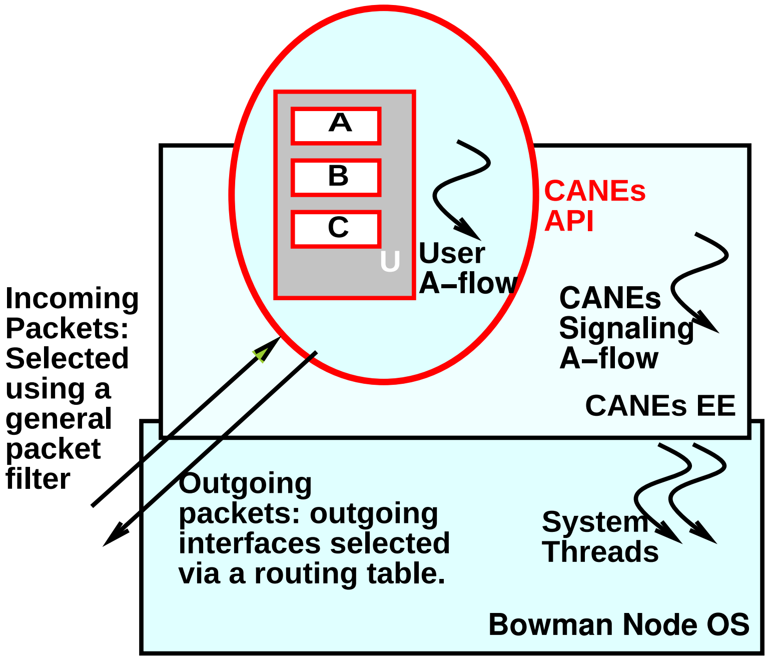

Figure 8: A high level view of processing at an active node running the CANEs EE over the Bowman Node OS. User flows consist of an underlying program (U in this figure) and a set of injected programs (A, B, and C). This figure also shows a nominal thread level structure of our implementation. The CANEs EE spawns a control a-flow that contains a signaling thread; each user flow has its own a-flow and set of threads. There are also a set of system threads that run within the Node OS.

API provides comprehensive support for slots and variable sharing. An overview of the data types defined by CANEs is given in Table 2. The defined data types map to abstractions present within the EE: user flows, slots, channels and timers. Apart from the usual interface for packet transmission and timers, CANEs provides a set of functions for slot management and for shared variable manipulation. Next, we classify the CANEs API by functionality and provide details of each API call. Examples illustrating the use of the API for both underlying and injected programs is presented in 3.4.7.

Table 2: CANEs API: Defined Data Types. Internally, each type is mapped onto a data type in C or to a type defined by Bowman.

Type Definition Name	Denotes
canes_boolean_t	Boolean
canes_generic_t	Opaque (no specified type)
canes_channel_id_t	Channel identifier (maps to Node OS channels)
canes_flow_id_t	CANEs internal user-flow id
canes_slot_id	CANEs slot identifier
canes_program_t	Prototype for injected, underlying programs
canes_packet_t	CANEs packet (maps to Node OS packets, if any)
canes_packet_length_t	Packet length
canes_tick_t	Time unit specifier
canes_timer_func_t	Prototype for timer handler functions
canes_timer_handle_t	Timer Identifier

3.4.1 The Slot Mechanism

Prototypes for the CANEs slot control functions are shown in Figure 9. Slots must be declared using the `canes_slot` macro and exposed to injected programs using the `canes_export_slot` call. The argument `slot_name` for both declarations can be any valid C variable name. In effect, the `canes_slot` call declares a slot for use (internally a C variable of type `canes_slot_id` named `slot_name` is declared and initialized). Exporting a slot makes it visible outside the program where the slot is declared; this allows injected programs to be *bound* to the slot. The `canes_slot` declaration will fail iff the slot name is not a valid C variable name. The export and raise calls will fail iff the slot name is unknown. Slots can be exported more than once; a warning message is generated in this case.

he `canes_raise_slot` function takes a slot identifier as an argument and invokes injected programs *bound* to the slot. The execution order for bound injected programs is not specified, i.e. all of the injected programs bound to the slot may execute concurrently with arbitrary interleavings. For a raise call to succeed, `slot_id` must be a variable previously declared a `canes_slot` call and exported using the `canes_export_slot` call.

CANEs Slot Control Functions
canes_slot(slot_name);
canes_boolean_t canes_export_slot(canes_slot_id slot_name);
canes_boolean_t canes_raise_slot (canes_slot_id slot_name);

Figure 9: Prototypes for CANEs slot control functions

Invoking a slot is analogous to raising a named event. In this metaphor, all injected programs bound to a slot are handers for the named event and are executed concurrently when the event is raised. Note the absence of a call that allows an injected program to be *bound* to an exported slot. Binding of injected programs to slots are part of CANEs signaling; the API does not export an explicit call to bind injected programs to exported slots.

3.4.2 Variable Sharing

The CANEs API provides a variable sharing mechanism that is used by injected programs to communicate with underlying programs. We adopt the following terminology: a *shared* variable refers to a variable that can be accessed by multiple programs. Underlying programs declare shared variables and *export* them (make them eligible for sharing). Injected programs declare *references* to shared variables (the space for the actual data for references must be declared by the underlying program). Further, injected programs must *import* shared variables exported by underlying programs (this finally creates the binding between the shared and the references variables). Further references, internally, are pointers to shared variables. Thus their values change synchronously each time the value of the shared variable changes in the underlying program, i.e. there is no need to export a shared variable each time its value changes.

> Aside: CANEs injected programs can also export and raise slots to which "sub-injected" programs can be bound and so on. In effect, the slot exporting program (which may itself be an injected program) forms the *underlying* program to which injected programs can be bound. Note that variables are shared only between programs that export a slot and the set of programs immediately immediately bound to the exported slot. Variable sharing is constrained to one level of injected programs. Consider an example: Let U be an underlying program, I a program bound to a slot in U, and J a program bound to a slot exported by I. The program I can import variables exported by U. The program J can *only* import variables exported by the program I. However, it is possible for the program I to import a variable from U and subsequently export it. This variable, originally exported by U, and re-exported by I can now be imported by the program J. (End Aside).

Two different types of sharing is supported by CANEs. Multiple copies of an underlying program can be instantiated by different users concurrently. Each user-flow maintains a private copy of a *per-flow* shared variable. In this case, the sharing is between variable injected and underlying programs within the context of an user's flow. Shared variables can also be declared by underlying programs without the per-flow attribute. In this case, all the instantiations of the underlying program (and injected programs that bind to these instantiations) share a single copy of the variable.

For the rest of this section, we describe variable sharing in CANEs in terms of a single underlying program to which a single injected program is bound. The C prototypes for constructs that provide the CANEs shared variable interface is shown in Figure 10.

Declaring and Exporting Shared Variables
canes_shared_global(type, variable);
canes_shared_global_per_flow(type, variable);
canes_boolean_t canes_export_global_per_flow (char *name, canes_generic_t p);
canes_boolean_t canes_export_global (char *name, canes_generic_t p);

Referencing Shared Variables
canes_referenced_global(type, variable);
canes_referenced_global_per_flow(type, variable);
canes_generic_t canes_import_global_per_flow (char *name);
canes_generic_t canes_import_global (char *name);

Accessing Shared Variables
c_Ep(variable), c_Eg(variable)
c_Ip(variable), c_Ig(variable)

Figure 10: Prototype for CANEs variable sharing functions and declarations.

3.4.3 Declaring and Exporting Shared Variables

CANEs allows variables of arbitrary types to be shared between underlying and injected programs. The shared variable must be declared by the underlying program and then referenced by the injected program. (Underlying programs cannot access variables declared by injected programs simply because there is no guarantee about a particular injected program being bound to a slot). The underlying program must declare the shared variable using the `canes_shared_global[_per_flow]` macro call. The arguments are a defined but otherwise arbitrary C type and a legal variable name.

Before an injected program can reference shared variables, they must be exported using the `canes_export_global[_per_flow]` function call. The arguments to the export call is a key (`name`) and a pointer to the shared variable (`p`).

When an injected program tries to import this shared variable, the key `name` will be used to create the requisite *binding* between the shared variable and its referencing counterpart.

CANEs Packet Input and Output

```
canes_packet_length_t canes_send(canes_channel_id_t ch_id,
  canes_packet_t **pk, canes_packet_length_t len);
canes_packet_length_t canes_recv(canes_packet_t **pk);
```

Figure 11: Prototype of the CANEs data transmission functions

Referencing Shared Variables

An injected program that references a shared variable must know the type of the variable. References are declared using the `canes_referenced_global[_per_flow]` macro call.

Sharing must be explicitly initiated by the `canes_import_global[_per_flow]` call. The arguments is a key (that is matched against the key of the all variables exported by the underlying program). The value returned by the import calls should be assigned to a referenced variable.

Internally, references are declared as pointers. The export/import key is used to find a shared variable that the referenced variable should point to. This scheme *does* lose type information, it is the responsibility of the programmer to ensure that the shared and referenced variables are compatible. It is possible to impose a more substantial (but language independent) mechanism that ensures type safety.

Accessing Shared Variables

Exported (shared) and imported (referenced) variables must be accessed through the four macro calls shown above. The `c_Ep` and `c_Eg` calls are used to access exported per-flow and exported global variables, respectively. Analogously, the `c_Ip` and `c_Ig` calls are used to access imported per-flow and imported global variables.

3.4.4 Packet Transmission

The unit of data transmission in CANEs is a packet. The structure of a CANEs

Table 3: Fields in the CANEs packet structure (inherited from Bowman).

Field	Type	Description
`from`	`canes_channel_id_t`	Channel id on receive, Sender id on send
`time`	`struct timeval`	Receive timestamp
`ref_count`	`unsigned int`	Reference count
`buffer[]`	`char array`	Data buffer
`d_off`	`unsigned int`	Data offset
`len`	`unsigned int`	Valid length

packet is shown in Table 3. The prototypes of functions used to send and receive packets is given in Figure 11.

Send and Receive

User code uses the `canes_send` call to send packets. The send call queues a packet `pk` of length `len` on the channel `ch_id`, and upon success returns the number of bytes queued. The packet is mutated and freed by a successful canes_send. The `canes_recv` blocks for a packet on the flow's input queue. Upon success, the receive call dequeues the packet on the head of the input queue and passes the packet back (mutating the parameter `pk`). The receive call returns the length of the received packet.

3.4.5 Timers

The CANEs EE supports multiple timers per user flow. The timer implementation is multi-thread safe. The function prototype for all timer handlers is:

typedef void (*canes_timer_func_t) (int arg);

Thus, CANEs timer handlers do not return a value and may be passed an integer argument.

Setting and Canceling Timers

The prototypes for the CANEs timer handler routines is shown in Figure 12 The `canes_timeout` call is used to set a timer. After at least `num_ticks` "ticks"[2], function `timer_f` will be executed with argument `arg` (unless the timer has been successfully canceled). The function returns a handle to the timer established. The returned handle can be used to cancel the timer using the `canes_timeout_cancel` call.

CANEs timer handlers execute within a per-flow timer thread. Timers that expire at the "same" time are executed sequentially. The order of the executions of timer handlers that expire at the same time is not specified.

3.4.6 Miscellaneous

We describe initialization and logging routines supported by the CANEs API.

The CANEs Entry function

Every CANEs (injected or underlying) program must define a function named `_entry` with the prototype:

typedef void (*canes_program_t)(canes_flow_id_t f);

When a program is loaded on behalf of a particular flow, this function is executed.

CANEs Logging

Messages are logged (to a predefined log file) the `canes_log` function (prototype in Figure 12). Upon calling the log function, the boolean condition `b` is evaluated. If the boolean predicate evaluates to true, the message is logged. The `format string` and the variable set of arguments are specified like in the `printf` family of functions in the C library.

[2]Each "tick" is a fixed unit of time.

CANEs Timer Handling Functions
canes_timer_handle_t canes_timeout (canes_tick_t num_ticks,
  canes_timer_func_t timer_f, int arg);
canes_boolean_t canes_timeout_cancel (canes_timer_handle_t th);

CANEs Output Log
u_int canes_log (canes_boolean_t b, "format_string", variables...);

Unique Identifier within CANEs
canes_flow_id_t canes_getid();

Figure 12: Prototype of CANEs initialization and support functions

Unique identifier

The `canes_getid` function defined in Figure 12 returns an unique identifier for each flow within the CANEs environment. Note that this identifier may not correspond to identifiers (e.g. thread or a-flow ids) assigned by the node OS.

3.4.7 Example Usage of the CANEs API

In this section, we present examples of the use of the CANEs API. An underlying program `default-fwd` is shown in Figures 13 and 14. The `default-fwd` program can be used to forward unicast packets. It raises two slots `ProtocolProcessing` and `DefaultRoute`. Figure 13 shows the declarations of the slots, declaration of shared variables, and the initialization function. The initialization function exports the slots and the shared variables (such that these can subsequently be used by injected programs). Figure 14 lists the main packet processing loop. The `default_fwd` function continuously reads packets input on any of its subscribed channels. Upon receiving a packet, it exports the received packet to injected programs and raises the `PacketProcessing` slot. If the `PacketProcessing` slot does not modify the `route_this_packet` shared variable, then the `DefaultRoute` slot is raised. The

```
#include "canes.h"

Exported slots: A slot name can be any valid "C" variable name
canes_slot(ProtocolProcessing);
canes_slot(DefaultRoute);

Shared variables:
canes_shared_global_per_flow(canes_packet_t *, cur_pkt);
canes_shared_global_per_flow(canes_channel_id_t, Output_If);
canes_shared_global_per_flow(boolean, route_this_packet);


Each imported underlying/injected program must identify an _entry function.
It is typically used to initialize local variables, export slots and variables.

void _entry (void){
    Default-fwd is an underlying program, it cannot import any variables

    Export the variables ...
    canes_export_global_per_flow( "Cur-Pkt", cur_pkt);
    canes_export_global_per_flow( "Oif", output_if);
    canes_export_global_per_flow( "RouteThisPacket", route_this_packet);

    ... and the slots
    canes_export_slot(ProtocolProcessing);
    canes_export_slot(DefaultRoute);
}
```

Figure 13: Preamble and the initialization for the example CANEs underlying program. Comments are shown in italics.

```
void default_fwd (void) {

    for (;;){
        canes_packet_t *pk;
        boolean route_pk;
        canes_channel_id_t ch_id;
        canes_packet_length_t l;


        l = canes_recv (&pk);
        canes recv will receive packets from all the incoming channels

        c_Ep(cur_pkt) = pk;    Export the received packet
        c_Ep(route_this_packet) = TRUE;
          Initialize route_this_packet, default is to route the packet

        Raise slot for per protocol processing.
        canes_raise_slot(ProtocolProcessing);

        route_pk = c_Ep(route_this_packet);
        The slot may have changed route_this_packet ...

        if (route_pk) {
            pk = c_Ep(cur_pkt);    ... or the packet itself
            c_Ep(output_if) = -1;   initialize the output interface

            canes_raise_slot(DefaultRoute);
            Injected programs bound to DefaultRoute should set the output interface

            ch_id = c_Ep(output_if);
            if (ch_id ≥ 0) {
                l = canes_send(ch_id, &pk, pk→size);
            } else canes_log (1, "Cannot resolve output channel");
        } else canes_log (1, "Packet dropped by ProtocolProcessing Slot);
    }
}
```

Figure 14: The `default-fwd` function. This function is invoked when the user a-flow executes. (Calls to free the packet are not shown in this listing.)

DefaultRoute slot is used to find an appropriate output channel on which to dispatch each packet. The canes_send call is used to send the packet on the channel selected by injected program(s) bound to the DefaultRoute slot.

```
canes_referenced_global_per_flow (canes_packet_t *, cur_pkt);
canes_referenced_global_per_flow (canes_channel_id_t, oif);

void route (void) {
    canes_packet_t *pk = c_Ip(cur_pkt);
    c_Ip(oif) = o_rt_lookup(rt_id, pk);
}

void _entry(void) {
    cur_pkt = canes_import_global_per_flow ( "Cur-Pkt");
    oif = canes_import_global_per_flow ( "Oif");
}
```

Figure 15: An injected program designed to bind to the DefaultRoute slot of the default-fwd underlying program. Calls to resolve the routing table identifier (rt_id) are not shown.

Figure 15 shows an injected program that can be bound to the DefaultRoute slot. The injected program declares references to variables exported by the default-fwd program. It imports these variables in its initialization function. The route function is invoked each time the DefaultRoute slot is raised; this function uses an Odyssey extension to find the output channel for the current packet. The output channel is communicated back to the underlying program using the shared oif variable.

3.4.8 CANEs Signaling

The CANEs signaling interface implements the CANEs User Interface (**CUI**) signaling protocol. CUI messages are sent on a predefined port and consist of a CUI message specified in the CUI language. The CUI message is in two parts: computation and communication.

The computation part defines a directed acyclic graph (DAG). The root of the DAG is an underlying program. Each child node corresponds to an injected program bound to a slot in its parent. The arcs in the DAG identify the slot to which the child is bound. Each node contains enough information for the EE to fetch the code required to execute the computation DAG.

The communication part of the CUI message identifies a routing schema and a Odyssey topology for the user. Further, the user can specify a set of packet classifiers that is used to select incoming packets that the flow acts on. This communication part provides enough information for the CANEs EE to establish the input and output channels required for the user's computation.

We provide a definition for the CUI language and an example of its use next.

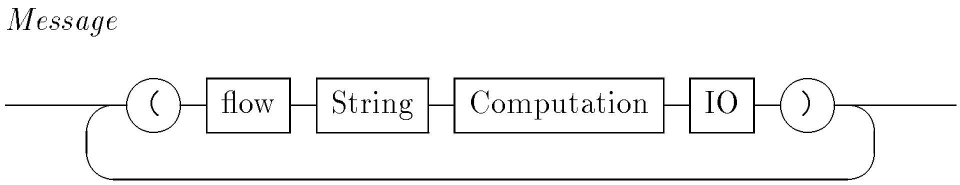

Figure 16: Top-level structure of CUI messages

Figure 16 shows the top-level BNF representation of a CUI messages; a CUI message consists of a flow name, specifications for a computation and its inputs and outputs. The specification of the computation defines a DAG with the underlying program as the root. A set of data items are also defined in the computation structure. The data items are keyed using an unique character string and are available to the computation via the state store.

Figure 17 shows the details of the computation part of CUI messages. Each data item (if any) is defined by the key and value string and is put in the per-flow

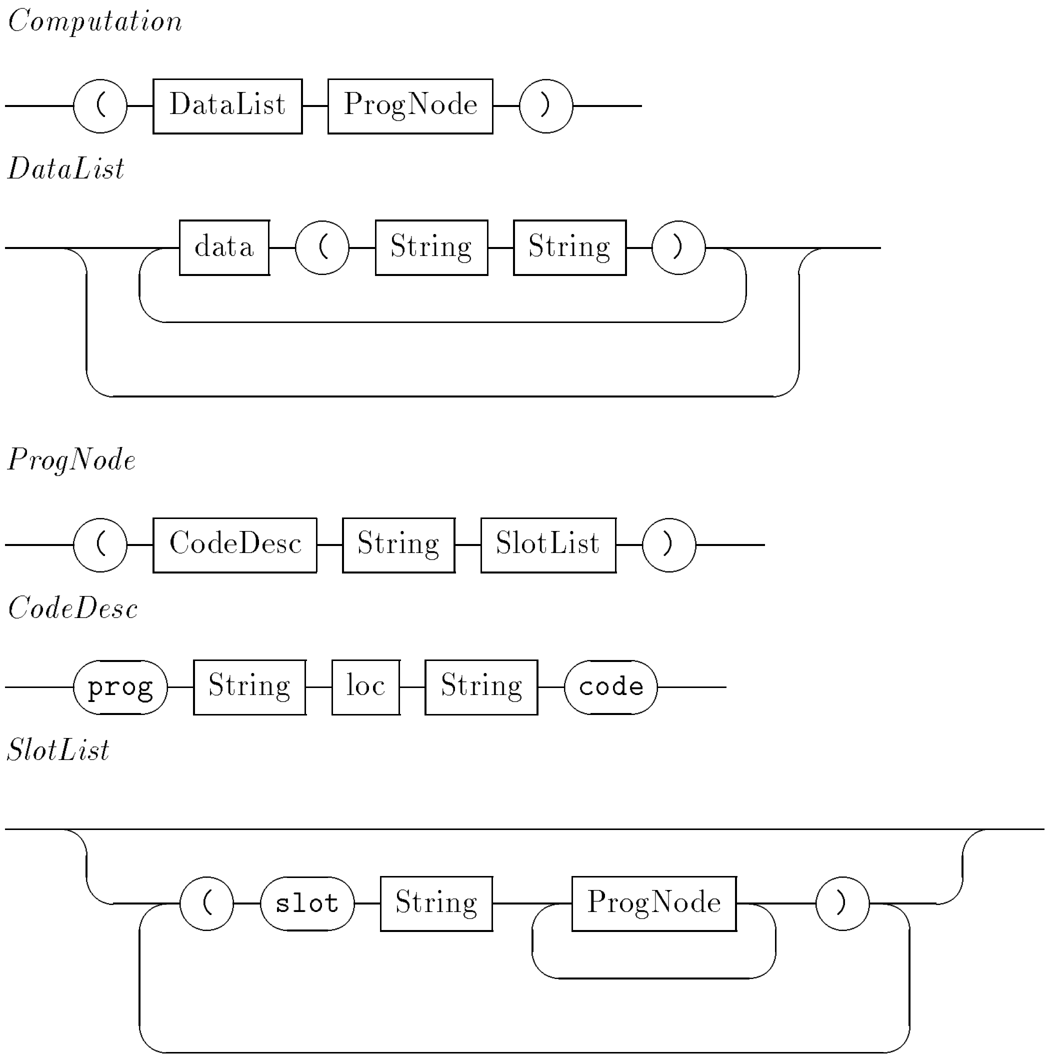

Figure 17: Computation part of CUI messages

store. Each node (`ProgNode`) is defined by the name of the program (specified after keyword `prog`), a location to fetch the program from (location specified after the keyword `loc`) and a function that is invoked to start the computation (specified after keyword `code`). Each node also specifies a list of slots to which other programs are bound. These slots must be defined by the program specified in the node. The slot list specifies the slot name and the root of another sub-DAG. Note that multiple programs can be bound to the same slot.

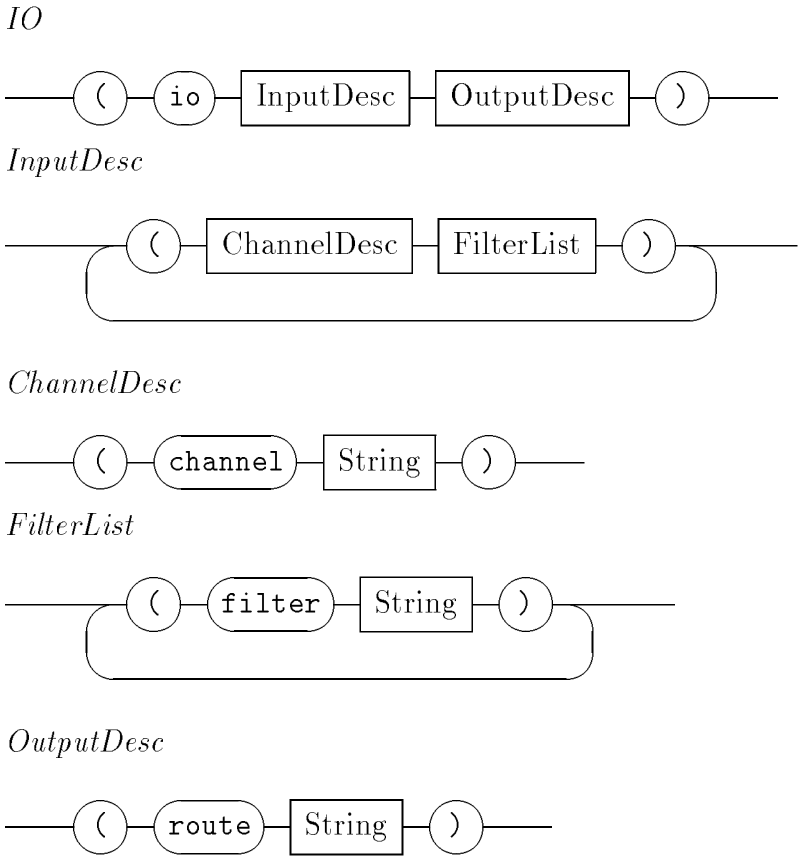

Figure 18: IO part of CUI messages

Figure 18 shows the details of the IO part of CUI messages. The input specification contains a set of of channel names and and a list of filters for each channel. Packets that arrive on each of the specified channels and match *any* of the filters specified for that channel will be routed to the computation. The output specification is the name of a routing schema.

An Example CUI Message

(**flow** "count-IP"
                    ; This message defines the count-IP *flow*
    (**computation**
        (**prog** "default-fwd"
                    ; default-fwd *is the underlying program . . .*
         loc "cfs://canes.codeserver/underlying"
         code "default_fwd
            (**bind** DefaultRoute
                    ; . . . which exports a slot named DefaultRoute
                    (**prog** "route"
                     loc "cfs://canes.codeserver/injected"
                     code "count")
                    ; Program `route` *will route incoming packets*
)
)
)
    (**io**
        (
            (**channel** "Default:.*")
                    ; This computation will run on all links of the Default *topology*
                    (**filter** "[ip (src = erlang) (dst = gladstone) (proto = TCP)]")
                    ; for all TCP packets from hosts erlang *to* gladstone
)
        (**route** "IP:RIP")
                    ; Each matching packet will be forwarded using the IP:RIP table
)
)

Figure 19: An example CUI message that binds the program `route` to the slot `DefaultRoute`. CUI keywords are shown in bold face; comments are italicized and guarded by semi-colons.

Figure 19 shows an example CUI message. In our implementation, program locations are specified as Universal Resource Locators (URLs). In this message, the underlying program is called `default-fwd`. It resides in the `underlying` directory of the `canes.codeserver` host and is fetched using the `cfs` protocol. The program `default-fwd` exports a slot `DefaultRoute` to which is bound the injected program `route`. This defines a computation DAG with two nodes (root: `default-forward` and leaf: `count`) and one arc (slot: `DefaultRoute`). This entire computation occurs on packets that arrive on links with name that match the wildcard `Default:.*` and pass through the filter. In our implementation, this link specification corresponds to all of the links of the topology `Default` that is incident on the current node. The filter imposes the condition that the packet IP source must be `erlang` and the packet IP destination must be the host `gladstone` and the IP protocol must be `TCP`. Each matching packet is forwarded using the `IP:RIP` routing table.

3.5 Related Work

In this section, we provide a survey of composition mechanisms employed by other active networking EEs.

3.5.1 SwitchWare

The SwitchWare project [1, 96] makes a distinction between two levels of programming interface. The upper (packet) level provides a scripting language that has limited functionality, but strong safety properties. Programs written in the packet level programming language are carried by packets and invoked as part of generic packet processing. The lower (service) level interface is intended to be a richer programming language. Programs written in the service level programming language are used to provide more complex services that can be invoked by a packet level program. The current version of this two-level architecture uses a new language called PLAN (Packet Language for Active Networks) as the packet level specification language, and CaML (a dialect of ML) as the service level language [55, 2]. PLAN is a simple language based on typed lambda calculus. The language provides some safety guarantees (e.g., guaranteed termination, strong typing), however providing guarantees for PLAN programs will obviously depend on properties of the services called by the program. The services, implemented in CaML, are installed at a node using an unspecified API.

Therefore, composition in the SwitchWare EE is achieved in two steps: the user selects a set of services and composes these using the PLAN language. This approach is a *dual* of the CANEs approach. Under CANEs, the base computation is fixed by the underlying program and user's can supply injected programs to customize the properties of the underlying programs. However, in SwitchWare, the customization is in the form of selecting services already available at the node. A further difference between the two schemes is that under SwitchWare, each PLAN packet has to be interpreted at run-time to realize a particular service. In CANEs, the signaling phase is used to realize all the bindings between underlying and injected programs that constitute a particular service and there is no interpretation at run-time. The CANEs design is based upon the observation that for many services, packets in a flow require the same set of bindings — i.e. there is a common processing path for almost all packets in a flow and it is often not necessary to reconfigure the computation on a per-packet basis. The SwitchWare approach provides more flexibility on a per-packet basis; the CANEs approach however provide better performance while restricting flexibility of re-configuration to a per-flow granularity. Note that it is possible to provide per-packet reconfiguration of the injected and underlying program bindings under CANEs; however in this case, the bindings have to be interpreted per-packet (much like interpreting PLAN packets).

3.5.2 NetScript

The use of a "glue" language as the programming interface is also fundamental to the NetScript project [106]. Composition in the NetScript EE is based on the dataflow computation model. Code modules (boxes) communicate with each other using input and output *ports*. A NetScript computation consists of a computation graph: each node is a code box and the arcs in the graph correspond to data flow. Thus, NetScript is a "co-ordination language" for protocol composition. In structure, the NetScript dataflow approach [36] is similar to composition using Unix System V STREAMS [89], the x-kernel [61] protocol graphs, and the SM protocol subsystem [33]. The composition structure of NetScript is similar to that provided by STREAMS. However, STREAMS does not provide a mechanism to specify computations using a high-level language. Unlike the x-kernel, the NetScript computation graph can be dynamic, i.e. connections between boxes or entirely new boxes can be added at run time. Like SM, Netscript boxes are hierarchical (i.e. they can encapsulate other boxes); however, the SM system also has a strong notion of control flow (using synchronous reactive computations) that is missing in NetScript.

Like CANEs, the NetScript EE does not mandate a particular code load mechanism. The NetScript approach is useful to create and instantiate new protocols inside the network (much like the x-kernel protocol graph mechanism is useful for the implementation of protocols at a host). However, service composition in Netscript differs from CANEs in one crucial aspect: there is no notion of an underlying program. The entire service for an user is constructed of boxes which are the only *first-class* object inside the EE. Boxes abstract both the data and the control flow. Thus, there is no notion of a generic packet path within the EE — this makes *service* creation and customization difficult in NetScript.

3.5.3 ANTS

The MIT Active Networks project defines the ANTS toolkit for programming active nodes [103]. In ANTS, packets carry an identifier that indicates the routine to be used in processing the packet and parameters specific to the referenced routine. Some routines are well-known and available at every active node; other routines are demand-loaded, using an in-band code-loading mechanism. ANTS routines are written in a high level programming language that can contain primitives exported by an active node. The current ANTS prototype is written in Java using four primitive classes [103].

ANTS does not provide any particular mechanisms for composition, however the ANTS implementation does inherit object-oriented composition techniques due to the current Java implementation. Thus, within the ANTS EE, static composition can be achieved by class inheritance in which sub-classes are automatically generated using virtual methods provided by users[8, 60]. These virtual methods override their abstract counterparts in the base classes exported by the active node. Note that the notion of a generic per-packet processing path is absent inside the ANTS EE.

ANTS inherits safety properties from its execution within the Java Virtual Machine. Mobile code in ANTS is in the form of Java classes and executes within the JVM "sandbox" [8, 74]. Both ANTS and PLAN include an additional safety mechanism in the form of a generalized time-to-live field in the header of a packet. This value is decremented by active nodes that are visited by a packet, and the packet is discarded when the limit reaches zero. Any additional packets created by packet processing must inherit a resource bound from the original packet. Precise policies, and associated guarantees, regarding the allocation of resources to children have not been developed.

3.5.4 SmartPackets

In the SmartPackets environment, code is written in an imperative language that forms the basis for composition. There are no other specific mechanisms for composition of code modules.

Chapter 4

Application Specific Congestion Control

4.1 Introduction

In this chapter, we introduce a specific application of active networking: Application-Specific Congestion Control (ASCC). ASCC uses application-level knowledge to tailor the network's behavior upon congestion. In the rest of this chapter, we refine this basic idea, apply it to specific end-to-end applications, and present results and conclusions. We begin with an overview of general problems in congestion control and specific techniques to contend with these problems.

4.1.1 Background

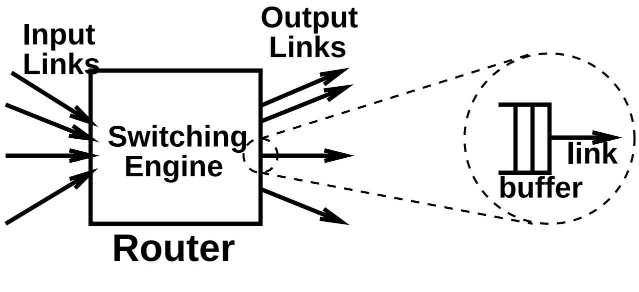

Figure 20: Schematic of an output buffered router. Packets come in on input links and are routed through the switching engine to the output links. Packets routed to a busy output link are buffered (if possible).

Figure 20 is a schematic diagram of a typical Internet router. Packets are received on incoming links and are routed through the switching core to an output interface[1]. In general, multiple incoming packets can be switched through the router simultaneously, and multiple incoming packets can be destined to a single output link. We say that a link is congested when the number of packets (or equivalently, the number of bits) that have to be transmitted on a link exceeds its capacity (usually measured in bits per second). Usually, there is a buffer associated with each link. Packets routed to a busy link are queued in this buffer. It is possible for this buffer to fill up. In the current Internet, when the buffer associated with an output link is full, relatively simple mechanisms —e.g. drop subsequent packets that do not fit in the buffer (Tail Drop)—are employed to alleviate the situation. A more sophisticated scheme, Random Early Discard (RED), is also being deployed [50]. Under RED, two queue occupancy levels corresponding to a high and a low watermark is associated with the output queue. If the long-term (exponentially averaged) buffer occupancy is less than the low watermark, no packets are discarded. If the long-term occupancy is greater than the high watermark, all packets are discarded. If the long term occupancy is between the high and the low watermark, the packet drop probability increases with buffer occupancy. Probabilistically, the RED scheme discards packets in proportion to each connection's use of the congested link.

In either case, congestion manifests itself as packet loss. In fact, on a (wired) network, congestion is the primary reason for packet loss [65]. As such, lost packets are a congestion signal. Figure 21 shows the prototypical congestion scenario on the Internet. Packets from the sender are lost due to congestion inside the network. Assuming not all packets are lost, the congestion (packet loss) is first detected at the receiver. The receiver then generates a congestion notification, possibly in form of a transport-layer acknowledgment, for the sender. Upon receiving this notification, the sender decreases its transmission rate. The sender does not alter its sending rate till it receives a notification of congestion. Thus, the receiver encounters uncontrolled packet loss in the period between when the data first encounters congestion and when

[1]In case of an error, packets may resolve to no outgoing interface. In case of multicast, a packet may be routed to multiple outgoing interfaces. For this discussion, it suffices to consider the unicast error-free case, i.e. packets are routed to precisely one output interface.

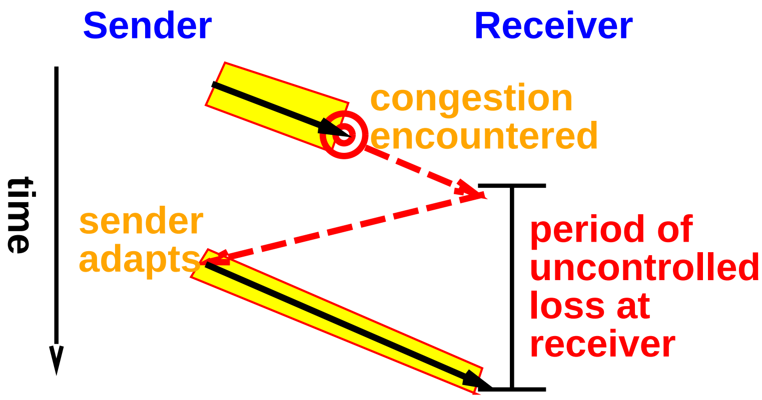

Figure 21: The sender adaptation model. Congestion is detected by the receiver in the form of lost packets. The receiver then sends a congestion notification signal to the sender. Upon receipt of this signal, the sender decreases the transmission rate.

the congestion notification is received at the sender. This period is proportional to the latency between the sender and the receiver; the amount of data lost is proportional to both the latency and the original rate at which the data was being transmitted by the sender. Note that when a large number of consecutive packets are lost, or when the congestion notification itself is lost, the sender can still detect congestion using a timer (set to expire after a factor of the estimated packet round-trip time). We refer to this model of adaptation as the *sender adaptation model.*

Why is the problem of congestion persistent?

Recent advances in understanding the self-similar nature of traffic [71, 73, 104] indicate that effective congestion control is likely to require longer time scale methods (e.g., source adaptation and admission control) rather than shorter time scale methods (e.g., adding more buffering to output links). Packets that are buffered experience longer delay. Thus, adding buffering beyond a certain amount prolongs packet transfer to an extent that the source is not able to distinguish between lost packets and long delays. In this case, buffering exacerbates the situation as the sender times out and retransmits [66].

The sender-adaptation model [65] described above has worked well in the Internet. However, it presents some well-known challenges. The first is the time interval required for the sender to detect congestion, adapt to bring losses under control, and have the controlled-loss data propagate to the receiver. During this interval, the receiver experiences uncontrolled loss, resulting in a reduction in quality of service that may "magnify" the actual bandwidth reduction. (For example, if a portion of each of several application data units is lost, each entire data unit may become useless to the receiver.) The amount of data lost is proportional to the bandwidth-delay product on the end-to-end path between the sender and the receiver. As transmission bandwidth and application requirements increase, the amount of lost data increases (because propagation delays remain constant) and this problem is exacerbated .

The other well-known challenge of sender adaptation is detecting an *increase* in available bandwidth. This problem, which is worse for continuous-media applications, arises in traditional best-effort networks because loss is the only mechanism for determining available bandwidth. Thus, for example, if a sender adapts to congestion by changing to a lossier encoding, it must detect the easing of congestion by periodically reducing compression and waiting to see whether losses ensue. In the case of long-lived congestion, this dooms the receiver to periodic episodes of uncontrolled loss.

What about Active Networks?

Active networks provide applications an interface to customize the network's behavior. The usual problems of congestion are also present in an active network. However, note that both Tail Drop and RED are generic mechanisms; in each case, the drop criterion is independent of any application-specific information. Application-specific congestion control (ASCC) exploits the interface provided by active networks to tailor the network's behavior during periods of congestion. Specifically, we show that best-effort service provided to *adaptive applications* can be enhanced by allowing applications some control over the way their packets are processed in network switches when they encounter congestion.[2]

Thus, instead of applying "one size fits all" congestion reduction techniques, mechanisms can be placed in the network to allow packet processing —e.g. discarding or transforming— to proceed according to advice supplied by the application. The main observation is that the application knows *how* to adapt to congestion, while the network knows *when and where* adaptation is needed. In essence, the adaptation that the sender would have applied can be performed at the point of congestion.

Adapting at the router effectively solves the problem of uncontrolled data loss while the congestion signal reaches the sender. In effect, the congestion notification is instantaneous as the signal does not have to traverse the network. Further, if the available bandwidth should increase, the network will stop applying the data reducing transformations. Once again, in effect, the increase in bandwidth is also detected instantaneously.

Note that the ASCC mechanism can be co-employed with end-to-end feedback. In this case, the network-based adaptation will be applied on shorter timescales, during the period it takes for the sender to be notified of the long-term changes in the network conditions.

4.1.2 A Nomenclature for Congestion Control Mechanisms

In Table 4, congestion control mechanisms are classified by type of data transport required and whether network support for congestion control is available. Thus, the *Network Support Present* row depicts scenarios when some form of support from the network is available to applications upon congestion; conversely, the *Network*

[2]Note that ASCC can be applied only to applications that can adapt their rate, i.e. a transformation can be applied over the application's data to reduce its bandwidth requirements. Prototypical examples are continuous media transfer applications.

Table 4: Congestion Control Space

Network support	Data Transfer *Reliable*	*Unreliable*
Present	II DECbit, ECN, ACC Exp. backoff	I Application-specific congestion control
Absent	III RED, Tail drop Exp. backoff	IV RED, Tail drop

Support Absent row correspond to cases in which the network drops packets — without regard to end-application— when links are congested. The *Unreliable Data Transfer* column corresponds to cases in which the data delivery need not be bit-perfect: such a service is useful for applications that can sustain some loss (such as on-line voice). The *Reliable Data Transfer* column corresponds to cases in which every bit of data must be delivered without error.

Congestion control without network support

Quadrants III and IV correspond to the current Internet: packet loss is the only network response to congestion. As discussed above, sender adaptation is the mechanism used to detect and avoid congestion. In such a network, the routers "notify" the sender of congestion using generic mechanisms such Tail Drop and RED that results in losses at the receiver. The sender is, in fact, notified of the loss by the receiver. If the data is be reliable transferred, senders adapt to congestion by exponentially reducing data rate (quadrant III). Note that upon prolonged congestion, exponential decrease of the sender's rate is the *only* mechanism that can *reliably* transfer data. Network support can only reduce the length of the congestion notification feedback loop back to the sender; the sender must allow the buffers in the network to subside and its rate must eventually match the bottleneck link capacity in order for the network to be stable. Otherwise, retransmissions from the sender will either cause the size of network buffers to increase without bound or the network will experience complete congestion collapse [65].

With unreliable data, upon detection of loss (congestion), the sender does not necessarily re-transmit. Instead, it transforms the data or alters its transmission pattern such that the amount data transmitted per unit time is reduced. Consider an example in which the sender encodes and transmits *real-time* (or on-line) video, i.e. the receiver displays the video as it is received. Upon detecting congestion, the sender may transmit re-quantized video such that the transmission uses less bandwidth. Another alternative may be to only transmit certain frames; this would reduce the frame rate and the required bandwidth. This behavior is fundamentally different from that of a reliable data source. Upon detecting congestion, a reliable data source reduces its rate but still transmits the same data. In contrast, an unreliable (adaptive) data source may *transform* the data itself to reduce its bandwidth requirement.

Congestion control with network support

The second quadrant corresponds to the case when the data must be reliably transferred and there is some mechanism, other than packet loss, that is supported by the network to signify congestion. Under the DECbit scheme [86, 87] routers in the network detect congestion and set a congestion-indication bit on packets traversing from the source to the destination. This congestion notification is communicated back to the users through a higher (transport-layer) acknowledgment. This Explicit Congestion Notification (ECN) [88] scheme for the Internet is an adaptation of the DECbit scheme and proposed for the Internet. The Active Congestion Control (ACC) scheme [48] applies active networking ideas to feedback congestion control, specifically TCP. Under ACC TCP, active routers generate a congestion signal for the TCP source, thus shortening the time required for congestion notification to reach the sender. The congestion signal is in the form of a TCP acknowledgment that advertises a TCP window that is half the size of the current window. Further, ACC-augmented routers install a filter on the *previous* router on the connection's path. This filter discards all packets from the sender that violate the newly established advertised window value.

In case of unreliable data transfer, it is possible for an active network to emulate the behavior of the sender and perform "exactly" as the sender would have—had the sender been aware of the current network conditions. Thus, upon congestion, the network itself can transform application data in order to reduce its bandwidth requirements. This observation is the motivation behind *application-specific congestion control* (ASCC), shown in the first quadrant in Table 4, and is the topic of discourse for the rest of this chapter.

4.1.3 Roadmap

We present details of ASCC in Section 4.2, develop viable architectures in Section 4.2.1, demonstrate techniques and outline mechanisms that can support application-specific congestion control of unreliable data in Section 4.3. In Section 4.4, we show results for the application of our mechanisms to the unicast delivery of MPEG video. We extend our techniques to network support for best effort multicast video delivery in Section 4.5 and present simulation results of multicast topologies in Section 4.5.4. A general discussion of issues that we did not address in our experimental setup is presented in Section 4.6, and a survey of related work is provided in Section 4.7. Finally, a summary of the results and some extensions are given in Section 4.8.

4.2 Application-specific congestion control

In the following discussion, we assume that the application-specific congestion control is applied on a per-*flow*[3] basis. We define a flow to be a sequence of packets

[3]In later work [16], we have successfully applied application-specific congestion control techniques on a per-application basis.

each of which contain a particular label (called a flow identifier). Packets in a flow may or may not originate and terminate at the same pair of end-points; however, all packets in a flow contain the same flow identifier. Further, all user-state at routers is instituted on a per-flow basis.

Generically, application-specific congestion control operates as follows:

> Based on triggers that indicate congestion control should take place, flow state is examined for advice about how to reduce quantity of data.

The important components of this generic model are the *triggers* responsible for initiating congestion control, the *flow state* that contains the specific advice for this flow, and the *reduction techniques* defined by the network and made available to the users. An important feature of this model is its consistency with traditional best-effort service. That is, a flow provides *advice* about what to do with its data. The network node is not required to take the advice, and may apply generic bandwidth reduction techniques. Thus, the techniques are inter-operable with standard best-effort service and place no undue burden on applications or network nodes.

4.2.1 Architectural Requirements for ASCC

Active networks provide mechanisms to place flow-specific computations at routers [59, 103]. The packets of a particular flow form the *data* for these computations. Per-flow advice can be carried in the flow packets themselves and/or stored at active nodes. The issues of how to incorporate state and computation in routers is moot in an active network as the user network interface provides the requisite mechanisms. However, it is also useful to consider ASCC as an extension to a best-effort networks. Such a network is not *active* per-se but has been augmented with extra functionality beyond packet forwarding. Specifically, the mechanism for performing ASCC already exists in the network; however, the issues of state establishment etc. must be resolved within the confines of existing network protocols (e.g. IPv4 or

IPv6). In the rest of this section, we discuss each component of ASCC. Our discussion is in terms of a best-effort network augmented with ASCC mechanisms. In our discussion, we assume there is no network-level notion of bandwidth guarantees and the network does not provide explicit notification of congestion back to the source. Our conclusions apply to active networks as well.

- **Triggers**

  Triggers for initiating congestion control have been studied, both for end-to-end mechanisms as in TCP [65] and for intra-network mechanisms as in ABR protocols [22] and RED [50]. This issue is orthogonal to the other components of ASCC, and we have not investigated congestion triggers in detail. Instead, we use two simple congestion triggers: (1) an output queue is full (like in Tail Drop) or (2) the occupancy on an output queue crosses a certain high-water mark (as in RED).

- **State Establishment**

  There are two orthogonal components to the establishment of state for ASCC. Firstly, a mechanism is required that associates a particular flow with a particular form of application-specific processing. Once such a computation has been established, finer grained changes to flow state, possibly due to processing within the network, may have to be communicated to routers on the end-to-end path.

  We have used network level packet options (IPv4 options [14], per-hop options in IPv6 [59, 58]) to associate computation with specific flows. Explicit out-of-band protocols, e.g. the CUI protocol (discussed in Section 3.4.8), or Active RSVP [24] can also be used to set up this state.

  Finer grained updates in the flow state can originate at two locations. The source application itself may provide some advice (state) about the packets in the flow, e.g. identifying a set of packets as one application level unit. An

ASCC router could use this information to drop all the packet in the unit if any one of these packets have to be dropped. The second method of advice establishment occurs *en route*, independent of the source. Based on conditions at network nodes, congestion control advice may be attached to packets to influence processing at downstream nodes. This advice is in reaction to intra-network events that is not known at the source of the data. For example, discarding a specific packet may elevate the priority of a subsequent packet. This change in packet priority then has to be advertised to downstream routers.

  In our work, all fine grained state updates are carried in-band with the data in a network level header option [14] or as an application level header [59].

- **Data Reduction**

  In its most powerful and general form, ASCC might reduce bandwidth requirements by applying *transformations* to reduce the quantity of data at a congestion point. Such transformations "shrink" the data while preserving as much useful information as possible.

  Transformations are usually computationally complex [5, 4]. In our implementation, we have focused on the special case of *intelligent discard* of data. Using a natural extension of packet-level discard [7], we allow applications to define units based on application semantics, with aim of discarding the entire unit if any portion must be discarded. We have also extended our reduction to consider priorities within application units, and discarding entire lower priority units in order to accommodate higher priority units.

In summary, the architectural mechanisms required to implement ASCC are as follows:

- Support for storage and retrieval of per-flow state at the network node.

- Ability to add a small amount of "advice" information to each packet where it can be inspected (and possibly modified) by network nodes.

- A per-flow queue of packets waiting for output, the ability to examine the advice attached to each packet in the queue, and the ability to remove packets from an arbitrary point in this queue.

Note that it is not required that every network node have these mechanisms; the scheme backward compatible, and nodes that do not support it are free to ignore the user-specified advice.

4.3 ASCC applied to MPEG Video Streams

To illustrate specific mechanisms and evaluate performance, we focus on the use of congestion control advice to improve the best-effort service provided to MPEG [51], a widely used standard for video coding. In our implementation, we only consider MPEG-1 streams.

MPEG encodes video as a sequence of *frames*, hierarchically encoded into three layers. An I-frame is encoded independently of all other frames. A P-frame may be encoded relative to another I- or P-frame. A B-frame may be encoded relative to two other I- or P-frames. A *Group of Pictures* (GOP) consists of a single I-frame followed by structured order of P- and B-frames. The video is encoded as a sequence of GOPs; the structure of the GOPs may change within a stream.

The exact dependencies between frames will depend on the particular video being encoded. For our simulation experiments, we model the dependencies in the following way. First, we assume a GOP structure encoded as the 10-frame sequence "IBBPBBPBBP" and transmitted as "IPBBPBBPBB" [4]. We assume that each P-frame is encoded using the I-frame that starts the GOP; we assume that each B-frame is encoded using both the preceding P-frame in the transmission sequence and the I-frame that starts the GOP. These assumptions are consistent with the encoding rules for MPEG. (We have experimented with both fixed and varying

[4] MPEG transmission typically occurs in a different order than the playout order to facilitate decoding. This is because the B-frames require the subsequent P-frame in order to be decoded.

GOP structures for our unicast emulation experiments.) Since none of our discard algorithms depend specifically on the structure of a GOP[5] we expect their benefits to be relatively independent of particular GOP structures.

4.3.1 MPEG Discard Algorithms

The basic goal of our discard algorithms is to transmit complete and useful application level units. For example, in case of MPEG, a frame is the smallest decodable application-level unit. However, a single frame may be transmitted in multiple packets; our discard algorithm tries to transmit all the packets that constitute a frame. Even if a frame is received in its entirety at the receiver, it may not be useful as some other frames it depended on were dropped in the network. Thus, our techniques will try to transmit only complete frames that can be decoded at the receiver. Thus, our reduction technique exploits two features of MPEG: the application-layer frame units and the decoding dependencies across frames.

We will discuss four algorithms for MPEG discard. They are summarized in Table 5

Table 5: ASCC algorithms for MPEG transfer. The Frame drop and GOP drop algorithms are able to recognize application-level units (frames in the case of MPEG) while the Static Priority and Tail Drop algorithms operate on individual packets.

Algorithm	Unit	Priority
Tail drop	Packet	None
Static priority	Packet	I > P > B
Frame drop	Frame	None
GOP drop	GOP	I > P > B

- The **Partial Packet Discard** mechanism is for baseline comparisons; it defines each IP packet to be an application-unit. Datagrams are discarded if they cannot be buffered in the output queue.

[5] Of course, the GOP should not contain only I-frames.

- **Static Priority** imposes a packet level priority on the stream. Under this scheme P- and B- packets can be evicted to store an I-packet (i.e., a packet carrying an I-frame)[6]. Similarly, a P-packet that cannot be enqueued may evict a B-packet already in the queue.

- Our **Frame Level Discard** mechanism defines a unit to be an MPEG frame. The advice given is to queue a datagram if and only if its corresponding frame can be queued in its entirety. We maintain state for each frame that is being discarded or buffered, and use this state information to decide, in constant time, to buffer or discard a particular datagram. Note however that without a one-frame buffer in which frames are assembled, the frame-level discard technique cannot guarantee that partial frames will not be transmitted.

- We further consider a mechanism that identifies dependencies between units. Our **Group of Picture (GOP) Level Discard** maintains state about the type of frame discarded. In case an I frame has been discarded, we discard the dependent P- and B-frames as well.

Discussion

The Tail Drop and Static Priority algorithms do not recognize frame boundaries. Thus, if the network is congested, these discard algorithms may transmit partial (undecodable) frames. Conversely, the Frame Drop and GOP Drop algorithms *will not* transmit partial frames. (As we will see later, it is possible for some implementations of these algorithms to transmit partial frames when the trailing packets of a frame, whose leading packets have already been transmitted, cannot be accommodated in a network buffer). However, the Frame Drop algorithm, due to its lack of recognition of frame dependencies, may transmit complete but useless frames.

[6]Typically, I-frames are much larger in size than P- or B-frames. Thus, to accommodate one I-frame, multiple P- and B- frames may be evicted from the output queue.

4.3.2 Implementation

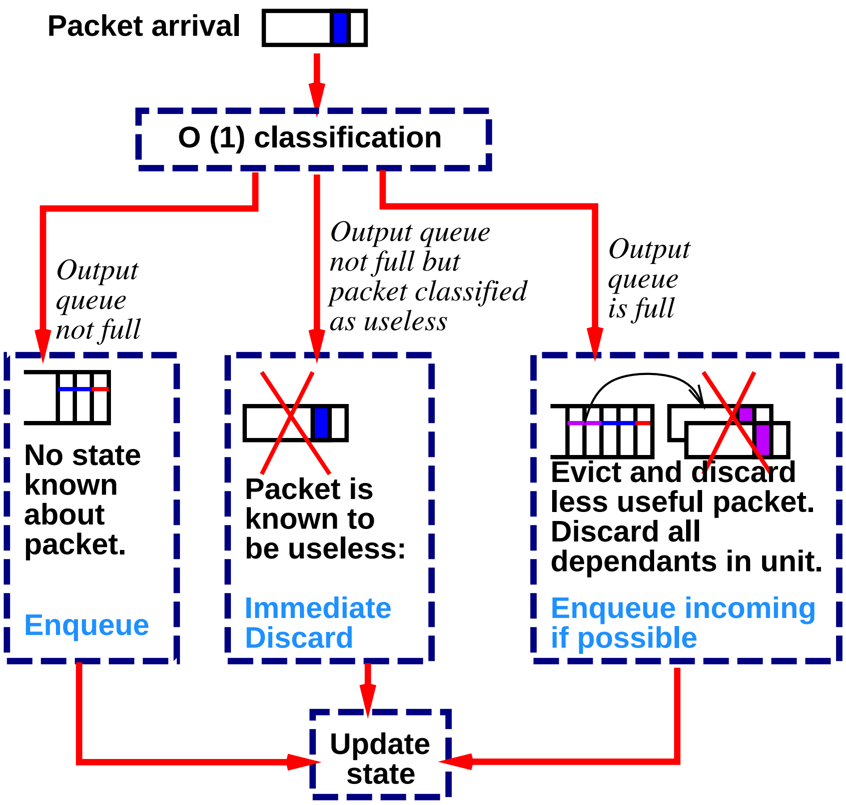

Figure 22: A schematic diagram of the implementation steps for MPEG discard algorithms. The Tail Drop and Static Priority algorithms do not maintain any state. The queue eviction step is required only for algorithms that maintain priority (GOP Drop and Static Priority).

Figure 22 shows the generic steps in the implementation of all four algorithms. The specific algorithms are mapped onto the generic implementation as follows:

- **Tail Drop** The Tail Drop algorithm does not classify incoming packets, i.e. the input classification step is null. If the incoming packet can be accommodated in the output queue, it is enqueued. As there is no notion of priority, incoming packets are dropped if the output queue is full. The Tail Drop scheme does not maintain any per-flow state.

- **Static Priority** Like Tail Drop, the Static priority scheme does not maintain state or classify incoming packets. If an incoming packet does encounter a full output queue, its priority is matched against the priorities of packets from the *same* flow in the output buffer. If possible, lower priority packets from the same flow are evicted from the output queue and discarded such that the incoming (higher priority) packet can be transmitted.

- **Frame Drop** The frame drop scheme maintains per-flow state that consists of a frame number that is currently being dropped. The frame number of incoming packets (carried in a header in the packet) is checked against this state. Upon a match, the incoming packet is dropped immediately. Otherwise, if the incoming packet can be accommodated, it is enqueued in the output queue. If the incoming packet cannot be enqueued, all packets in the output queue with the same frame number are dropped. Further, the node state is updated to discard all subsequent packets with the same frame number.

- **GOP Drop** The GOP Drop scheme maintains state about the sequence number of the packet being dropped and its type. Assume a packet arrives and there is no state at the node about this packet. If there is sufficient space in the output queue, this packet is enqueued. If not, then *frames* with lower priority are discarded till this packet can be enqueued. Thus, if the packet that cannot not be enqueued belongs to an P-frame, then B- frames are discarded to accommodate this packet. If the packet belongs to an I-frame, both B- and P-frames are discarded to accommodate this packet. Further, if a P-frame is discarded, all B-frames that depend on this P-frame is also discarded.

  It is possible that an incoming packet cannot be accommodated in the queue, i.e. there are not enough frames in the output queue of lower priority. In this case, all the packets from this frame that are already in the queue are discarded. Further, all frames that depend on this frame in the queue are discarded. The per-flow state is updated such that all incoming packets of the discarded frame and all subsequent dependent *frames* are discarded upon input. Thus, if a P-frame cannot be enqueued, the subsequent packets of this P-frame and its dependent B-frames are discarded when they arrive at the router. An entire GOP worth of packets are dropped at input when a packet belonging to an I-frame cannot be enqueued.

The GOP Drop scheme is the most involved and is implemented using three separate mechanisms. The first two mechanisms involve discarding packets that are already in the output queue.

- **Discard-Other.** When a packet arrives and the queue is full, packets belonging to this flow but of lower priority are discarded in order to fit the current packet in the output queue. Further, if a packet from a particular frame is discarded, all packets that depend on the discarded packet are discarded. In our implementation, B-frames are always discarded before P-frames. Thus, a P-frame is discarded if and only if an I-frame is congested and the output queue does not contain any more B-frames.

- **Discard-Own.** When a packet arrives and it must be discarded, then all packets already enqueued and that belong to the current frame are discarded as well. For example, if a packet belonging to a B-frame encounters a full queue, it must be discarded. The Discard-Own mechanism is used to discard all the other packets that belonged to the current frame and were already enqueued in the output queue.

- **Discard-On-Arrival.** The third mechanism involves discarding arriving packets before they enter the output queue. If a packet is discarded (due to congestion or other discards described above), the node state is set such that all subsequent packets that depend on the discarded packet are discarded upon arrival. For example, if a packet from an I-frame is discarded, all subsequent dependent packets (in this case, all packets until the next I-frame) are denied entry to the output queue. Similarly, if a packet from a P-frame is discarded, all arriving B-frames that depend on this P-frame are discarded on arrival, etc. Note that for this mechanism, only a few bits of state need to be set (and reset), and access to the output queue is not required.

4.3.3 State and Computation Requirements

All three mechanisms described above can be implemented with a relatively small amount of per-flow state and per-packet-arrival computation. The Discard-Other mechanism must be able to (1) determine if enough lower priority packets are in the queue to make room for the incoming packet, and (2) remove specific packets in the queue. The first task can be accomplished by keeping per-flow state that records the cumulative size of B-packets and the cumulative size of P-packets that are currently in the queue. This state is updated whenever a packet for this flow enters or departs the queue. The second task requires an interface to the queue that supports deletion based on a pattern match. If the streams are properly labeled, the pattern match can be extremely efficiently implemented (as is the case in our implementations [19, 59] and simulations). These deletions can clearly be implemented in time linear in the length of the queue; more efficient implementations are possible (by keeping a separate queue per frame type). The Discard-Own mechanism can either be implemented similar to the Discard-Other (deletion based on pattern-match), or be implemented by keeping a one frame buffer in which complete frames are reassembled before entry to the output queue. The reassembly buffer makes it easier to implement Discard-Own and guarantees that frame fragments are not transmitted; however, it also adds a reassembly latency at each node. In our case, we do not implement the one frame reassembly buffer.

The incoming discard mechanism must keep state recording the frame type and frame sequence number of the packets of each frame type that were discarded. A simple finite state machine is used to decide whether to drop an incoming packet, using the information on the packet that was dropped and the contents of the incoming packet header. The decision consists of a small number of comparisons.

In Section 4.6.1 we examine the performance of the GOP-level discard scheme with some, but not all, of the three mechanisms enabled.

4.3.4 Router Information

As indicated above, the reduction techniques make use of a combination of information carried in packets (e.g., frame type and frame sequence number) and information resident at routers. The information resident at a router might be installed off-line and out-of-band (e.g., by a routing manufacturer) to provide a value-added service to MPEG video. Alternatively, the router might support a programmable interface that allows some form of on-line installation of state and computation. Such interfaces are being developed under various Active Networking projects [3, 103].

4.3.5 Congestion Triggers

As noted before, we consider two types of triggers. In the first, we detect and respond to congestion only when data arrives that cannot fit in the output queue. All three mechanisms mentioned above use this trigger. We also consider an "early" trigger, following the Early Packet Discard of Romanow and Floyd [90]. This trigger detects and responds to congestion when the output queue occupancy exceeds a certain threshold.

4.3.6 Evaluation Metrics for MPEG

We use the following metrics to evaluate the quality of the received streams.

- **Fraction of I-frames decoded.** An entire MPEG GOP can be lost if the associated I-frame cannot be decoded (as all the frames in the GOP may depend on the I-frame). We report the fraction of transmitted I-frames that the receiver is able to decode. In our experience, this metric also correlates "well" with the perceived quality of the received video stream.

- **Fraction of total frames decoded.** We report the number of decodable frames over the total number of frames sent by the sender. This "end-user-centric" metric takes into account that after the decode process, the I-, P-, or B-frames each account for the same time duration of playback (i.e., 1/30-th of a second). Thus, the type of the decoded frame, from the end-user's point of view, is irrelevant. Of course, in order for a P- or a B-frame to be decoded, all the frames they depend on must also be received and decoded.

- **Fraction of data received.** We also note the usual metric of amount of data received over the amount sent. As we will see, in case of hierarchically encoded media, this metric may not correlate well with fraction of data that is actually useful.

- **Useless data received.** As we have noted, some data may be received that form partial frames, or frames that cannot be decoded. We report the fraction of data that is received but is useless at the receiver. This is a "provider-centric" metric as it accounts for wasted bandwidth on the network links that may as well be used for some other purpose.

- **Detailed results.** We present details of particular runs in the form of decoded frame sequence numbers versus time and decoded frames per second over time. For emulation results, we report the decoded signal-to-noise ratio of the received stream. The decoded signal-to-noise ratio is computed as follows: MPEG is encoded by processing a set of static pictures. The encoding process involves a quantization step (as part of a discrete cosine transformation) that introduces noise in the encoding. There is a signal-to-noise (SNR) ratio computation that quantifies the level of noise introduced during the encoding of each frame. Thus, each transmitted MPEG frame has a particular signal-to-noise ratio. We compare each received frame to the original picture and re-compute the SNR. If the received frame is identical to the encoded (and hence transmitted) MPEG frame, then the computed SNR is the same

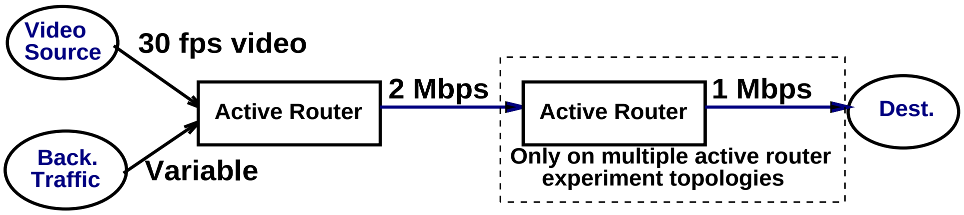

Figure 23: Emulation topology for unicast video streams: The active router functionality is implemented within a SunOS 4.1.3 kernel. The end-hosts are Solaris 2.5.1. The second router is present only for multiple active router experiments; otherwise, the first active router is connected directly to the destination.

as before. However, if a frame could not be received or could not be decoded, we use the last decoded frame to compute the SNR. Obviously, it is possible that the last decoded frame is different from the picture it is being compared to — this manifests as a higher level of noise in the encoding. In effect, both the original MPEG encoding and the network transmission has served to introduce noise in the received set of frames. For this metric, higher values of SNR imply less noise and hence a better picture.

4.4 Network support for unicast video

In this section we describe a set of experiments that apply MPEG-specific congestion control algorithms to unicast streams. The results were obtained using an emulation testbed that we implemented. We describe this testbed next and report our experiences.

4.4.1 Experimental Methodology

For unicast experiments, we implemented our MPEG-specific discard schemes within the kernel of a 60 MHz Sparcstation 20 running SunOS 4.1.3. The emulation topology is shown in Figure 23. Physical connectivity was provided via 155 Mbps ATM links; each machine in the testbed was connected to a Fore ATM ASX-100 switch. IP connectivity was provided via the Fore proprietary SPANS protocol. The end hosts were Sparcstation 20 and UltraSparcs I machines, all running SunOS 5.5.1.

The MPEG video was streamed using UDP transport over IP. Application-specific processing was selected using an IP option. Each video packet carried a special IP option that contained (1) a flow identifier and (2) a tag that selected a specific discard algorithm. The congestion inducing packets also used UDP transport and carried the same IP option. Their flow identifier was different from the video packets and their discard algorithm tag always chose partial packet discard (tail drop). The size of congestion packets were chosen using packet sizes from traces collected on the backbones at Georgia Tech., and the San Diego Supercomputing Center. We also conducted experiments in which the packet sizes were chosen using the library `tcplib` [39] which produces packet sizes according to a mix of applications (such as `ftp`). Our results did not vary significantly with the choice of the background packet size generator. As such we only report results in which the size of the background traffic packets were determined using collected packet traces.

Output Buffering Packets from both sources were queued at a shared output buffer. The size of the shared output buffer was fixed for each experiment. Specific discard techniques were applied on a *per-flow* basis, i.e. under no circumstances could a video flow evict a packet from the background traffic flow and vice-versa. To induce congestion, the output buffers were serviced at specific rates.

As a rule of thumb, the amount of buffering on a router output queue is given by the expression $\alpha r B$, where r is the round-trip time on the link, B is the bandwidth on the link, and α is a scale factor (> 1). For our experiments, the round trip time on the local network is on the order of one millisecond and the bandwidth on the link is software-limited to 2 Mbps. Nominally, this would imply a buffering of 250α bytes. It is not clear such rules of thumb apply to active networks since the buffering requirements of different computations can be extremely variable. For example, the MPEG discard techniques could easily require buffering entire frames (such that only complete frames are transmitted). On the other hand, large buffers introduce more delay and jitter. As we will see in our results, for MPEG video delivery, larger buffers are more useful for generic mechanisms (like Tail Drop and Static Priority). The active schemes reach optimal performance with far less buffering than the generic schemes. In our experiments, we will report results for 8K and 24K bytes buffer sizes — this corresponds to α values of approximately 30 and 100, respectively.

4.4.2 Single Active Router

In this section, we present results from an experiment with a single router. Video packets are streamed from the video source and share an output link at the active router with packets from the congestion source. The output link is serviced at 2 Mbps. For the results in this section, the average rate of the video source is approximately 725Kbps. The average size of an I-, P-, B-frame is 12.5, 8, 2.2 K byte respectively. The arrangement of I-, P- and B- frames vary in each GOP. Each run consists of approximately 1200 frames (40 seconds of video at 30 frames per second). Each reported result is an average of 10 runs. We vary the average rate of the congestion source from 1Mbps – 2 Mbps (corresponding to an average load of 85% to an overload of 30% on the output link).

Figure 24 shows the fraction of I-frames delivered to the receiver. The average link load is plotted along the x-axis — this directly corresponds to the rate of the background source as the video source transmits data at 725 Kbps. The fraction of I-frames received is plotted along the y-axis. For this experiment, the size of the shared output buffer is fixed at 8K bytes.

The four curves correspond to four different discard techniques for the MPEG flow. (The congestion inducing flow is always tail dropped). The curve labeled "Tail Drop" corresponds to behavior that would be observed over the current Internet; the extra fraction of I-frames that are received using the "Static Priority" and "GOP discard" mechanism are direct results of the extra processing introduced inside the

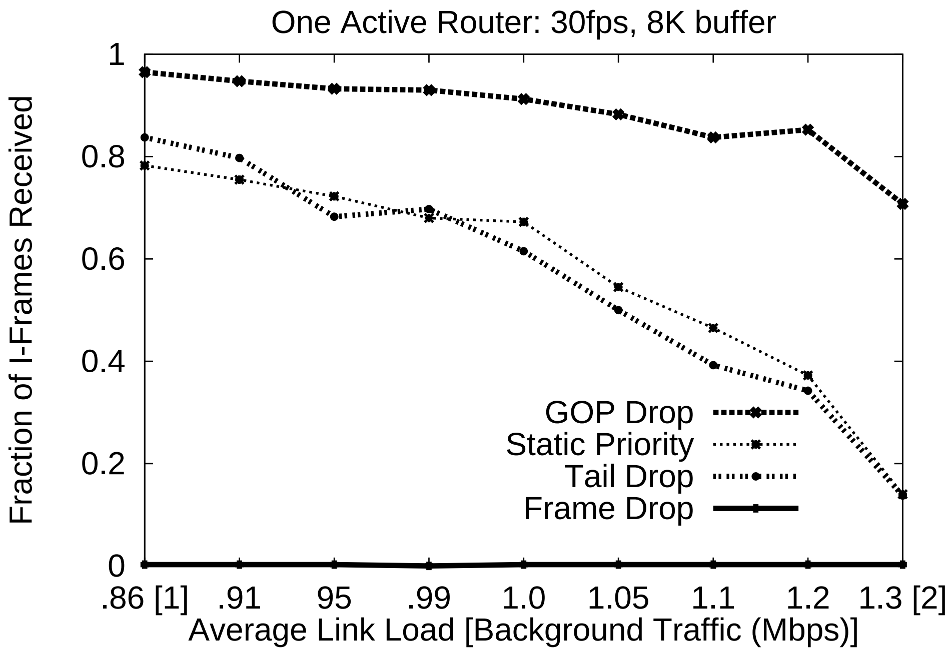

Figure 24: Fraction of I frames received when output link is congested. Note that the GOP discard scheme performs relatively better as congestion increases.

network. The most application-specific scheme, "GOP discard" outperforms all other schemes when the output link is congested. Further, the relative gains due to application-specific congestion control increase as congestion worsens. We realize that most networks usually do not operate under 30% overload, as represented by the rightmost set of points in Figure 24. However, it is interesting to note that even under such heavy congestion, the application specific scheme is able to transmit over 75% of the I-frames with only 8K bytes of buffer. The next best scheme in this example, Static Priority is only able to less than 20% of the I-frames. The Frame Drop scheme fares the worst, and is not able to transmit any I-frames with 8K bytes buffering; we address the performance of the frame-drop scheme later in this section.

For the same experiment, Figure 25 shows the fraction of received data that is discarded at the video receiver. This receiver may discard data for two reasons: an

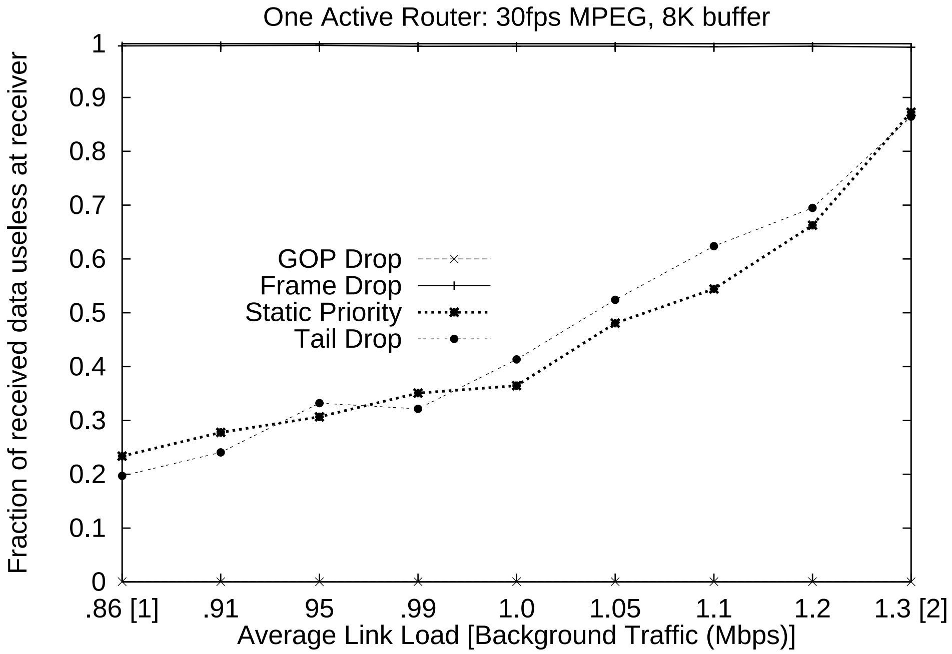

Figure 25: Fraction of received data discarded at receiver when output link is congested. The GOP discard mechanism does not transmit *any* useless data while all of the data transmitted by the Frame Drop scheme is useless. With a 8K buffer, the performance of the Static Priority and Tail Drop mechanisms are nearly identical.

incomplete frame is received (i.e. some packets from a frame were dropped at the router) and an indecipherable frame is received (i.e. though a complete frame was received it could not be decoded as another frame that the received frame depended on was dropped). The x-axis, is the same as the previous figure. The y-axis plots the fraction of received data discarded. Once again, the four curves correspond to the different video discard techniques. Firstly, we note that the application-specific scheme — GOP discard — does not transmit *any* useless data[7]. As expected, the

[7]In our implementation, the GOP discard scheme pre-emptively discards B-frames when the output queue reaches a high water mark, set to 90% of capacity in these experiments. (This corresponds to the *early* trigger mechanism discussed in Section 4.2.) It is possible for the GOP discard scheme to transmit packets from a frame and not be able to accommodate subsequent packets from the same frame. However, such implementations do transmit more frames but may end up transmitting approximately 10% useless data as well

Tail Drop and Static Priority schemes do transmit data that is useless at the receiver. Approximately, 45% of the data transmitted by the Static Priority and Tail Drop mechanisms is useless at the receiver (with a 5% overloaded link). It is important to note that the useless data consumes resources (bandwidth) at the network level along with processing at the receiver. In fact, it is better not to transmit this data at all and to use the bandwidth for some other flow entirely.

Note that with 8K bytes of buffering, all of the data transmitted by the Frame Drop scheme is useless. Further, the performance of the Static Priority and Tail Drop schemes are similar. These observations are due to the relatively small buffer size at the active node. Next we show results of the same experiment with a larger buffer size.

4.4.2.1 Changing Buffer Capacity

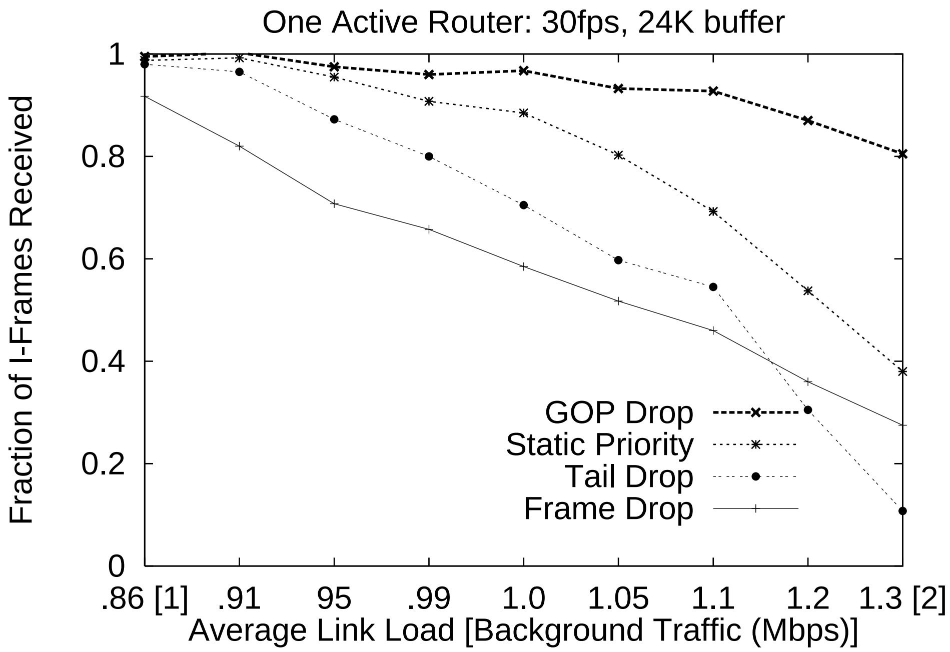

Figure 26: Fraction of I frames received with 24 K bytes of output buffering. Note that the Static Priority and Frame Drop schemes perform better with larger buffers.

Figure 26 shows the fraction of I-frames decoded at the receiver when the output buffer size is increased to 24K bytes. All four schemes perform better under low link loads. Unlike in the 8K byte buffer case, as the link load is increased, Static Priority significantly outperforms Tail Drop — this is because the larger buffer offers more opportunities for congested I-packets to evict P- and B- packets that may already be in the buffer. With a 24 K byte buffer, the Frame Drop scheme is able to transmit some I-frames (though it is still outperformed by the simple Tail Drop mechanism).

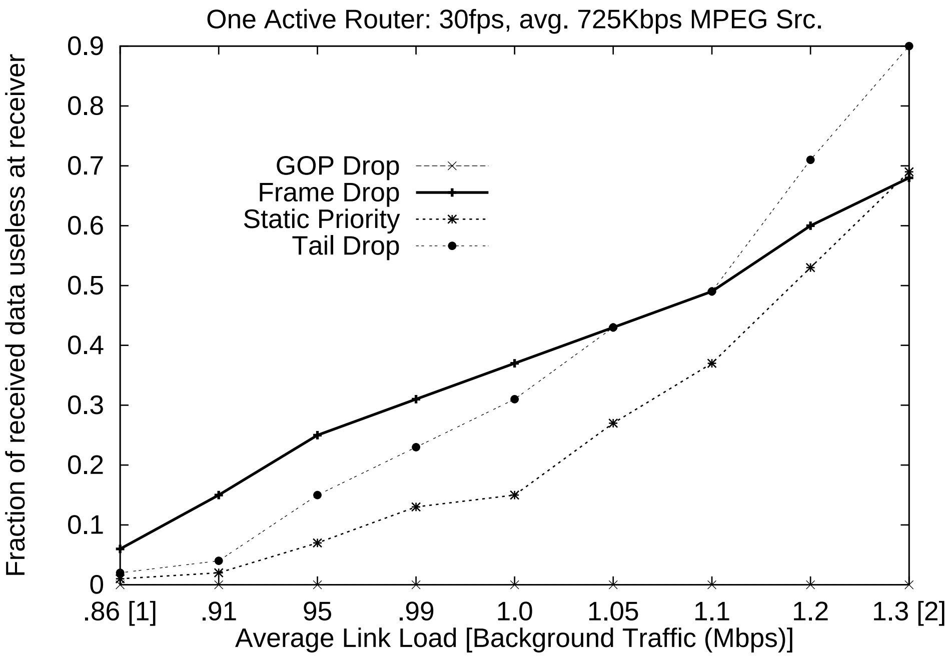

Figure 27: Fraction of received data discarded at receiver with a 24 K byte output buffer. Note that even with larger buffers, the Tail Drop, Static Priority and Frame Drop schemes transmit substantial amounts of useless data.

Figure 27 shows the fraction of useless data received with a 24K byte buffer. Once again, a significant fraction of the data transmitted by the the generic mechanisms (Static Priority and Tail Drop) is useless at the receiver. In fact, at even a 5% overload, 45% the data transmitted by the Tail Drop mechanism is useless at the receiver.

What about Frame Drop? Except for the extreme overloads, all other schemes (including Tail Drop) performs better than the application-specific Frame Drop (Figure 24) scheme. Post analysis of the data shows that the Frame Drop scheme does transmit a large number of complete frames — unfortunately, most of these are not I-frames. The Frame Drop scheme is programmed to only transmit complete frames without regard to the type of frame, thus it routinely discards the (relatively large) I-frames even when there are some B- and P-frames in the output queue. There is a lesson here: not all application-specific tweaks to the network will result in better performance. The application designer and the network programmer must pay close attention to the exact functionality being introduced in the network. Otherwise, as we see in this example, application-specific congestion control may provide *worse* service than even Tail Drop.

4.4.2.2 Bits vs. Information

Table 6: Loss details corresponding to 99% link load (average background source rate of 1260 Kbps). The 8K and 24 K are the sizes of the shared buffer at the router. Note that, without application-specific techniques, even a small fraction of lost bits manifests itself as far greater losses at the application layer.

	8K buffer			24K buffer		
Mechanism	GOP lost	Bytes lost	Waste	GOP lost	Bytes lost	Waste
TD	28.4	4.8	32.1	17.9	2.2	22.5
SP	30.2	4.5	35.0	6.9	2.2	12.7
FL	100	45.5	100	32.5	12.7	30.9
GL	4.6	12.5	0.0	1.5	7.0	0.0

In Table 6 we consider the issue of lost data versus lost application units. The table shows the fraction of bytes lost, fraction of GOPs lost (GOPs in which the I-frame was not received), and the fraction of received data that is useless.case When the congestion source generates traffic at average rate 1260 Kbps (corresponding to link utilization of 99% on average). We show results for the two buffer sizes discussed above: 8 K bytes and 24 K bytes. Note that the "Waste" column in the

24 K bytes case is, in fact, exactly the data that is plotted in Figures 25 and 27 (with the x-axis set to .99).

From Table 6, we see that for the 8K buffer size, even a 5% data loss can result in over 25% of the GOPs lost. For small buffer sizes, Static Priority performs worse than Tail Drop, 30% of the GOPs for only 2% data loss. Closer examination of the data shows that the Static Priority scheme transmits a large fraction of partial frames. These effects persist for larger buffer sizes; in the 24 K buffer case, 2% indiscriminate data loss results in about 18% of the GOPs being not decodable for the Tail Drop scheme. With a larger buffer, the Static Priority scheme is able to fare much better — it loses only 7% of the GOPs. In contrast, the application-specific GOP drop scheme transmits less data (which corresponds to a larger fraction of bytes lost). However, the fraction of affected GOPs are far less: 5% for 8 K buffers and 2% for 24 K buffers. The Frame Drop scheme is not able to transmit a single I-frame with a 8 K buffer size (as all of the I-frames were larger then the size of the output buffer). Thus, all of the data that it did transmit was absolutely useless at the receiver.

In all of these experiments, the source does not adapt its rate even when the network is highly congested. This is *plausible* for adaptive applications as its data does not need to be re-transmitted. However, some adaptive sources may transform their data upon notification of congestion. The congestion notification is via application-layer end-to-end feedback feedback in a best-effort network. In the next section, we present results of experiments that take such end-to-end feedback into account. We describe an end-to-end adaptation scheme we have implemented for MPEG streams and present our results.

4.4.2.3 End-to-end Feedback

Our end-to-end congestion control scheme works as follows: The receiver notifies the sender about the state of the path with both positive and negative acknowledgments. Upon receipt of a congestion notification, the sender exponentially its sending rate.

The sending rate is not decreased below a specified minimum rate (in order to sustain a minimum frame rate). The sender additively increments its sending rate when it receives positive acknowledgments (i.e. a no data loss signal) from the receiver. The rate is increased at a quanta that is proportional to the frame rate. For the experiments that follow, the receiver sends one congestion report every 10 packets, the minimum frame rate at which the video is sent is set to 1 frame per second.Upon receiving a positive acknowledgment, the sender increases its rate by 1 frame per second. Note that for these experiments, the sender does *not* transform the data. Instead, the original set of frames are served at a lower frame rate.

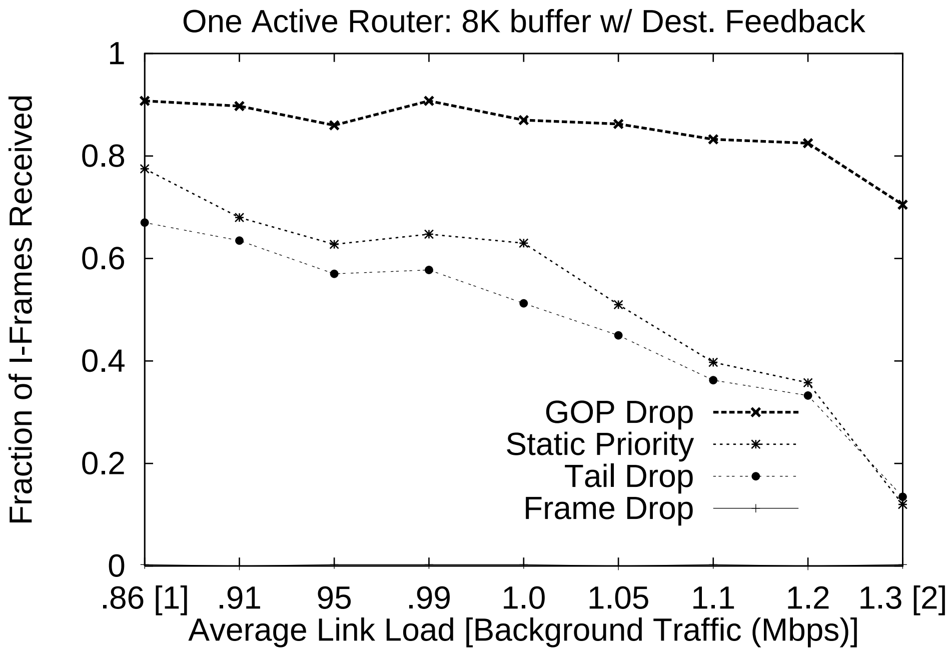

Figure 28: Fraction of I-frames received with a 8K buffer and end-to-end feedback active. Note that the performance with feedback is not significantly different from performance without feedback (Figure 24). This is because the background traffic does not back off during congestion.

Figure 28 shows the effect of our destination feedback scheme with an 8K byte buffer. As the background traffic source is non-adaptive, the introduction of the feedback scheme does not dramatically change the fraction I-frames received. In fact, as the MPEG streams back off under congestion, more of the shared buffer is allocated to the background source. This results in 5–15% *worse* performance for the generic mechanisms with feedback enabled. Note that the GOP drop scheme is complementary to end-to-end feedback mechanisms — its performance is not adversely affected by the introduction of destination feedback.

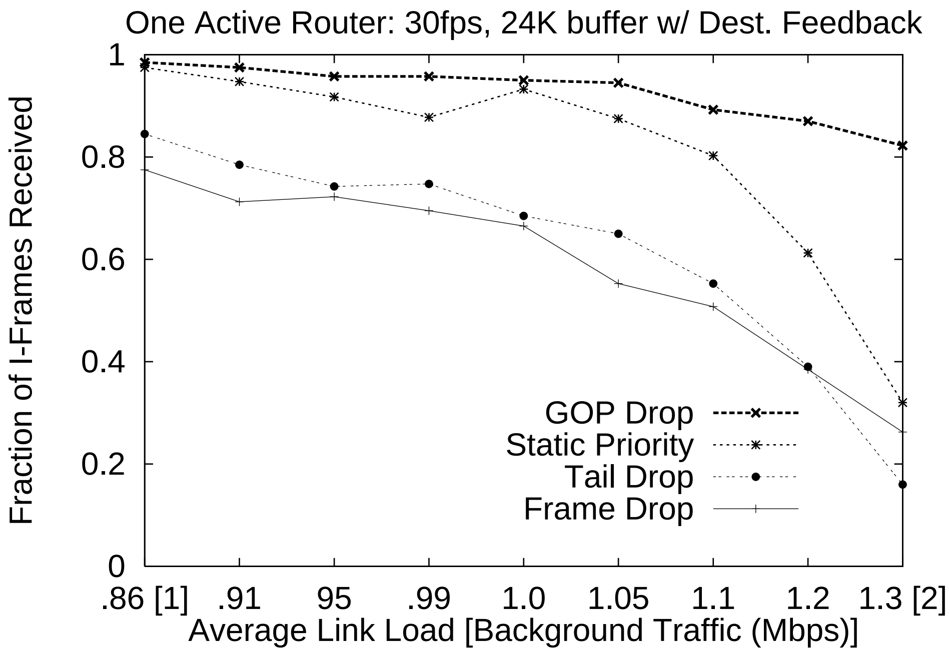

Figure 29: Fraction of I-frames received with a 24 K buffer and destination feedback active. Note that under low link loads, Static Priority performs as well as GOP Drop.

Figure 29 shows the same experiment repeated with a 24K buffer size. The larger buffer along with source backoff allows the Static Priority mechanism to perform on par with the GOP Drop mechanism when the output link is not heavily congested. However, as the load on the output link is increased, the performance of the Static Priority mechanism degrades rapidly. This is because the larger buffer is no longer able to absorb the congestion due to the background source. Further, the MPEG stream backs off and does not receive its fair share of buffering in the output queue. The GOP drop scheme is able to deliver over 80% of the I-frames even under heavy overloads with end-to-end feedback adaptation enabled.

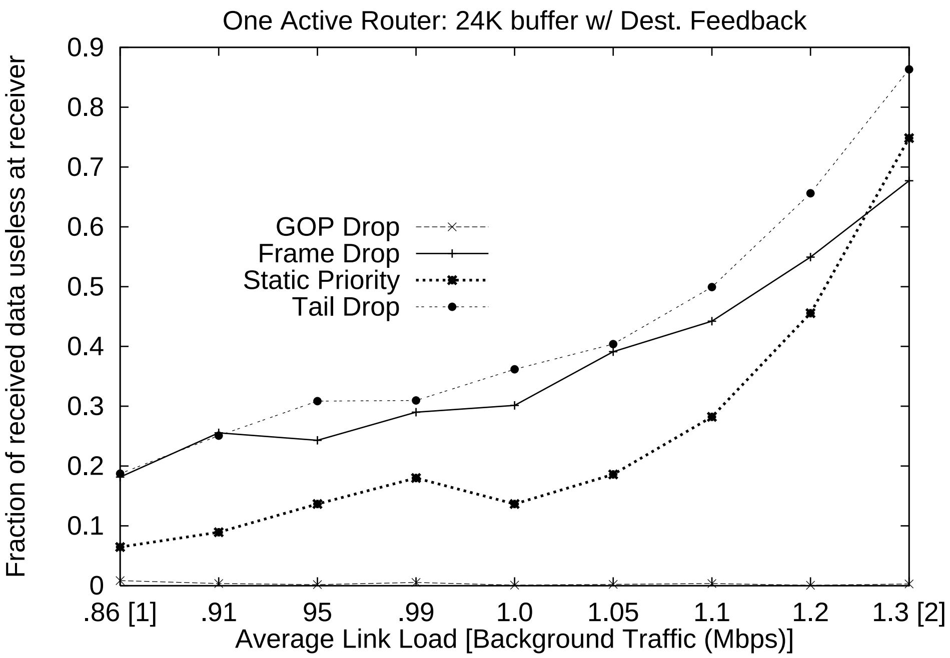

Figure 30: Fraction of received data that is useless: 24K buffer and destination feedback enabled. As noted before, when a link is congested, a significant fraction of data transmitted by generic mechanisms is useless at the receiver.

Figure 30 shows the fraction of received data this is useless at the receiver. The active routers have 24K bytes of buffer in its shared queue and the destination feedback scheme is active. Note that as the output link is loaded, the share of the queue devoted to the MPEG stream decreases. Thus, the amount of useless data transmitted by the Static Priority scheme increases significantly.

4.4.2.4 Effect of Output Queue Manipulation

In Section 4.3.2, we noted how the GOP Drop scheme can be decomposed into three separate mechanisms: Discard-on-Arrival, Discard-Own, and Discard-Other.

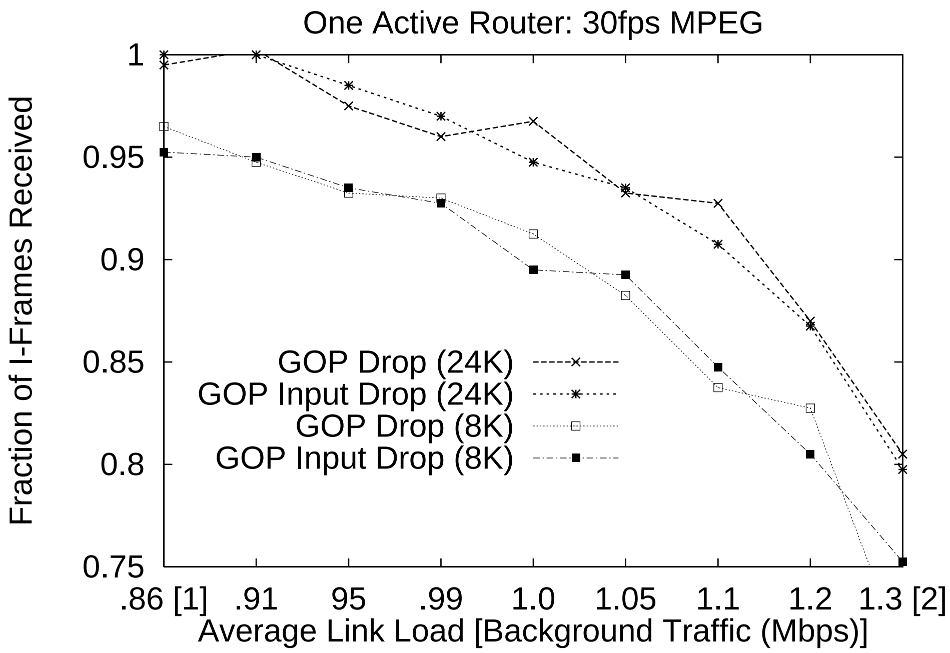

Figure 31: Comparison of the different mechanisms that comprise the GOP Drop scheme. For each of 8K byte and 24K byte buffer sizes, we plot the fraction of I-frames received when the GOP drop scheme is implemented completely (GOP Drop in the figure) versus when only the Discard-on-Arrival mechanism is implemented (GOP Input Drop in the figure). Note that in these experiments, most of the benefit is derived with just the (relatively more efficient) GOP Input Drop scheme.

The Discard-on-Arrival scheme does not require access to the output queue but may discard packets upon arrival purely based on the per-flow state kept at the node. Figure 31 shows the fraction of I-frames received for different buffer sizes. Further the curves labeled "GOP Input Drop" correspond to cases when only the Discard-on-Arrival mechanism is used — i.e. no packets are evicted from the output queue. The effects of increasing the output buffer size is evident in the figure. More interestingly, the Discard-Own and Discard-Other mechanisms do not significantly increase performance. For small buffer sizes, the queue manipulation techniques are useless (as not many frames are buffered in the queue). For larger buffer sizes, the queue manipulation is able to transmit more I-frames under heavy congestion. We would expect the queue manipulation techniques to perform better as even larger output queues are considered. It should be noted that in our case, the I-frame is transmitted before all of the frames that depend on it. In cases when a unit on which other units depend on is transmitted *after* the dependent units, the Discard-Other scheme is likely to provide more benefit (especially with respect to fraction of useful data transmitted). In case of MPEG streams however, it may be *more* efficient to implement a variant of GOP drop (Discard-on-Arrival only) than even a priority scheme like Static Priority (which may require scanning the queue for lower priority packets or keeping more complicated data structures such as overlay queues.).

4.4.2.5 Signal-to-Noise Ratio

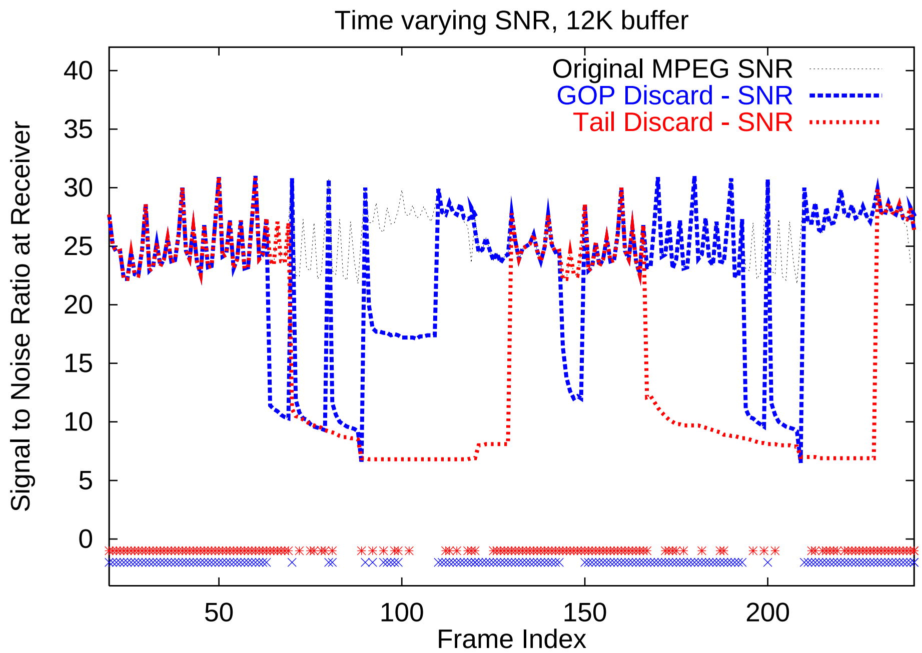

Figure 32: Signal to noise ratio of the decoded MPEG stream for a segment of one particular run. End-to-end feedback is enabled. The points below the x-axis correspond to complete frames received under each scheme. In this segment, the GOP Drop scheme manages to receive all but one I-frame (in this MPEG stream, I-frames have frame index 0, 10, 20, ...).

Figure 32 shows the decoded Signal-to-Noise ratio of the frames for one representative run of the experiment. In this case, the end-to-end feedback is enabled, and the buffer size is set at 12 K bytes. The points above the x-axis represent the SNR of the received frames. The points below the x-axis represent indices of complete frames received. Under GOP Drop, 163 of 220 total frames are received, while under Tail Drop, 148 total frames were received. However, the GOP discard mechanism was able to transmit 22 of the possible 23 I Frames, while Tail Drop was able to transmit only 12 I Frames. From Figure 32, it is clear that Tail Drop is susceptible to periods of catastrophic losses (as seen in indices [60—120], and [160—230]). During the same period, GOP discard experiences heavy frame losses as well but is able to transmit the crucial I frames (except for frame index 100), thus preserving some picture quality. This leads to a visibly graceful degradation of the signal (indices [90—110]) – and recovery from signal loss is much quicker than Tail Drop (e.g indices [190—210]). In this particular run (as with all of the runs in all of the experiments), the Tail Drop mechanism did transmit more bits than the GOP drop mechanism.

4.4.3 Multiple Active Nodes

In this section, we present results with two active routers. The topology is shown in Figure 23 (with the second router enabled).

Figure 33 shows the fraction of I-frames received when data is transmitted through two routers. Once again, the x-axis is the rate of the background traffic (which corresponds to the load on the 1 Mbps bottleneck link) and the y-axis plots the fraction of I-frames. For these particular results, the buffer sizes at each router are 8K bytes. With a load of 92% on the bottleneck link, the Tail Drop and Static Priority schemes both transfer 65% of the I-frames while the GOP Drop mechanism transfers over 95% of the I-frames. Under extreme overloads, the fraction of I-frames transmitted by the non application-specific schemes drops to zero.

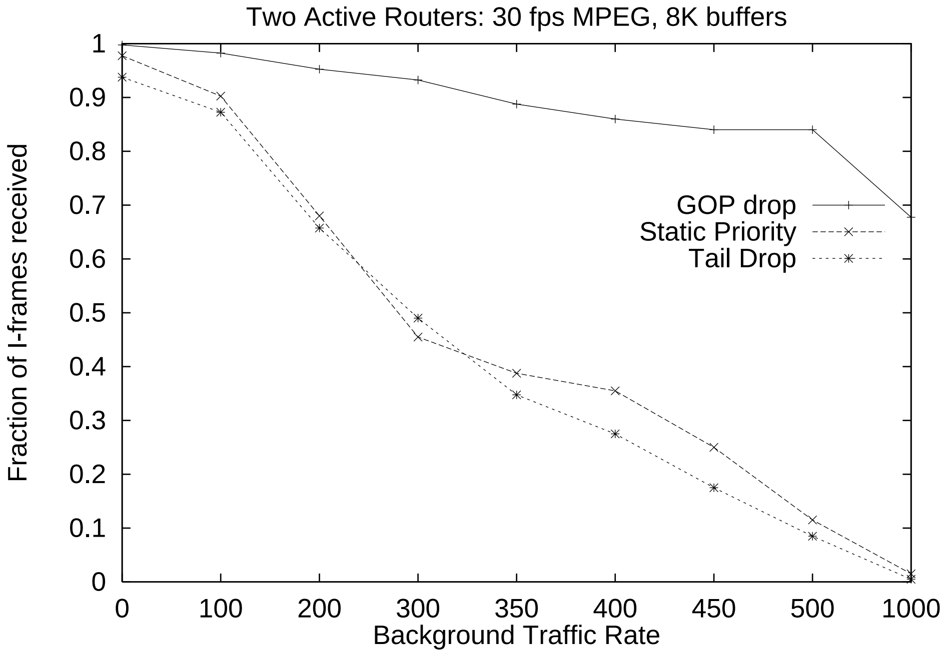

Figure 33: Fraction of I-frames decoded at the receiver. There are two active nodes in this topology; the buffer size at each node is 8 K bytes.

Under similar overloads, the GOP Drop scheme is able to transmit close to 70% of the I-frames. Note that the Static Priority scheme performs slightly better than the Tail Drop scheme as congestion worsens; however, the relative gain due to the application-specific scheme is much higher.

Figure 34 shows the fraction of received bytes discarded for the same experiment. The x-axis is the rate of the background traffic and the y-axis is the fraction of received bytes that are useless. The performance of the non application-specific schemes are even worse in this scenario: when the second link is 92% loaded, around 40% of the transmitted data is useless. Under heavy overloads, all of the transmitted data is discarded at the receiver.

Figure 35 shows the results of repeating the previous experiment with our end-to-end feedback scheme enabled. In this case, for all three schemes (especially the

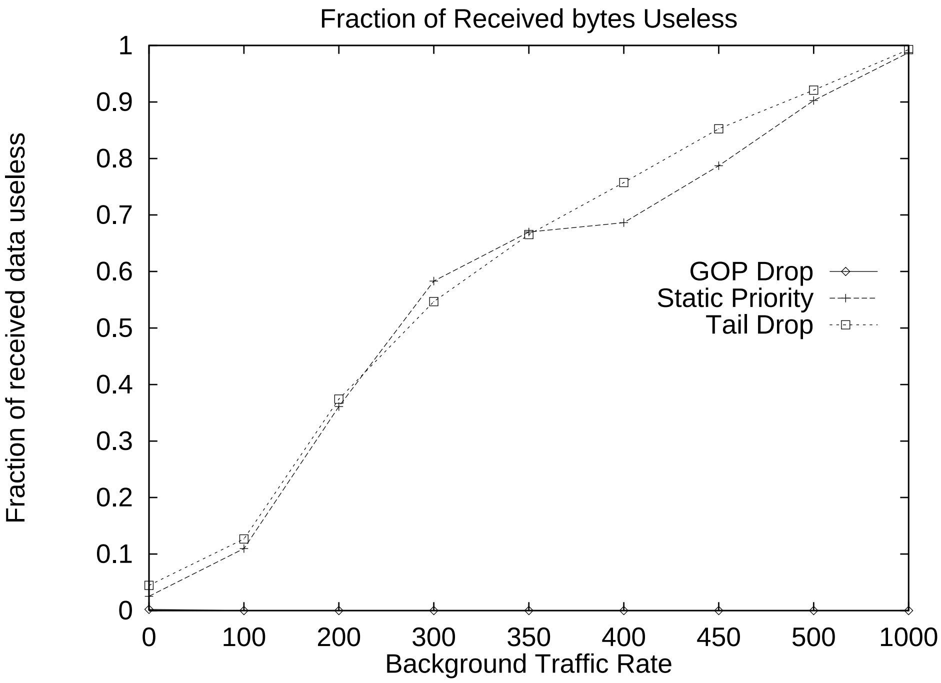

Figure 34: Fraction of received data that is useless at the receiver. Data is transmitted through two active nodes with 8K bytes buffer each. The fraction of useless data transmitted by generic mechanisms increases when the data traverses through multiple bottleneck nodes.

Tail Drop and the Static Priority scheme) the fraction of I-frames decoded degrades more gracefully as congestion increases. Analogously, we see (in Figure 36) that the fraction of useless data transmitted also increases more gradually.

4.4.4 Discussion

We have experimented with different topologies, other MPEG streams, and with other mechanisms (such as RED). The general trends are the same: non application-specific schemes transmit more bits but less information. In our experiments, RED performs worse then Tail Drop as the random discard of packets tends to produce more fragmented frames (that cannot be decoded).

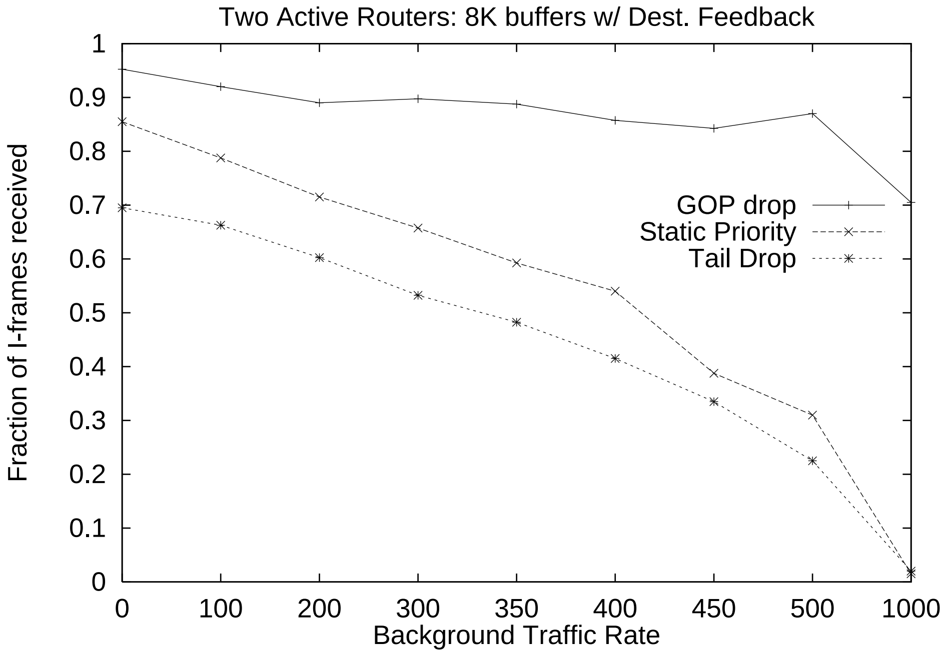

Figure 35: Fraction of I-frames received with end-to-end feedback active. Note that while traversing multiple nodes, Static Priority outperforms Tail Drop even with a 8K byte buffer at each node.

Fundamentally, our conclusions are the following: it is possible to design efficient application-specific congestion control schemes for MPEG streams and such application-specific congestion control mechanisms provides better application-specific *goodput* than other schemes. However, care must be taken about the exact functionality that is added inside the network; it is possible to add *enhancements* to the network that performs worse than simple Tail Drop. We further observe that if the adaptation in the network is via a generic scheme such as Tail Drop or Static Priority, then, under many circumstances, a significant fraction of the transmitted data is likely to be useless at the receiver.

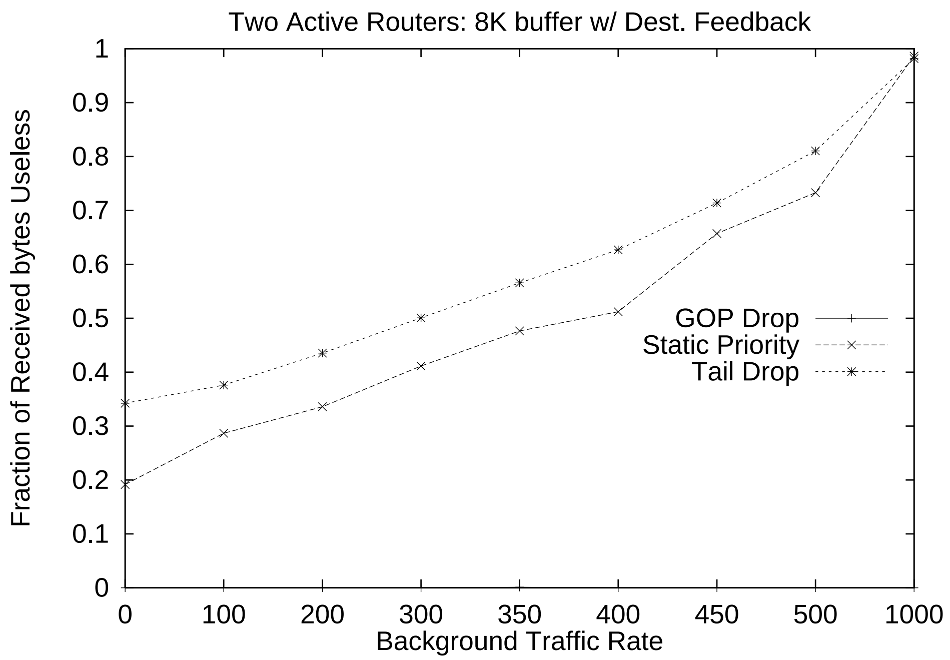

Figure 36: Fraction of received data useless; multiple active nodes with 8 K byte buffer each.

4.5 Network Support for Multicast Video

In this section, we extend our MPEG-specific algorithms to multicast best-effort settings. Three characteristics of multicast video distribution present particular challenges to system designers working in a best-effort environment. First, the paths from the sender to the receivers are likely to be heterogeneous in bandwidth availability. Second, the system will typically exhibit dynamic behavior over time. For example, the bandwidth on a single path may change during the lifetime of the connection and/or the set of receivers in the application may change. Finally, for applications with a large number of receivers, scaling becomes a concern.

As discussed earlier, the classic approach to dealing with dynamic network conditions in unicast applications is *sender adaptation* [65]. Multicast compounds the difficulty of sender adaptation [99]. If the sender adapts to the least capable receiver, then all receivers must suffer lowest-common-denominator performance. If the sender relies upon feedback information from receivers (e.g., concerning losses) then the system is prone to feedback implosion for large numbers of receivers. Despite these difficulties, systems have been developed that use sender adaptation for multicast video distribution (e.g., [21]).

The difficulty of sender adaptation for multicast has led more recently to the exploration of *receiver-based adaptation* [76, 105, 72, 99]. Under these schemes, the sender transmits multiple streams suitable for a range of path characteristics. Each receiver joins the stream (or streams) that can be supported by the individual path from the sender. This approach solves the problems of feedback implosion and lowest-common-denominator performance, however other problems persist. The difficulty in detecting an increase in bandwidth remains. To detect a bandwidth increase, the receiver must conduct a "join experiment," attempting to receive at a higher quality and monitoring for losses. Unless these experiments are coordinated, receivers will interfere with one another. To make a change in the received transmission, the receiver must join or leave a multicast group, incurring the delay required for routing changes to propagate. include this? Finally, to avoid complexity at the sender and in the multicast group maintenance, receiver-based schemes typically support coarse granularity in receiver capabilities. That is, the number of distinct bandwidth classes is often quite limited.

In what follows, we discuss the extension of our unicast congestion control schemes to operate on multicast flows (Section 4.5.1) and provide a summary of receiver-based adaptation schemes (Section 4.5.2). Our multicast experiments are simulation based. We explain the experimental environment in Section 4.5.3 and present our results in Section 4.5.4.

4.5.1 ASCC for Multicast

The basic operation of multicast ASCC is as follows: the sender transmits a full rate stream on a single multicast group, to which all receivers subscribe. As in the unicast case, we assume the routers in the multicast tree have information (established *a priori* by out-of-band mechanisms and/or carried in-band as part of data packets) about how to reduce the rate of the stream in the face of congestion on an outgoing link. For routers in the tree that do copying, the rate adaptation occurs *after* copying the multicast packet. Thus each outgoing link is treated separately and each receiver obtains performance based on the path from the sender, not influenced by other receivers or parts of the multicast tree. The adaptation occurs when dictated by congested conditions, thus an increase in available bandwidth can immediately be utilized and a decrease in available bandwidth immediately triggers controlled reduction of the rate. Note that as in the unicast case, in multicast ASCC, the network applies the adaptation. Thus, the time and place for adaptation can better match network conditions. Further, the adaptation can occur more rapidly, without the need for route changes. Finally, the adaptation can occur at finer granularity, providing better quality and more graceful degradation to receivers.

4.5.2 Receiver-based Adaptation

We compare our network-based adaptation scheme to a receiver-based adaptation scheme based on the main ideas in the Receiver-driven Layered Multicast approach [76]. Specifically, the sender transmits three streams —each on a different multicast group— corresponding to the I-frames, P-frames and B-frames of the MPEG video. The I-frames form the base layer; the P-frames form the first enhancement layer and the B-frames form the second enhancement layer. A receiver initially joins all three groups. As described in more detail below, a receiver uses periodic monitoring of the received streams to decide whether to add a layer (by joining a multicast group) or delete a layer (by leaving a multicast group). Each receiver acts independently.

The receivers join and leave multicast groups based on monitoring the quality of the received video stream. During each monitoring interval, the receiver measures two quantities: (1) the number of bytes received that could be decoded and (2) the number of bytes received. At the end of each interval, the receiver takes the ratio of the first quantity to the second. If the result is greater than or equal to a fixed threshold $hiWater$, then the lowest layer that is not currently being received is added. Conversely, if the result is less than a fixed threshold $loWater$, then the highest layer that is currently being received is deleted. However, our receivers never drop the lowest layer (the I-frames). In our experiments, we set the value of the monitoring interval to 6 GOPs, or every two seconds. We experiment with different values for the thresholds; most of our experiments use values of 0.85 and 0.90.

We bias our simulations against the network based adaptation by modeling all multicast group joins and leaves as instantaneous events. That is, when a receiver decides to join or leave a group, no time elapses for the updates to the corresponding routing tables.

4.5.3 Experimental Environment

Our results were obtained using a locally developed discrete-event simulator — ANSWER— that has specifically been developed for evaluating active networks. In this section, we detail the source models used for both video and data sources, the experimental topology, and the metrics we use to evaluate the performance of each adaptation scheme.

4.5.3.1 Source Models

The video sources in the simulations generate three distinct video layers, corresponding to layered MPEG video with 10 frames per GOP, at the rate of 30 frames per second. The I-, P-, B- layers generate 3, 9, 18 frames per second, respectively. The

GOP structure, and the frame sizes are shown in Table 7. The maximum size of a packet is 576 bytes, and larger frames are segmented over several packets. Note that the B-frames account for 60% of all frames, but only 13% of the bandwidth, while the I-frames only constitute 10% of the total frames, while consume more than half the bandwidth. Our analysis and the analysis of others [52, 44, 20] indicate that this is a reasonable model for frame sizes for MPEG.

For layered video, each layer is transmitted over a separate multicast group, and the destinations may choose to receive only a subset of all the layers. For the network-based adaptation, all the packets generated by the video sources are transmitted over a single multicast group. All video receivers have a re-assembly buffer of size 2 GOPs, which is used to reassemble the packets to reconstruct the original GOP structure[8].

Table 7: Details of Simulated MPEG Streams

Frame Type	Max. (bytes)	Mean (bytes)	Min. (bytes)	Gen. Dist.	Frames/ Second	Kbits/ Second
I	17K	16K	15K	Unif.	3	385
P	17K	2100	512	Exp.	9	150
B	17K	550	256	Exp.	18	80

The background traffic generators produce packets of size 576 bytes, with packet inter-arrival time obtained using a truncated exponential distribution. The mean of the exponential distribution is chosen such that the background traffic has the desired long-term average rate. For some of our experiments (reported in [15]), the background traffic sources use an exponential-backoff algorithm (multiplicative decrease, additive increase similar to TCP) under congestion.

4.5.3.2 Topology

[8]Packets are not re-ordered in course of our simulations.

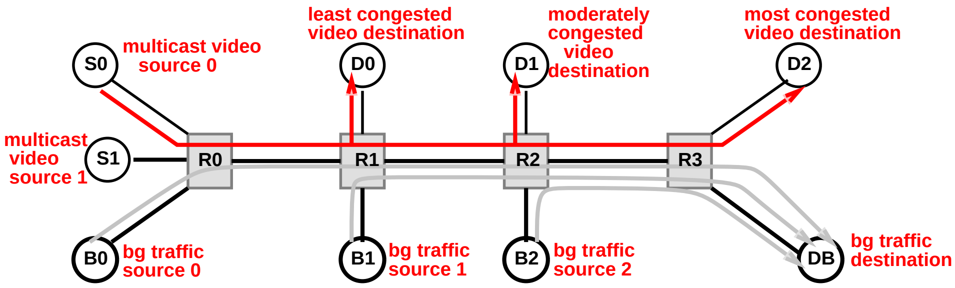

Figure 37: Experimental Topology

The experiments were simulated on the topology shown in Figure 37. There are four routers in the topology: nodes R0 through R3. Node S0 is the source of multicast video to destinations D0, D1, and D2. The multicast video is congested at various links between the routers by the background traffic sources (nodes B0, B1, B2). Note that the traffic from node B0 congests all the video destinations, while the traffic from node B1 only affects video traffic to nodes D1 and D2. Further, the background traffic from node B2 only affects the most congested video destination: node D2. All background traffic is destined to the background destination BD.

For all of the reported results (except details of particular runs), each reported data point is a mean of 10 separate runs of length 30 seconds each. The cyclic order of frames in MPEG account for the (relatively) quick convergence and tight confidence intervals shown in the results.

4.5.4 Results

The results that follow fall in two broad categories:

- Results that show performance at all three video destinations (D0, D1, D2). For each destination, the results show the effect of network-based adaptation and of receiver-based adaptation. For these results, the plots have three distinct "regions" corresponding to the performance of each destination, commensurate with the bandwidth on the path from the source to the receiver.

- To preserve legibility and space, for a subset of results, we only present the performance at the moderately congested receiver (D1). In these cases, the performance at the other receivers, in general, follow the same trends.

4.5.4.1 Single Video Source

In this series of experiments, we analyze the effect of network- versus receiver-based adaptation for a single video source. Unless otherwise stated, the background sources B0, B1, B2 (in Figure 37) have average rates of 350, 100 and 150 Kbps respectively, and the capacity of all links is fixed at 1000 Kbps. Before examining any results, we analyze the expected behavior at the receivers based on the long-term average rates of the traffic. The background traffic only accounts for 350 Kbps on the link between routers R0 and R1. Since the video rate is 615 Kbps, we note that destination D0 should be able to receive all the video frames. The background traffic from source B1 further congests the link between routers R1 and R2, and the residual capacity of 550 Kbps is only sufficient for destination D1 to receive the I- and the P-frames. Similarly, we expect destination D2 to receive only the I-frames, since the residual capacity on the link between routers R2 and R3 is only 400 Kbps. Of course, this is a simple analysis based upon average frame sizes and data rates at each video layer. Short-term increases in rate will cause router buffers to fill and receivers to experience performance that differs from these expectations. It is precisely these fluctuations in load that we expect the adaptation mechanisms to deal with.

Parameters for Receiver-based Adaptation

Before we present the results for the single video source experiment, we have to choose values for the different parameters for the receiver-based adaptation scheme. In Figures 38 and 39 we investigate the effect of changing the values of the high and low watermark parameters in the receiver-based scheme. Figure 38 shows the fraction of transmitted frames decoded at each receiver for three values of the high

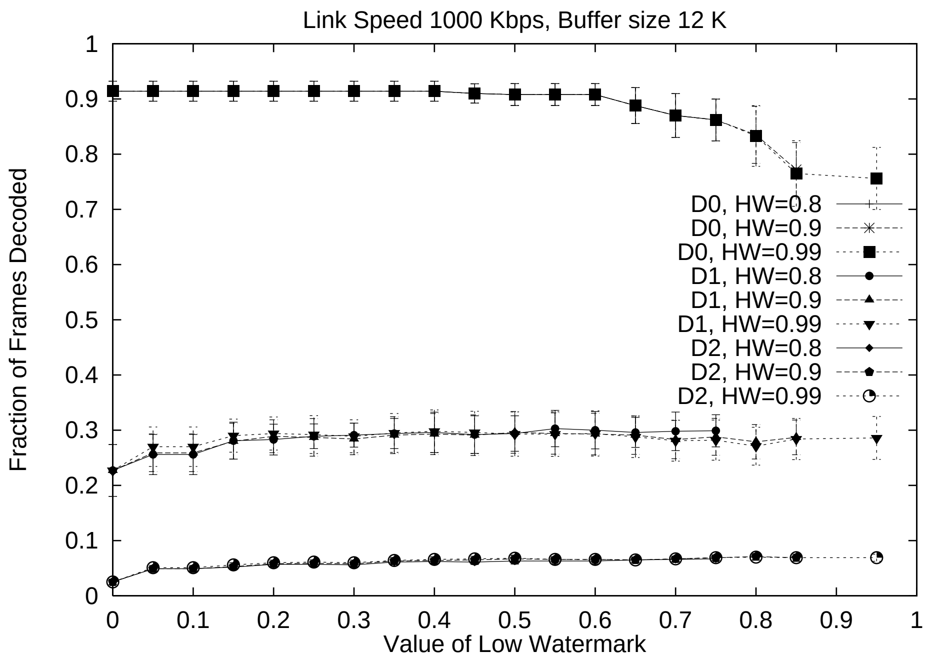

Figure 38: Single video source, varying high, low watermark

watermark (HW=0.8, HW=0.9, HW=0.99) as the value of the low watermark ranges from (0 ... HW). Note that when the low watermark value is 0, the receivers never drop a layer.

We see in Figure 38 that for all three receivers, the value of the high watermark does not make an appreciable difference with respect to the number of decodable frames received. However, as the low watermark increases, the uncongested receiver (D0) drops layers more readily and performs worse. The congested receivers (D1, D2), however, perform marginally better as the low watermark increases.

The reason for the variable sensitivity across receivers is shown in Figure 39, which gives the fraction of received data that is useful (i.e., is part of a decodable frame) for the same values of the high and low watermarks. In this case, neither the high nor the low watermark affects the useful data received at the uncongested receiver. However, for the congested receivers, the values of both the high and the low watermarks make a difference. More data is wasted for lower values of the high

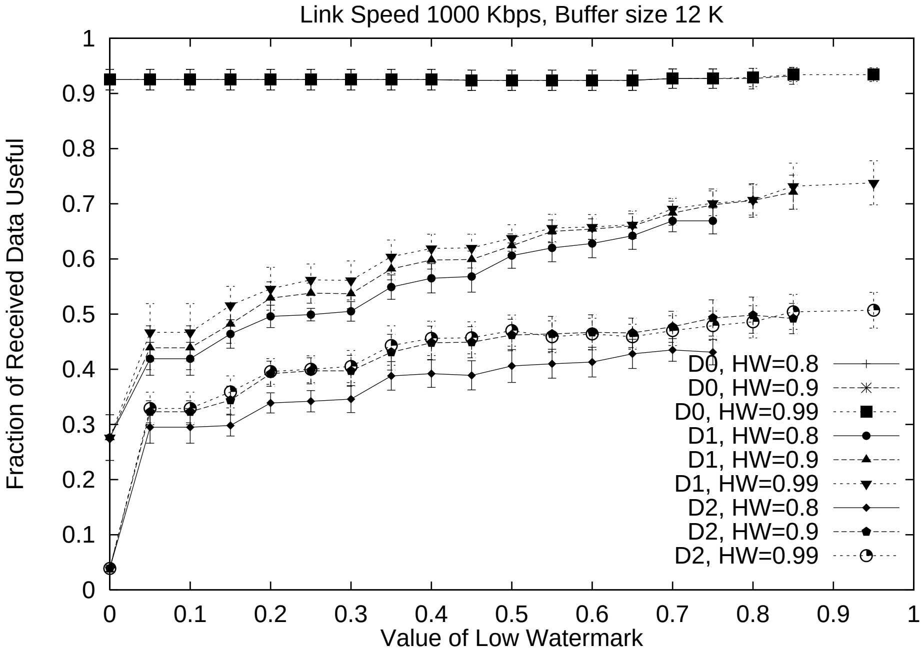

Figure 39: Single video source, varying high, low watermark

watermark, since the destination is more likely to add a layer even when the path may be congested. The fraction of received data that is useful steadily increases as the value of the low watermark is increased. As will be shown shown in detailed results (Figures 42 and 43), the effect of receiving data that cannot be used to decode frames is particularly severe for congested receivers.

In summary, a higher value for the high watermark tends to be preferred. However, the choice of low watermark is more complicated. Congested receivers prefer a higher value of the low watermark, while uncongested receivers prefer a lower value. For our experiments, we choose a high watermark of 0.90 and a low watermark of 0.85.

Performance of Network- and Receiver-based Adaptation

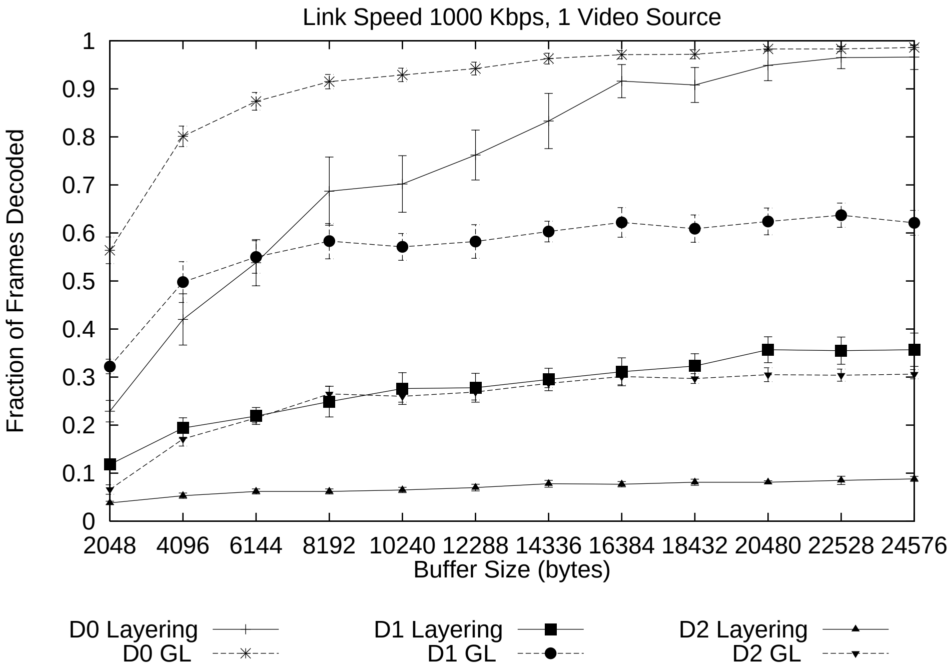

Figure 40: Single video source, varying buffer size, 1000 Kbps Link

Figure 40 shows the fraction of frames decoded at each receiver (D0, D1, D2) for network- and receiver-based adaptation when the link speed is fixed at 1000 Kbps. The network-based adaptation curves are labeled “GL” for GOP-level discard; the receiver-based adaptation curves are labeled “Layering”. Further, we also evaluated the case in which both the network adapts using the GOP-level discard scheme, and the receivers adapt by adding and deleting layers. This hybrid scheme (not shown) performs within 5% of GL in all cases.

For the congested receivers (D1 and D2), the network-based adaptation consistently delivers 2-3 *times* more decodable frames than layering. In fact, the performance of the most congested receiver (D2) in case of the network-based scheme is comparable to the performance of the moderately congested receiver (D1) under layering. We note that none of the schemes perform well with a very small amount of buffering at the output queues (less than 4KB). In case of the least congested receiver (D0), the network-based scheme has far better performance at smaller buffer sizes. This is because the receiver-based adaptation "over-reacts" to transient congestion and drops the B-layer, which accounts for 60% of all the frames. As the buffer sizes are increased, the network is able to absorb the transient bursts, and the performance of receiver D0 under receiver-based adaptation approaches 100% goodput. However, increasing buffers do not help in case of long-lived congestion as is seen in the case of the congested receivers (D1 and D2).

In other experiments (not shown), we increase the link speed to 1100 Kbps. This causes the long-term average load on the links to be about 90%. Under these conditions, receivers D0 and D1 both perform well using receiver-based adaptation, receiving more than 90% of all frames using a reasonable amount of buffering. However, the network-based scheme still outperforms the receiver-based scheme. For receiver D2, the network-based scheme is able to successfully transmit more than 4 *times* as many decodable frames for all buffer sizes. Similar trends were observed when the link speed was set to 900 Kbps (i.e. permanent overload in the network).

Further Variation in Background Traffic Rates

In Figure 41, we vary the rate at which the background source 0 (B0) generates traffic. By virtue of the location of node B0, a change in rate affects all the destinations. In this experiment, we simulate various underload and overload conditions. (Note that congestion *decreases* as one goes from left to right on the x-axis.) Along with the GL and Layering schemes, we also report the results for simple Tail Drop at the congested routers. That is, no adaptation occurs; if a packet encounters congestion, it is discarded.

Once again, the network-based scheme outperforms the receiver based scheme (Layering). However, an interesting effect is seen for the Layering and the Tail Drop schemes. For the moderately congested receiver (D1), the active mechanisms

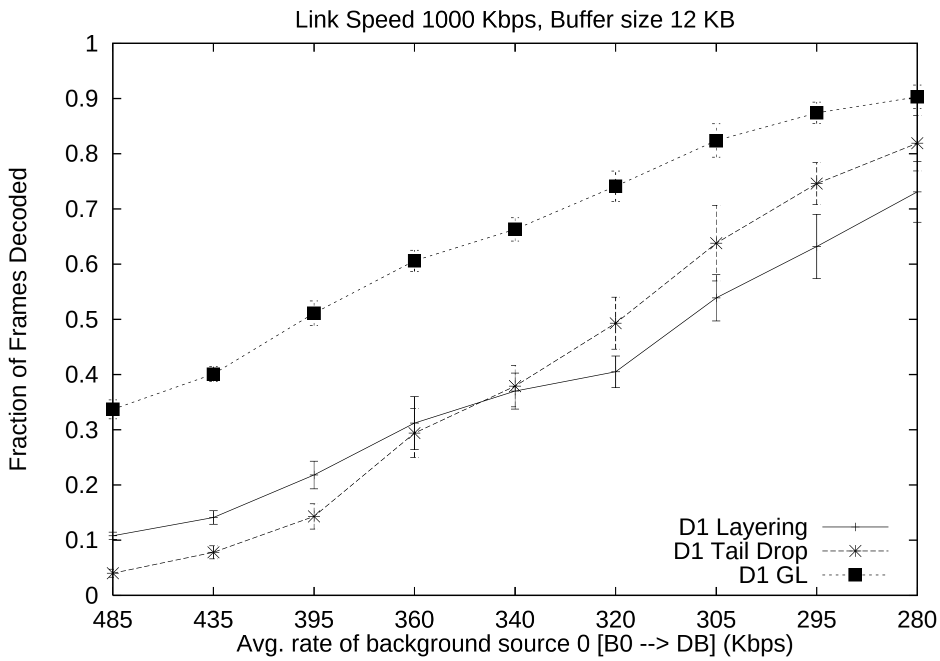

Figure 41: Fraction of frames decoded at destination D1 with varying background traffic from B0 $\rightarrow$ DB. Note that when the congestion is not severe, Tail Drop can outperform Receiver-based schemes.

outperform the receiver-based schemes, achieving up to three times better performance under severe congestion. Under less severe congestion, the Tail Drop scheme outperforms the Layering scheme. As congestion is increased, the difference in performance between Layering and Tail Drop decreases, and the two curves cross over at a background rate of about 340 Kbps (average link load 95%). As expected, under heavy congestion, the Tail Drop scheme performs worse than layering, and the difference increases as congestion worsens. The layering scheme is able to adapt to congestion by (correctly) discarding the appropriate layers while the Tail Drop scheme is faced with indiscriminate packet loss, and the ensuing degradation to the received stream. Analysis of the received streams shows that the Layering scheme often transmits a larger fraction of decodable bits, compared to Tail Drop. However, even while transmitting less useful data, the simple Tail Drop is able to outperform

Layering if the congestion is not severe.

Figure 41 does not show the results for the uncongested or the heavily congested receiver. In this experiment, the unloaded receiver (D0), Tail Drop *outperforms* the Layering scheme. This is because the layering scheme is prone to drop a layer in response to transient short-lived congestion, even though there is enough long-term capacity. For the heavily congested receiver (D2), the active outperform the receiver based schemes by at least a *factor of 3*. Further, for the reasons mentioned above, Layering always outperforms simple Tail Drop at this receiver.

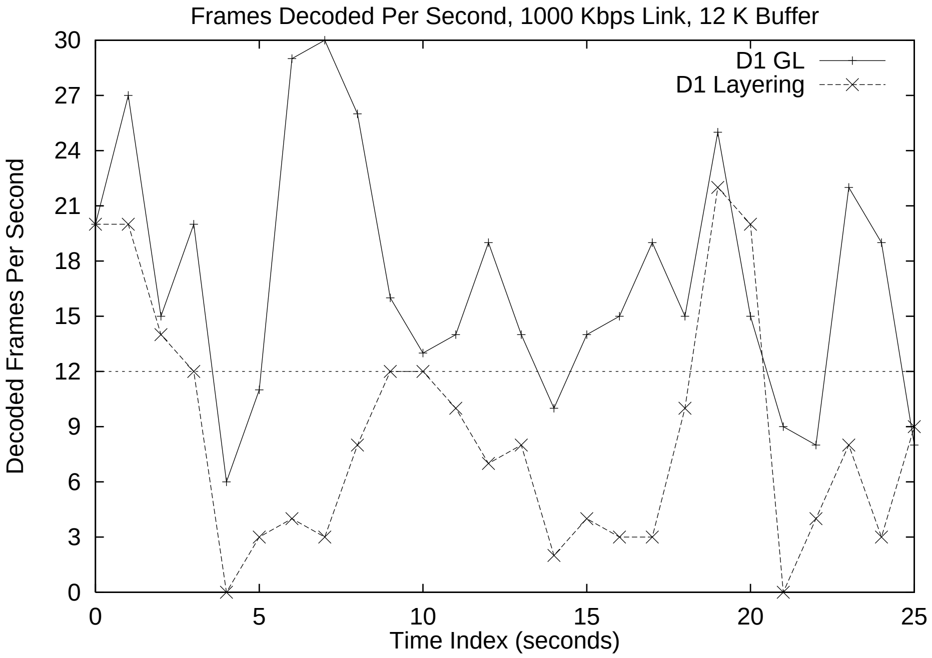

Figure 42: Single video source, decoded frames per second at destination D1

4.5.4.2 Detail Evaluation of Received Stream

In Figures 42, 43 and 44 we investigate, in detail, the received stream for one run of the one video source experiment with the buffer size fixed at 12 Kbytes and the link capacity fixed at 1000 Kbps. The background traffic was generated using the

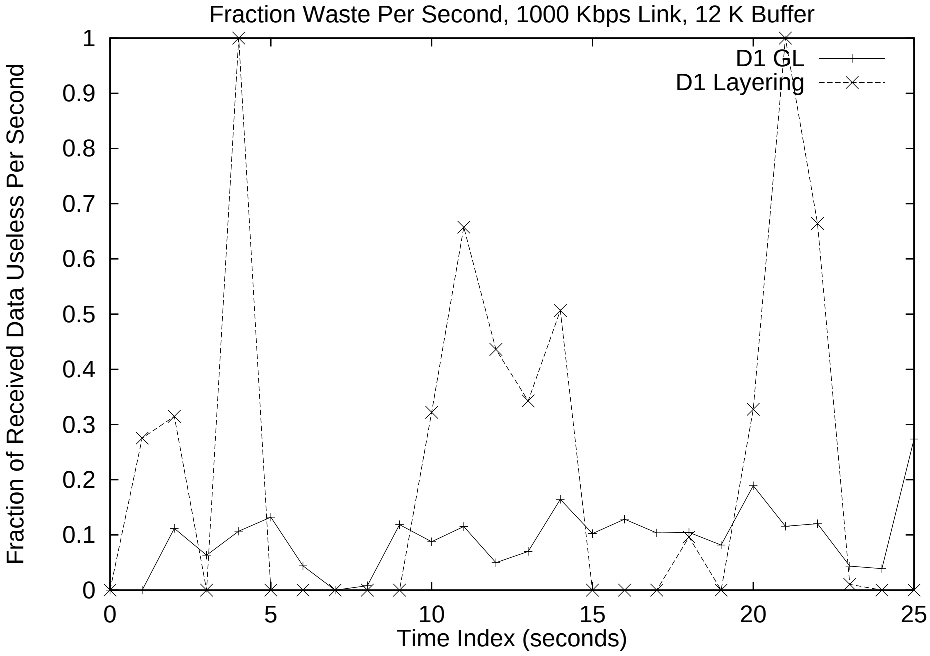

Figure 43: Single video source, fraction of data wasted per second at destination D1

values given in Section 4.5.4.1. In these plots, we only analyze the stream received at the moderately congested receiver (D1). These results are representative of the trends observed at other receivers.

Figure 42 shows the number of frames decoded at receiver D1 for the first 25 seconds of the simulation. Note that node D1 is congested by both sources B0 and B1 and hence, on average can only receive the I- and P-frames (i.e., 12 frames per second). The network-based scheme achieves over nine frames per second for about 96% of the interval, and over 12 frames per second for 75% of the interval. The minimum performance is 6 frames per second. Thus, on average the GL scheme transmits well above the nominal 12 frames per second. On the other hand, the Layering scheme spends one-third of the time below three frames per second, and about 70% of the interval strictly below 12 frames per second. Hence, the perceived quality is nearly always worse using the layering scheme.

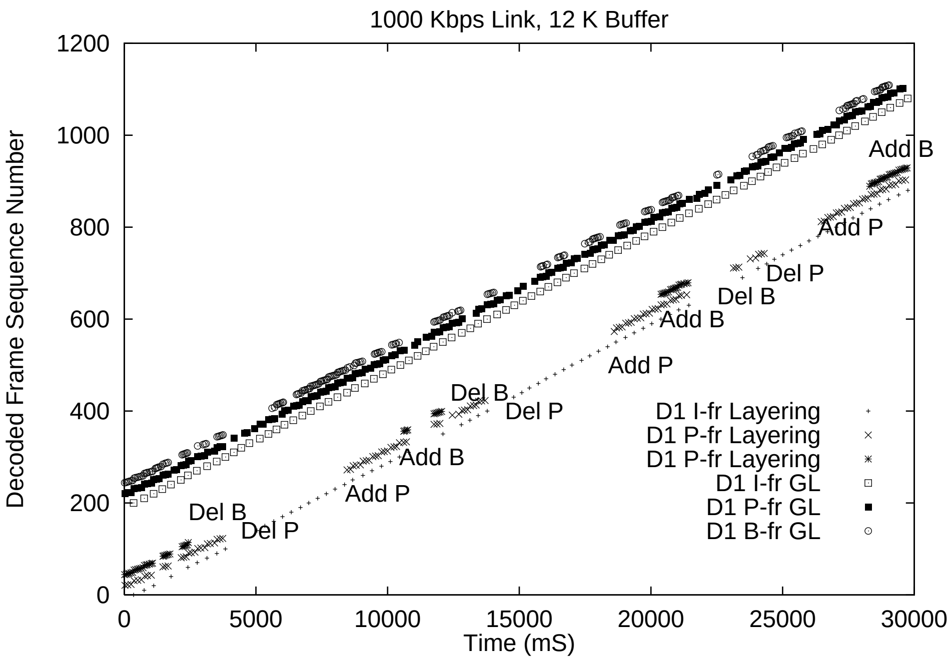

Figure 44: Decoded Frames over time at destination D1

Figure 43 shows the fraction of bits received that could not be used to decode a frame because other required data for decoding was missing. This plot provides some insight into the rather poor performance of the Layering scheme in the previous plot: when the decoded frames per second metric is low, it is because the wrong bits are being transmitted on the congested link. Lastly, we note that the GL scheme consistently transmits about 10% useless data. This is because our implementation does not buffer a frame before transmitting. In some cases, the beginning packets of a frame have already been transmitted when the middle/end of the frame encounters congestion and has to be discarded. In these cases, the receiver has to discard the received frame fragment since it is not decodable.

Figure 44 shows the actual frame sequence numbers and types that are decoded at destination D1, and is annotated with the layer additions and drops for the Layering scheme. To facilitate readability, the P- and B-frame sequence numbers are offset by 10 and 20 respectively, in the plot. Further, the entire GL stream sequence is offset by 200. This result clearly shows the reasons for the sub-par performance of Layering. The Layering scheme has to continuously probe the conditions in the network in order to get the maximum number of frames. In our case, as soon as the B-frame layer is added, the receiver is doomed to indiscriminate losses (as the capacity on the link between R1 and R2 exceeds 100%). This explains the periods when all data that gets through is useless, and the frames decoded per second goes to zero. On the other hand, the GL scheme is able to accommodate all the P-frames and is also able to send an appreciable number of B-frames. This is because the congestion rates are variable, and the majority of B-frames are very small (less than 1K byte). We also note that due to transient congestion, there are times when even the GL scheme is not able to transmit P-frames (and this leads to less than 12 frames decoded per second). These snapshot results show the value of placing application-specific adaptation precisely at the congestion point *inside*, as compared to the probe-and-adjust mechanisms of receiver adaptation.

4.5.5 Multiple Video Sources

In this section, we consider cases in which there are two video sources (at S0 and S1), each of which generate 615 Kbps MPEG video. The link speed was fixed at 1500 Kbps, and the average rates of background traffic from B0, B1, B2 were set to 250, 170, 320 Kbps. These rates are chosen such that on average, there is enough bandwidth on the link between R0 and R1 so that receiver D0 can receive both streams. Further, the extra 170 Kbps from source B1 congests the path to D1 such that, on average, D1 can only receive the I- and the P- frames from streams. Finally, there is only enough bandwidth on the path to D2 such that it can, on average, only receive the I-frames from both the sources.

4.5.5.1 Multiple Equal Weight Sources

Figure 45 shows the case when both the sources S0 and S1 are considered equal

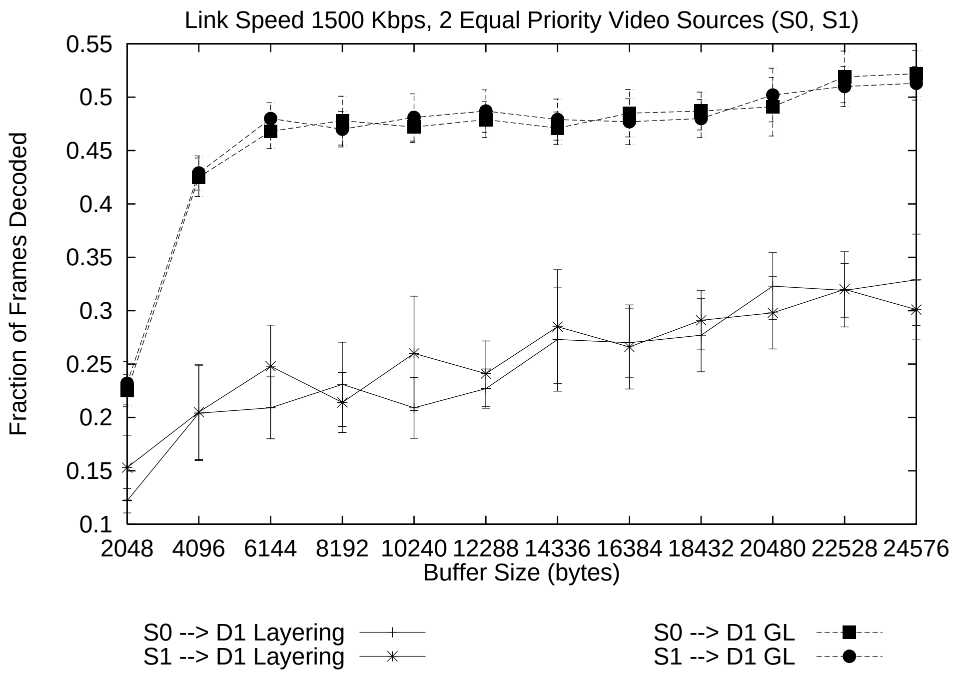

Figure 45: Two equal priority video sources, varying buffer size, 1500 Kbps link

priority at each receiver, i.e., each receiver tries to maximize the number of frames received from each source. In this case, both adaptation schemes treat the sources independently. In effect, each stream acts as congestion for the other.

Figure 45 shows the fraction of frames decoded for each source at each receiver for the network-based adaptation scheme (GL), and the receiver-based adaptation scheme (Layering). We see that the number of frames decoded for each source under the GL scheme is extremely close, and intuitively, compared to Figure 40, larger amounts of buffering is required to achieve the same fraction of decoded frames. As in the single source case, for congested destinations (D1 and D2), the network-based adaptation outperforms the receiver-adaptation schemes. Like the single source case, the gains for the congested receivers, are independent of buffer size. We also note that there is significantly more variability in the fraction of frames decoded (as shown by the larger confidence intervals) in case of layering. This is because of the effects of running multiple simultaneous layering algorithms at each receiver. Recent work on layering schemes have produced algorithms in which receivers "learn" from neighbor's join and leave experiments [72]. Though we did not implement these algorithms in our experiments, we expect such algorithms to reduce the observed variability in frames decoded for different sources.

4.5.5.2 Multiple Prioritized Sources

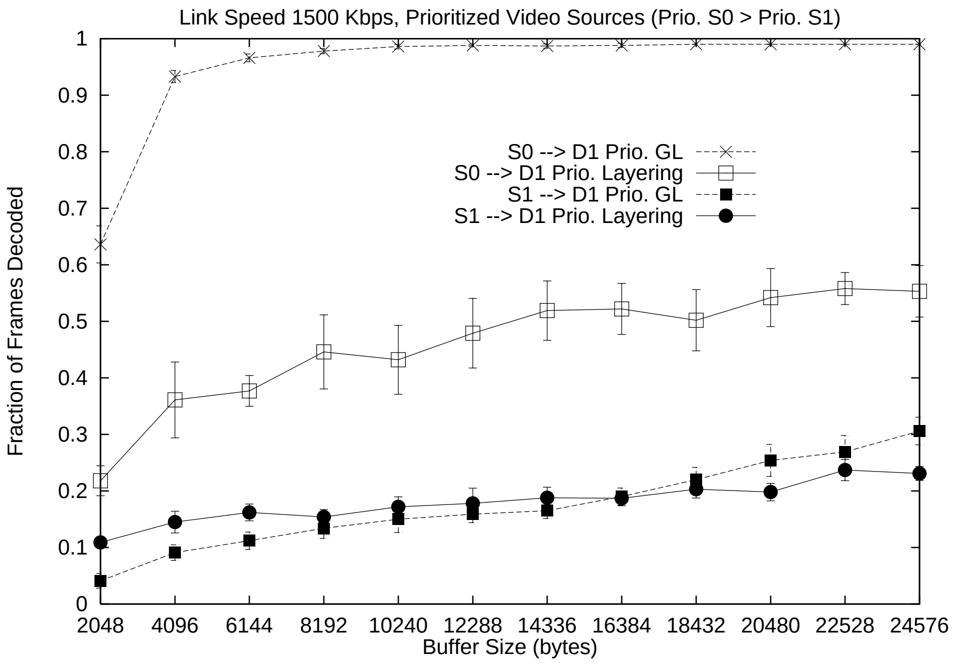

Figure 46: Fraction of frames decoded at receiver D1 with prioritized video sources(Priority of source S0 > S1). The active and the receiver-based schemes both receive the same fraction of low priority frames; however, the active scheme delivers substantially more decodable high priority frames.

In the following experiments, we consider the case of multiple sources with different priorities. In our case, source S0 is deemed more important than source S1. In what follows, we extend the GL and Layering schemes to handle multiple priorities as follows:

- **Priority GL.** A flow priority field is added to the application-level header for each video packet. If a packet from a high priority frame is congested, then the priority discard algorithm may evict packets of lower priority frames in order to accommodate the high priority packet. Further, for any packet discarded, all its associated packets (other packets of the frame and other frames that may depend on this packet) are also discarded. It is interesting to note that it is not clear how a prioritized discard algorithm can be implemented at a router without explicit access to packets already enqueued.

- **Priority Layering.** We extend the regular layering algorithm as follows: if a high priority flow receives a "poor" quality stream over the last 2 seconds (i.e. decodable fraction of bytes less than the low watermark), then if possible, layers of all *lower* priority streams are deleted before deleting any layer from the higher priority scheme. The issue of handling multiple priority streams at receivers is currently a topic of considerable research, and we acknowledge that there may be more intricate ways to extend the simple layering scheme than what we have implemented that provide better performance.

Figure 46 corresponds to the scenario in which both the high and low priority source runs for 30 seconds. In Figure 46, we report the fraction of frames decoded at destination D1 for each source. Note that the aggregate background traffic at node D1 is 420 Kbps (250 Kbps from B0 and 170 Kbps from B1). Thus, on average 1080 K bytes remain for the video traffic. If the priority mechanisms execute perfectly, the high priority source will receive approximately 615 Kbps and the residual 465 Kbps will remain for the low priority video flow. Recall that on average, I-frames account for 10% of the frames and 385 Kbps and the P-frames account for 30% of the frames at 150 Kbps. Thus, 465 Kbps is enough for the I-frames (385 Kbps) and about half of the P-frames (150 Kbps total). Thus, a perfect adaptation algorithm (without transient bursts of congestion and constant frame sizes) would deliver 100% of the high priority frames and about 25% of the low priority frames.

In Figure 46, we see that both the active- and receiver-based mechanisms deliver between 15–20% of the low priority frames. The fraction of low priority frames increases with increase in buffer size. However, with at least 8K bytes of buffering, more than 99% of the high priority frames are decoded using the active scheme. Regardless of buffer size, the receiver-based schemes are not able to transmit more than 50% of the high priority frames.

The uncongested receiver (D0) under receiver-based adaptation does receive up to 90% of the frames. However, 18 Kbytes of buffer is required before the receiver D0 can receive 90% of the high priority stream (using receiver-based adaptation) and about 70% of the lower priority frames. For the same amount of buffer space under active networking, receiver D0 receives over 92% of the *lower* priority stream along with 100% of the high priority stream.

Unfortunately, no amount of buffering is enough for the congested receivers under prioritized layering to receive more than 50% of the high priority stream. In fact, receiver D2 is never able to receive 10% of the high priority stream. This is because the join experiments of the lower priority stream continuously disrupt the high priority stream by overloading the links between the high priority sender and the receiver. The active scheme is able to transmit over 95% of the high priority frames with about 12 K bytes of buffering. For both the active and receiver-based schemes, less than 5% of the lower priority frames are received (regardless of buffer size).

4.5.5.3 Periodic High Priority Source

The situation is similar in the next experiment, in which the high priority source is periodic. In this experiment, the high priority source transmits for ten seconds each starting at 5 and 20 seconds. Figure 47 shows the fraction of frames decoded at receiver D1. For all three receivers, the results are nearly identical to the continuous high priority source — the active mechanisms achieve more than 90% goodput for the high priority source with about 12 Kbytes buffer at each router while Layering requires 18 Kbytes to get the same goodput for destination D0. Once again, destinations D1 and D2 are not able to decode more than 65%, 9% of the high priority

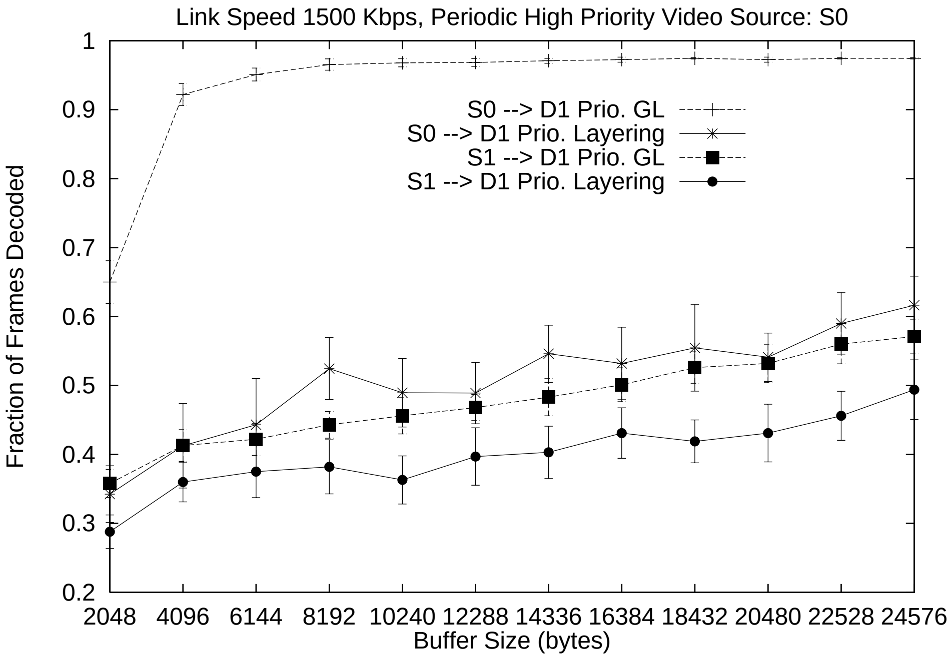

Figure 47: Periodic high priority source, varying buffer size, 1500 Kbps link

stream, respectively. For this particular experiment, the fraction of frames decoded metric can be deceiving — as the high priority source is not always transmitting. Detailed analysis of per-frame behavior is more interesting.

Figure 48 shows the decoded I-frames at destination D2 for one particular run for each adaptation scheme. Once again, the plots have been offset to enable readability. The problems with layering are evident as soon as the high priority stream begins transmission (5th second). As all the layers for the low priority stream was being received, there is a period of uncontrolled loss when nothing is decoded for either stream. Then the adaptation takes effect and the lower priority layers are dropped in quick succession. As soon as the lower priority stream performs a join experiment (approximately 12 seconds), the link is overloaded, and the receiver encounters uncontrolled congestion. The same pattern repeats the next time the high priority source transmits. The network based adaptation is simple and nearly perfect. As the high priority stream starts, the low priority stream is "thinned" and all

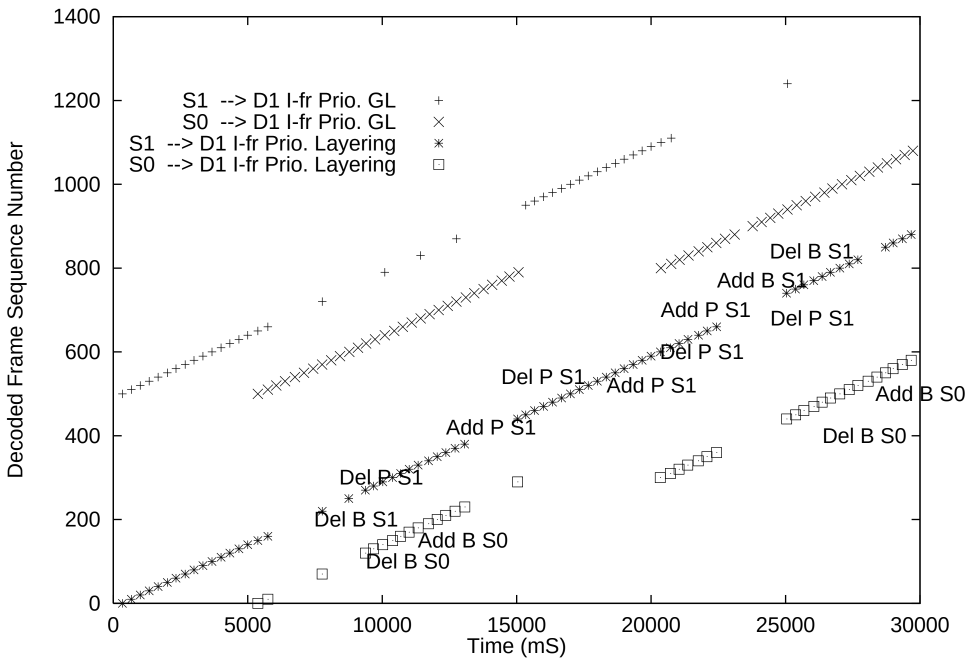

Figure 48: Periodic high priority source, Received Frame Sequence

the frames of the high priority stream are received. There is a period of transient congestion around the 23rd second in which even the high priority source loses an I-frame (and the associated GOP).

4.6 Discussion

We have presented a set of network-based techniques for reducing the effect of congestion in a best-effort unicast and multicast video distribution. Our results show that over a wide range of operating points, these techniques are superior overall to the layered receiver-driven approach, in terms of both end-user quality (as measured by fraction of total MPEG frames decodable), and network efficiency (as measured by the fraction of bytes delivered that were useless). However, these packet discard and queue manipulation strategies require supporting mechanisms in the nodes of the network (at least in those nodes where packets may be dropped because of congestion), while the end-system-based techniques do not. For this reason we consider some practical concerns related to these mechanisms, and argue that they are not insurmountable.

Concerns about these kinds of mechanisms fall generally into the categories of scalability, complexity, and, more generally, network "citizenship". The *scalability* issue is simply that these techniques require per-flow state. There is currently no consensus as to whether any kind of per-flow mechanism can be practically realized in the backbone of the network. One response is that the per-flow state required for these techniques is on the order of a few bytes per flow per outgoing queue. This is approximately the same amount of state as is required for multicast *routing*, according to present schemes.[9] Thus if multicast routing can be supported, the additional cost to implement the state required for these techniques should not be unreasonable.

The issue of *complexity* focuses on the fact that the operations required for these techniques —e.g., examining the contents of the queue in order to evict packets matching a particular pattern— are somewhat complex, and must be done at line speeds. In the following section, we present results (in the multicast setting) of active schemes that are more "light-weight" in their complexity.

4.6.1 Alternate Models of Network-based Adaptation

The GL adaptation scheme in Section 4.5 incorporated all the three mechanisms for discard discussed in Section 4.3.2. In Figure 49, we set the link speed to 1000 Kbps, and present results for the single source scenario when one or more of these mechanisms are not implemented at the routers. (This result is representative of the performance noticed at the other receivers). To facilitate comparison, we have included the GL result (which incorporates all three mechanisms) from Figure 40 in

[9]State required for multicast routing is $O(n)$, where n is the number of flows.

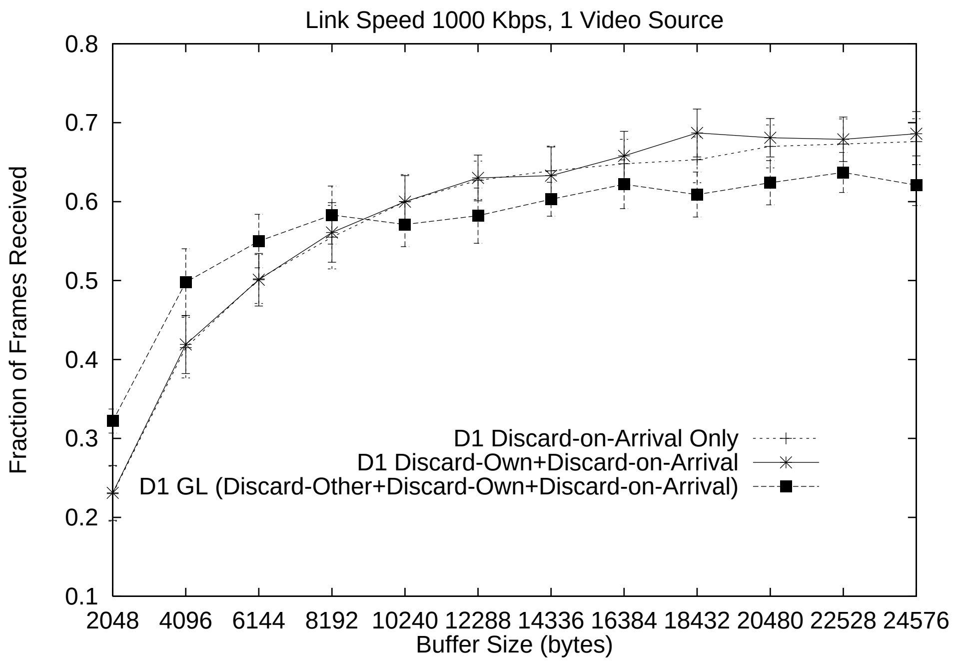

Figure 49: Single video source, varying buffer size, 1000 Kbps Link, Alternate Network-based adaptation schemes

this figure as well. Figure 49 contains results for two additional schemes: "Discard-on-Arrival Only" and "Discard-Own + Discard-On-Arrival". Recall that using the Discard-Own mechanism, when a packet encounters congestion, it along with all the packets for *its* frame is discarded. No packets other are discarded from the queue in order to create space for this frame. Under the "Discard-on-Arrival Only" scheme packets that encounter congestion are discarded, the output queue is not manipulated at all, and state is kept such that subsequent packets that depend on the current (discarded) packet are discarded before entry into the queue.

Like in the unicast case (Section 4.4.2.4), these two (simpler) mechanisms perform extremely well compared to GL (for all receivers), and for some buffer sizes are able to *outperform* GL. Our detail analysis (not shown) of the individual streams show that these schemes are able to transmit more "total" decodable frames as they transmit more B-frames. However the number of I-frames and P-frames sent using the GL scheme is more than either the Discard-Own+Discard-On-Arrival or the Discard-On-Arrival scheme (as the GL scheme selectively deletes B-frames to transmit congested I- and P-frames. As noted before, if an I-frame cannot be decoded, the receiver is not able to decode any other frame for 1 GOP time period (1/3 second). Thus, the perceived quality of the video at the receiver shows less variation using the GL scheme as compared to the other schemes. Note that even the "Discard-on-Arrival Only" scheme, which does not use any potentially costly queue manipulation mechanism, clearly outperforms the layering schemes (compare to Figure 40). Thus, this result makes a strong case for placing even a very small amount of application-specific state at network nodes.

4.6.2 Citizenship

Perhaps the most important concern is about the *citizenship* of these techniques—that is, their effect on other traffic in a best-effort network. In particular, TCP backs off under congestion, so that all flows can receive their fair share of bandwidth through the bottleneck. It is well-known that if a TCP connection shares a bottleneck with another flow that does not adapt, or does so according to a different algorithm, the non-TCP flows may end up with an unfairly large share of the available bandwidth.

The techniques investigated here —both network-based and end-system-based—react to congestion and adapt to available bandwidth, but differently than TCP. Our simulations confirm that a multicast flow of this kind would end up with an "unfair" share of bandwidth in competition with TCP. However, this is a problem for the receiver-driven techniques as well—as it is, potentially, for *any* protocol that adapts differently than TCP.

The best approach to ensuring "fair" sharing of bandwidth among such different kinds of flows seems to be the use of weighted fair queuing [11] or similar techniques. Consider reserving half the bandwidth of an outgoing link for TCP and half for video or other flows that adapt using different mechanisms. This ensures that bandwidth is shared fairly between *type* of flows, while the individual mechanisms ensure that it is shared efficiently (and fairly) *within*. Moreover, the scheduling mechanism need only scale to handle a small number of classes, determined by the number of different protocols in use.

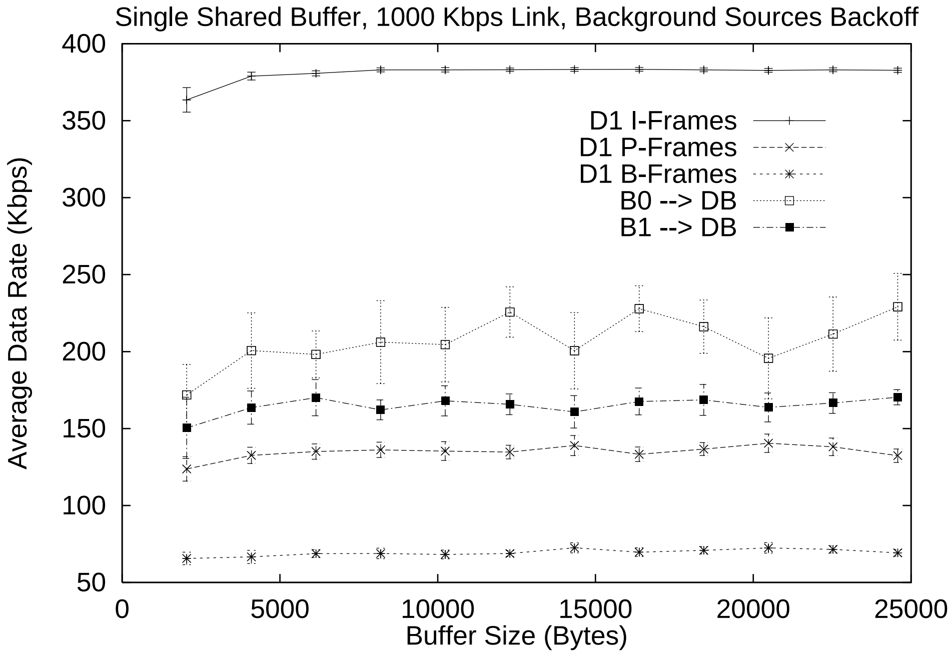

Figure 50: Single video source, varying buffer size, 1000 Kbps Link, Single shared buffer at routers with backoff-sources

Figures 50 illustrates the concern of citizenship when the background sources back off upon congestion. We use the same topology and source rates as the single source experiment in Section 4.5.4.1 and report the average data rate available to each sender under the network based adaptation scheme. In case of the video source, we present the individual rates for the I-, P- and B-frames (as perceived by receiver D1). The data rates for B0 and B1 are the rates observed at the destination BD. We note that the data streams back off and the video stream captures more than

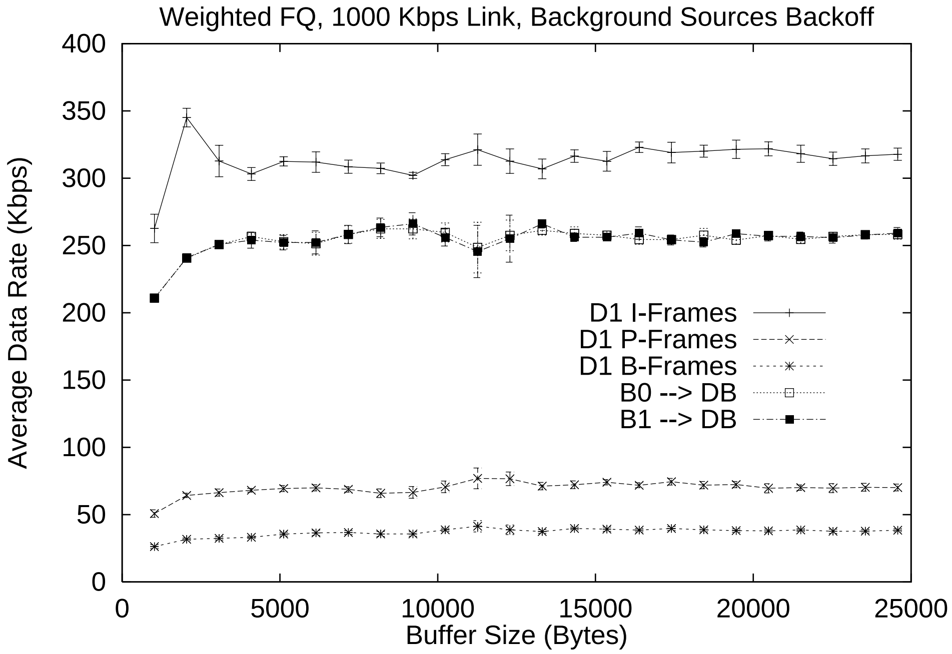

Figure 51: Single video source, varying buffer size, 1000 Kbps Link, Weighted Fair Queuing at routers with backoff-sources

its share of the bandwidth (greater than 500 Kbps). Further, there is significant variance in the data rates obtained by the data streams. Figure 51 shows the results when the routers perform weighted fair queuing, and each "flow" is weighted equally (all the video layers form one flow and each background source forms one flow). In this case, the data flows get their requisite 250 Kbps each, and the video flow to D1 gets the remaining 500 Kbps. We also ran the previous experiment in which all the background sources formed a single flow (and thus received 500 Kbps total on the link between R2 and R3) and in that case the background sources obtained an average of approximately 170 Kbps each (as there were three contending data sources on the R2 – R3 link).

4.6.3 Alternate Definitions of Decodable

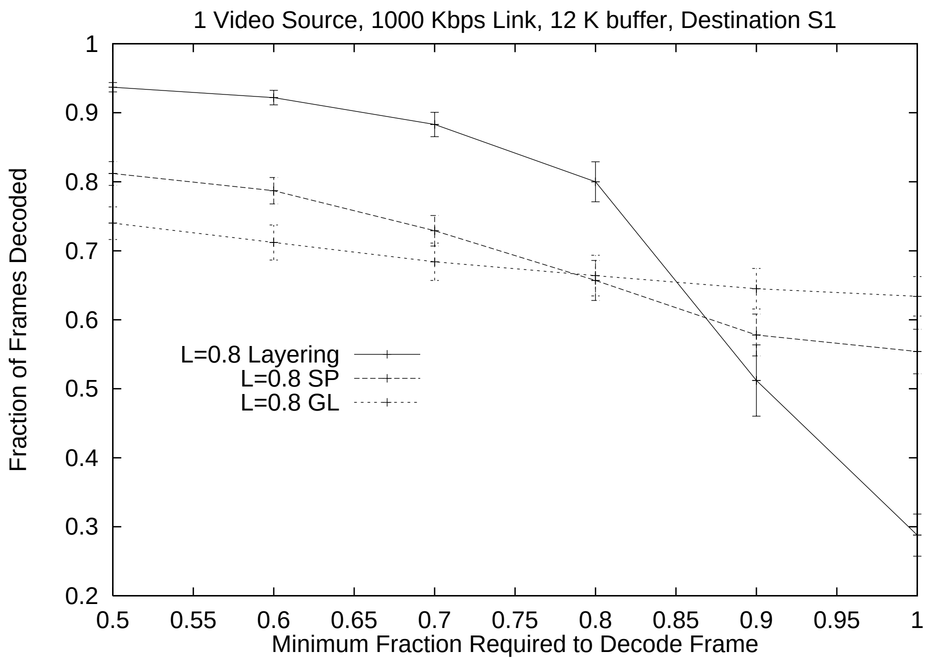

Figure 52: Decoder efficiency

Our experiments so far have assumed that all packets in a frame and all of its dependencies must be received in order for the frame to be decodable. We generalize the notion of a frame being decodable as follows: Given that all the layers a particular frame depends on have been decoded, a frame is decodable if and only if a fixed fraction F (or more) of the bits in the frame are received. Further, a layer is decodable if and only if a fixed fraction L of the frames within the layer could be decoded. Consider the following example: Let $F = 0.8$, and $L = 0.5$. Then an I-frame would be decodable if $\geq 80\%$ of the bits in the frame were received. As P-frames only depend on I-frames, a P-frame would be decodable if and only if more than 80% of it were received, given that more than 80% of the I-frame had been received. The B-frames depend on both P-frames and I-frames. Thus, for a B-frame to be decodable, more than 80% of it must be received, along with at least half of all the P- and I-frames it depends on. Note that in our experiments, our notion of decodable corresponds to L and F both equal to 1.0. If, due to advanced framing and coding, each packet could be decoded on its own, then the value of L would be 0.0. FEC techniques could be applied in this context to provide such sophisticated coding.

In Figure 52, we show the result of an experiment in which the value of F and L are not 1.0. For this experiment, we fix the value of L to be 0.6, and vary the value of F along the x-axis. The results show the fraction of frames decoded at the moderately congested receiver under the single sources scenario. Recall that this link speed corresponds roughly to an average offered load of about 95% on the links between the routers. The result also includes a new form of network adaptation: the routers employ a predefined static priority to discard packets under congestion (SP). In our case, packets carrying I-frames have strictly higher priority than packets carrying P-frames. Further, the packets carrying B-frames have the lowest priority of all. Thus, if an "I-packet" encounters congestion, all B- and P- packets may be discarded to ensure the I-packet can be accommodated. However, this scheme is not cognizant of frame boundaries etc., and may transmit fractions of frames.

The result in Figure 52 shows that if the decoder is sophisticated, ($F < 0.85$), then for $L = 0.8$), Layering outperforms the network based schemes. Further, there is a region where the static priority scheme outperforms the GL mechanism. This is because the network based mechanisms are engineered to transmit more I-frames, and in case of GL, more complete frames. Thus, the GL mechanism does not degrade appreciably when the amount of the frame required in order for it to be decodable reaches 100%. In case fractions of frames are considered useful, Layering (and even simple tail-drop) is the best scheme as every byte that reaches the destination is deemed useful. Thus, we see that an extremely "smart" end-system can compensate for the lack of sophisticated mechanisms within the network.

4.7 Related Work

Work related to application-specific congestion control falls two different categories: congestion-control mechanisms with network support and applications of active networks. ASCC for MPEG is related to other work in transport and transcoding of video — both in unicast and multicast settings. We discuss each category in turn.

4.7.1 Congestion Control Mechanisms

Existing congestion control mechanisms contain examples of mechanisms that have a similar spirit to our application-specific advice. Recognizing the performance degradation caused by fragmentation [68], packet-level discard techniques have been explored to improve the performance of TCP/IP over ATM [7, 90]. The Partial Packet Discard strategy drops all cells in a packet that follow a dropped cell [7]; the Early Packet Discard strategy improves performance by aggressively dropping entire packets when a congestion threshold is reached [90].

The Available Bit Rate (ABR) service defined for ATM [22] is also intended to relieve the two problems with sender adaptation mentioned in the Section 4.1.1. ABR reduces the time required for the sender to detect congestion by using resource management (RM) cells to convey congestion information. With RM cells, the time to notify the sender of congestion is reduced to the path delay from the point of congestion to the sender[10]. ABR addresses the problem of detecting an increase in available bandwidth by providing explicit rate feedback information. Switches can specify the allowed sender rate, based on calculations of current load and fair-share per sender. Thus ABR attempts to improve the sender-adaptation process, while maintaining the adaptation at the sender.

As we have noted in Section 4.1.2, efforts are underway to augment Internet routers to support network-level explicit notification of congestion [88].

[10]Note, however, that this delay can be considerable, if the point of congestion is far removed from the sender, or the path back to the sender is itself congested.

4.7.2 Applications for Active Networks

Under the topic of active networking, various research groups are considering ways that state and/or computation in the network might benefit applications. Selected work in this area includes support for reliable multicast [70], protocol deployment [3], and our own work in wide-area caching (Chapter 5). Other efforts in active networking bear on the feasibility of proposing that network routers perform computation on behalf of user flows.

The existence of an open and programmable network infrastructure is not necessary for the deployment of network-based mechanisms. If sufficient value is added by a mechanism, router vendors will have incentive to include it. Examples of recent mechanisms proposed include packet filtering (to protect end systems from attempts to exploit security holes) and TCP "ack spoofing" (to improve reliability over lossy links) [10].

4.7.3 Transfer and Transcoding of Continuous Media

In the area of multicast video distribution, the receiver-based adaptation scheme in Section 4.5.2 is based on the main ideas in the Receiver-driven Layered Multicast approach [76]. However, we do not include the more sophisticated approaches to control the frequency of join experiments and to share information about join experiments amongst receivers as described by McCanne et al. [76] or Li et al. [72]. These techniques may allow some improvement in the performance of the receiver-based scheme, however the fundamental challenge remains: receivers have limited information about the presence and absence of congestion in the network. Mechanisms placed in the network avoid join experiments and effectively coordinate receivers.

The Thin Streams approach [105] attempts to mitigate the problems that occur when the streams available to receivers are coarse-grained. The "thick" streams of video layering are divided into "thin" streams, each sent on a separate multicast group. The effect is that a join or leave causes less oscillation in network behavior, since the bandwidth of a stream is reduced. This approach, however, requires more state and overhead in the network to management multicast groups and more computation at the receivers to merge streams. The thin streams encoding could also be used by a network-based adaptation scheme, with natural extensions to the work we describe.

While dynamic congestion control is not an explicit goal of Amir et al.'s application level video gateway [4], their techniques for transcoding can have the effect of bandwidth reduction. Their focus is on accommodating an environment with heterogeneous transmission and end station capabilities, by converting video from one representation format to another. An implementation of a JPEG [62] to H.261 [63] transcoder can reduce the bandwidth of a video stream from 6 Mbps to 128 Kbps. Amir et al. further consider some temporal and spatial bandwidth reduction techniques to use in concert with format conversion.

4.8 Summary and Extensions

In this chapter, we have described an approach that allows applications to tailor the network's treatment of their packets when they encounter congestion. Our approach combines application-level knowledge (previously available only at the ends) with instantaneous congestion information (available only inside the network) to produce optimal network behavior upon congestion. By developing and evaluating specific algorithms, we have shown how this approach applies to a particular application, namely best-effort MPEG video transfer. We have implemented our mechanisms for best-effort unicast networks, and have performed simulations for multicast settings. We have shown how, in each case, our mechanisms compare against and can be co-employed with receiver-based feedback schemes.

In the course of our experiments, for both multicast and unicast delivery of data, we observed that following general trends:

- Maintaining per-flow state in order to discard selected packets on input provides advantages that are comparable to mechanisms that require the ability to manipulate output queues (in order to discard packets that are already queued). Effort expended at a router to maintain per-flow state for packet discard is equivalent (at least in order) to the effort expended for a multicast connection. Thus, deployment of ASCC for particular (popular) applications may be feasible. Note that as larger buffers and applications with more intricate dependency relationships are considered, the output queue manipulation mechanisms are likely to provide more benefit.

  Per-flow state at routers, especially for unicast connections, still poses problems of scale. The number of flows that traverse a router can be independent of the amount of data that flows through the router, and is dependent on the network topology and application characteristics. Thus, it is difficult to provision for the amount of state that may be required at a particular router for even ASCC input classification.

  We have developed an algorithm for processing *similar* MPEG streams in aggregate [16]. This work is not reported here; however, we provide a synopsis of the results and main contributions. With the per-application queuing algorithm, per-flow state does not have to be maintained at routers — instead, we maintain per-application state and buffering. In our experiments, MPEG streams with the same frame rate and similar frame sizes constitute an application (more details are available in [16]. The major contribution of this work is that our per-application algorithm, which is independent of the number of individual flows that are currently being served in an application class, provides the benefits of keeping per-flow state and buffers with no per-flow state and far smaller buffers. The per-application algorithm uses smaller aggregate buffering than per-flow algorithms due to gains from statistical multiplexing of buffer space; it also provides tighter delay bounds than traditional shared buffer schemes (such as Tail Drop). As the number of prevalent applications is relatively small, provisioning of resources on a per-application basis is likely to be easier than on a per-flow basis.

- Feedback based on too little data or too short a feedback interval can be harmful, as it leads to unnecessary reductions. We have reported results for varying the feedback interval for unicast data in [19]. As the multicast results in Section 4.5.4 show, over-reacting to short lived congestion bursts can adversely affect performance. This observation also clearly points out the fundamental problem with receiver-based schemes. The receiver is not able to discriminate between a short burst of packets from competing traffic versus changes in path capacity. If the receiver does not react at the onset of long-lived congestion, then it is doomed to a period of uncontrolled loss during which time it receives no useful data.

We have presented ASCC as an application for active networking, i.e. placement of general, user-controllable computing capabilities in network nodes. As such, it assumes architectural capabilities that are currently impractical in the current Internet. However, as the cost of computing continues to decrease, more and more intelligence is being placed inside the network in an *ad-hoc* manner. As we have noted before and as our implementations prove, ASCC can be applied within the constraints of the current IP protocol architecture. As such, ASCC is a viable extension for non-active networks as well.

Chapter 5

Self-Organizing Wide-Area Network Caches

5.1 Introduction

A substantial fraction of all network traffic today comes from applications in which clients retrieve objects from servers (e.g. the World-Wide Web). The *caching* of objects in locations "close" to clients is an important technique for reducing both network traffic and response time for such applications. Studies have shown [26, 38] that caching can substantially improve performance.

In this chapter, we consider the benefits of associating caches with switching nodes *throughout* the network, rather than in a few hand-chosen locations. We also consider the use of various *self-organizing* or *active* cache management strategies, in which nodes make globally consistent decisions about whether or not to cache an item in order to reduce overall latency. The per-packet processing required by the self-organizing caching algorithms is a natural application for active networks. Throughout this chapter, we use both terms ("self-organizing" and "active") to refer to cache management and organization techniques that make use of active network nodes. The techniques, however, are independent of any particular active network platform.

Our studies involve both analysis and simulation, and use object request distributions that are consistent with studies of actual Web traffic [6]. Our results indicate that in-network caching can make effective use of cache space. In particular, active caching yields round-trip latencies that are about as good, and in certain cases better than more traditional approaches, while requiring much smaller caches per node.

The remainder of this chapter is organized as follows. In Section 5.2 we provide background on wide-area caching and outline our assumptions about the application and network. We also describe related prior work in caching. In Section 5.3 we describe caching policies that take advantage of the ability to store and process information as packets pass through the nodes of the active network along with the more traditional methods to which we compare them. In Section 5.4, we describe our simulation study, including the parameters of our model, our simulation methodology and results. In Section 5.6, we develop an analytic model for network caching, and use it to validate the simulation model. We present conclusions in Section 5.7.

5.2 Background

5.2.1 Wide-area caching

We assume an application in which clients request objects from servers located throughout the network. Each object is assumed to have a globally unique identifier (e.g. a hash function of a URL), and to fit in a single "message". That is, each transaction consists of a request message sent from a client toward a server, containing the ID of one requested object. The request message travels through the network until it reaches a node where the requested object is stored, which may be a cache or the server itself. A response message containing the object then travels from the server (or cache) back to the originating client. This model ignores the significant problem of reliable transfer of large objects in a datagram network, by assuming that each object fits in a single message.

In our caching study, we model the network as a collection of *domains*, each of which is represented as a graph of switching nodes connected by links. Domains are of two types, *transit*, which (as their name implies) carry transit traffic, and *stub*, through which only packets addressed to or from some node in the domain are carried. The graph models used in our simulations are constructed using the GT-ITM internet topology modeling package [28]. The paths along which packets travel in the simulations have the following characteristics:

- The path connecting two nodes in the same domain stays entirely within that domain.
- The shortest path connecting node u in stub domain U to node v in another stub domain V goes from U through one or more transit domains to V, and does not pass through any other stub domains.
- In case two stub domains are connected directly via a stub-stub edge, the path between two nodes on the two domains *may* (but need not) go along that edge and avoid any transit domains.

5.2.2 Related work

The Harvest Cache [23] project at the University of Colorado is the largest wide area cache implementation in the Internet. Harvest caches are usually arranged in a hierarchy, and Web clients are manually configured to access a particular cache in the hierarchy that is designated as the client's proxy. If the client request can be satisfied at the initial proxy cache, it is served by the proxy. In case of a miss, the parent and sibling caches in the hierarchy are contacted using the connection-less mode of the Internet Cache Protocol (ICP [102] discussed below). If the requested object is not available at the sibling caches, or at the parent, then the client's proxy cache generates another HTTP query with its parent cache as the target using the HTTP proxy protocol or a connection-oriented version of ICP. This process is recursively repeated till the request is either served, or the object is retrieved from its origin (by the root of the hierarchy). Once the item is retrieved, it is cached on its way down to the leaf. This scheme is effective in reducing the wide-area bandwidth requirements, and in accessing "hot-spots" only once per hierarchy (as the "hot" item is then cached within the hierarchy, and subsequent requests to the same object can be satisfied by a Harvest cache hit).

The geographical push caching scheme [53] of Seltzer and Gwertzman at Harvard uses a friends-of-friends algorithm in which servers selectively push their content to *friend* caches that reside in active client networks. An similar push caching approach in which servers disseminate popular pages has been proposed by Oritz and German at the University of Waterloo [75]. In both these schemes, the server initiates the caching of an object (that it deems popular) at a remote site. In the simulation results given in the Harvard work, the "friendly" caches that act as proxys for a given server are chosen to be the nodes closest to networks that generate the most requests. Further, the server sends a redirect message to clients that contact the server but have a push cache close to them. In the Waterloo work, clients identify themselves to servers as willing push caches, and servers can choose to replicate popular objects at certain clients. The servers keep track of currently active push clients, and can redirect subsequent requests to its known proxies.

The Internet Cache Protocol (ICP) [102] defined by the Network Working Group of the IETF is a message format used for communicating among Web caches. Harvest (and its successor—Squid[1] use ICP to exchange information about objects cached at neighboring caches.

5.3 Network Caching Schemes

Traditional approaches to network caching place large caches at specific points in the network. In contrast, we consider networks in which relatively small caches are

[1]See `http://squid.nlanr.net/Squid/`

placed at every node. As a response message moves through the network, each node decides whether or not to store an object. Effective use of a large number of small caches is a non-trivial problem: unless they are effectively organized, only a small number of unique items will be cached (as the caches are much smaller), but these cached items will be replicated at many locations throughout the network. Thus, accesses to the few cached objects will exhibit low latencies, but overall average latency will not decrease appreciably. However, if objects are cached too sparsely, then the latency again does not decrease and caching provides little benefit.

In this section, we describe caching policies governing where an object is cached, along with mechanisms that allow the policies to be implemented locally at each node.

5.3.1 Self-Organizing Caches

Our self-organizing schemes are described here as if every node of the network caches objects, but all that is required is that caches be uniformly distributed. Thus, these schemes obviate the need to decide *where* to place caches—which can be a critical decision for traditional caching mechanisms.

In what follows, the network is considered to be a collection of *domains*, each of which is represented as a graph of switching nodes connected by links. Domains are of two types, *transit*, which (as their name implies) carry transit traffic, and *stub*, through which only packets addressed to or from some node in the domain are carried. The graph models used in our simulations are constructed using the GT-ITM Internet topology modeling package [28]. These graph models ensure that the paths along which packets travel in the simulations have the characteristics that (i) the path connecting two nodes in the same domain stays entirely within that domain, and (ii) the shortest path connecting node u in stub domain U to node v in another stub domain V goes from U through one or more transit domains to V, and does not pass through any other stub domains. Note that "nodes" in these models represent routers, and end systems are not explicitly modeled. Thus references to "servers" or "server nodes" should be interpreted as meaning nodes to which one or more servers are connected.

Our goal is to have nodes make local decisions about which objects they place in their (small) caches, in such a way that resources are used effectively overall. In particular, we wish to avoid having the same (few) objects cached at most of the nodes of the network. We describe two related approaches.

Modulo Caching. To ensure that cached objects are distributed through the network, we introduce a distance measure called cache *radius*, measured in transmission hops. The caching policy uses the radius measure as follows: on the path from the server (or cache) to the requesting client, an item is cached at nodes that are the cache radius apart. Thus, an object ends up being distributed in concentric "rings" centered on the server where it resides; the rings are separated by a number of hops equal to the cache radius. The radius is a parameter of the policy; it might be set globally, on a per-object basis, or even locally in different parts of the network. (Our simulation results assume a common global cache radius, equal to 3.)

The mechanism used to implement this policy locally at each node is a simple modulus. The response message contains a hop count that is initially set to the object identifier modulo the radius; the count is incremented by each node through which the packet passes. When the incremented count modulo the cache radius equals 0, the object is cached at that node.

Our results show that the performance of this *modulo caching* mechanism is extremely robust across a wide range of access patterns and client–server distributions.

Lookaround. Network caches store relatively large objects compared to the amount of space required to store the location of an item within the network. For example, an object in a network cache can be several thousand bytes, while its location could be an IP address (4 bytes). This fact can be exploited by having a self-organizing cache dedicate some of its cache space to store *locations* of (nearby) items.

Caching nodes keep a periodically-updated list of items cached at neighbors. Logically, each node's cache is divided into "levels": level zero contains objects cached locally. Level one contains the locations of objects cached at nodes one hop away, level two contains locations of objects cached at nodes two hops away, etc. When a request message is processed, the levels are searched in sequence beginning with 0; if a hit is detected in a nearby cache, the request is simply re-routed to that node (source and destination addresses are not changed). If the information about the neighbor's cache turns out to be incorrect, the neighbor simply forwards the datagram toward the destination. Thus, the mechanism is fail-safe and backward compatible: a mix of active and non-active nodes may exist in the network, and the active cache functions may fail at any time and fall back on the regular forwarding functions. (In our simulations, we constrain the lookaround to nodes in the same domain.)

The number of levels of adjacent caching maintained and checked in this *lookaround* algorithm becomes a parameter of the policy. With this approach, even very small caches can look like "virtual" large caches. We refer to this extension of the modulo caching scheme as **Modulo Caching with Lookaround**.

5.3.2 Traditional Mechanisms

Our simulations compare the above self-organizing caching schemes to "traditional" caching schemes, in which each cache attempts to store each (popular) object passing through it, without any global coordination beyond the placement of the cache within the network. We consider the following placement strategies:

Cache at Transit Nodes ("Transit-Only"). Transit nodes have to be traversed for every non-local stub domain access; a large fraction of paths in the network have to go through transit routers. This ubiquity of transit nodes in network paths make them prime candidates for caches.

Cache at Stub Nodes Connected to Transit Nodes ("SCT"). Stub nodes connected to transit nodes have to be traversed in order to access the transit network. Thus, these stub nodes form good locations for network caches. The Harvest [23] scheme recommends that caches be placed at such locations.

Cache at Every Node ("No AN"). We also consider an approach in which caches are located in every node (like the self-organizing schemes), but without any coordinating mechanisms enabled. This case corresponds to a "null" active function in an active network. We refer to it as **"No AN"**.

5.3.3 Discussion

Many traditional caching schemes do *not* explicitly minimize latency, but rather minimize bandwidth consumption on transit links. The classic example is the Harvest cache, in which several different queries may be initiated within caches in a domain before either the request is satisfied, or the query is sent to the original server. In many cases, this results in higher latency, but no bandwidth consumption in the transit network. In contrast, our schemes are designed to reduce latency—the bandwidth savings are secondary to reducing pure latency. Both traditional and the self-organizing schemes can be implemented in a network. For example, within a stub network, caches could be set up in a hierarchy to save bandwidth, while in the wide area, the self-organizing schemes could be used to reduce latency.

5.4 Simulation Methodology

We compare the performance of traditional and self-organizing techniques for wide-area caching using a locally developed discrete event network simulator called AN-SWER. AN-Sim simulates an active network as described in [19], and allows for realistic models of the network topology. This section discusses the various parameters of our simulations.

Table 8: Simulated Topologies

Graph	Avg. Deg.	Avg. SD/TN	Avg. SN/SD
`Base`	3.71	4	6
`More Stub Domains`	3.14	6	4
`Less Stub Domains`	4.11	2	12
`Higher Degree`	4.53	4	6
`Lower Degree`	3.06	4	6

Network Topologies. We simulated many networks that differ in number of nodes, diameter, average node degree, ratio of transit nodes to stub nodes, etc. Table 8 summarizes the properties of the topologies for which we present results. All of these graphs have 1500 nodes, of which 60 are transit nodes. In Table 8, SN, SD, TN and TD stand for "stub node", "stub domain", "transit node" and "transit domain", respectively. Thus the `Base` graphs average four stub domains per transit node, and six stub nodes per stub domain.

Servers and Objects. Each stub node is assumed to connect to one server, thus each graph has 1440 servers. A subset of the servers, chosen uniformly at random, are designated to be *popular servers*. The number of popular servers is nominally 300. (One experiment explores the effet of varying the number of popular servers.) There are 4 billion (2^{32}) unique objects in each simulation, the vast majority of which are not accessed. Each object is associated with a particular server, thus each server's content is unique. A subset of objects at each popular server is designated to be popular. (Unpopular servers have only unpopular items.) The number of popular objects is fixed at 48 per popular server (for a nominal total number of popular objects of 14400.) To decide upon a query, first a client is chosen. The client picks a server, then picks an object at the server. The access patterns governing the choice of client, server and object at a server are described below.

Access Patterns Every stub node is assumed to connect to one client. We simulate several different access patterns. In the *uniform access pattern*, a client is chosen at random. Then, a server is chosen using a Zipf distribution: the popular servers are chosen with a probability $1 - \epsilon$. All other client requests go to an unpopular server, chosen at random. If a popular server is selected, then a popular object at that server is selected 95% of the time. The remaining 5% of the accesses to a popular server select from the unpopular objects. If an unpopular server is selected, then a random object is selected. (Recall that all objects at an unpopular server are unpopular.)

We also model a *correlated access pattern*, in which accesses are not independent, but rather may involve the same client and server pairs. In the correlated access pattern, there are two types of accesses: initial accesses and dependent accesses. An initial access is generated using the uniform access pattern described above. Initial accesses are separated in simulation time using an initial access interarrival time distribution. With a fixed correlation probability, an initial access is the "anchor" for a train of future dependent accesses that are interspersed in time with the initial access generation process. A train has an average length of 16 accesses; the time of each future dependent access is determined by an offset distribution from the time of the anchor access. A dependent access has the same client and server as the anchor access. With a fixed probability, the item selected in a dependent access is the same as the previous item in the train. Otherwise, the item is selected according to the uniform access pattern described above.

In the third access pattern simulated, a set of servers is associated with each node. A fraction of all accesses from the node is then directed towards servers in this (per node) set. This models the case in which server popularity is not uniform throughout the network.

These access patterns are consistent with what is known about access to objects in the World Wide Web. Among the "invariants" found by Arlitt and Williamson in their study of server logs [6] was that the number of distinct objects requested is between .3% and 2.1% of the total number of requests. They also found that the vast majority of all requests are for objects that would fit inside a single IP datagram and that at least 75% of all requests to each server they studied are nonlocal.

Cache Sizes In our simulations, the size of the cache at each caching node is proportional to the node degree. This is equivalent to having a fixed amount of caching per edge (interface) incident upon the node. For each caching simulation, the total number of cache slots in the network is held constant. Thus, in cases where the total number of caching nodes is small (as in Transit-Only caching, or caching only in stub nodes connected to transit nodes), the caches are relatively large (compared to cache size when each node has a cache).

Limitations of Experiments. Our experimental setup has some known limitations. The simulator does not enforce link rates, and thus lost datagrams due to full buffers are not represented in the results. The lookaround algorithm does not generate actual packet traffic in the simulation. Also, the space required for storing the list of cached items at neighbors is not accounted for. We do not believe that any of these is particularly serious.

Performance Metrics We use the following metrics to evaluate the performance of network caches.

- **Round trip length (RTL).** We measure the number of hops traversed by the packets involved in a transaction. This is perhaps the simplest and most "true" measure of network cache performance.
- **Fraction of queries that generate cache hits** After an initial startup period, the cache performance stabilizes. We measure the fraction of queries that are serviced by cache hits. Note that queries served by caches not only reduce access latencies and conserve bandwidth, but also reduce server load.

5.5 Simulation Results

We have simulated cache performance for the topologies specified in Table 8 across a wide range of cache sizes, server distributions, and access parameters. We first summarize the general results, and then describe how the results are affected by variations in the certain simulation parameters.

5.5.1 Summary

For non-correlated accesses, the Transit-Only caching scheme performs best (in terms of average RTL). One explanation for this is that for the base graph, there are 4.92 stub nodes, and 3.32 transit nodes on the average client-server path. In this graph, there are 60 transit nodes, and 1440 stub nodes. Thus a much larger fraction of all transit nodes (and thus cache slots) are encountered on each access in the Transit-Only scheme than in the other schemes. Also, for Transit-Only caching, in the base graph, the average cache is 25 times larger than the average cache under active caching. Thus, the large caches lead to large gains due to multiplexing at the transit nodes—especially if the accesses are not correlated.

The active schemes are always within 10% of the Transit-Only RTL, even for uncorrelated accesses. However, Transit-Only is not able to adapt to correlated accesses from particular stubs. Thus the active mechanisms generally outperform all other methods (including Transit-Only caching) for correlated accesses.

In the following subsections we present details of a small cross section of our experiments. Except where otherwise noted, our results are for the base graph topology, and the LRU cache replacement policy.

5.5.2 Variation in Cache Size

We varied the nominal cache size from 4 to 48 cache slots per interface. The Modulo and No-AN methods use nominal cache size, as these methods cache at all nodes.

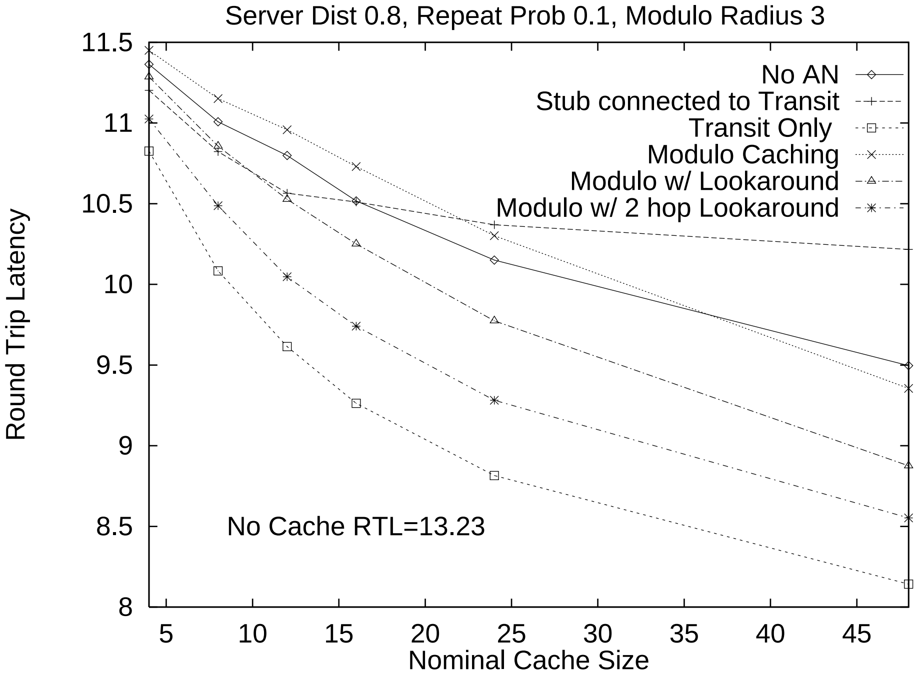

Figure 53: Latency with Low Access Correlation

The corresponding numbers for Transit-Only caching are 33 to 397 slots per interface, and for Stubs-connected-to-Transit (SCT) caching, 15 to 188 slots/interface. All of the caching mechanisms, except SCT, show a smooth decrease in number of hops traversed per round trip as the cache size is increased.

In Figure 53, the probability of repeating an accesses within a set of correlated accesses is 0.1. The Modulo cache radius is fixed at three. The Lookaround schemes perform better than all but the Transit-Only caching scheme, and the performance of the two-level Lookaround schemeis within 10% of the Transit-Only scheme in all cases. It should be noted that the number of cache slots per interface for the two-level Lookaround scheme is an order of magnitude smaller than for the Transit-Only scheme. Also, the average degree of the transit nodes is much greater than the average degree of the graph. Thus the transit node caches are 25 times larger than the Modulo caches. Comparatively, the SCT caches average 4.25 times larger than the Modulo caches.

As accesses become more correlated, as shown in Figure 54 (repeat probability 0.5), the Modulo with Lookaround scheme outperform all others. Also significant in Figures 53 and 54 is the behavior of the caches in the SCT scheme. Their performance improvements are negligible beyond 12 cache slots per interface, and as such this method does not scale well with increase in cache size.

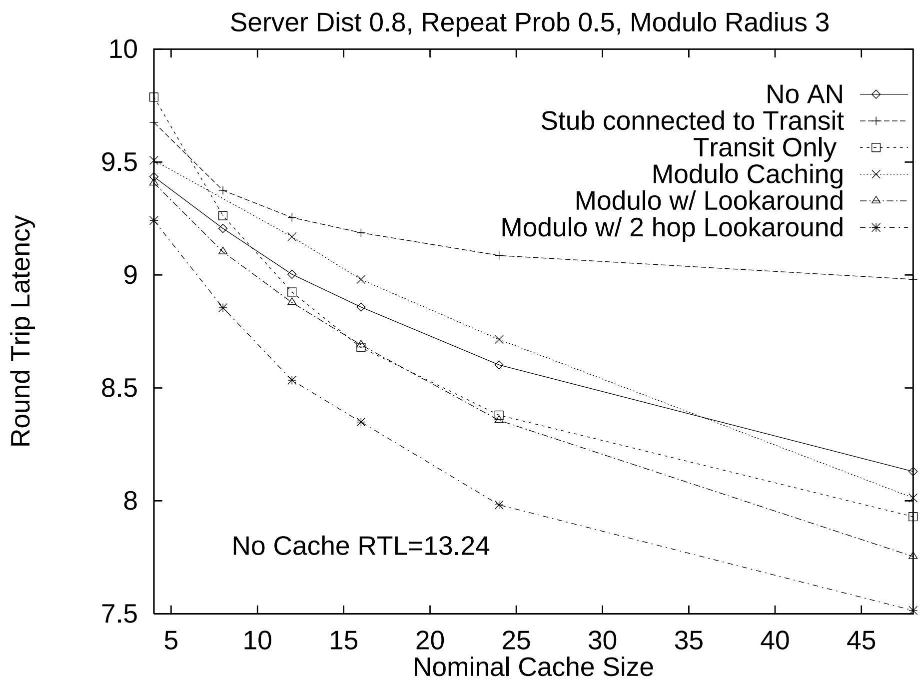

Figure 54: Latency with High Access Correlation

For the same parameters as in Figure 54, Figure 55 shows the fraction of queries that generate cache hits. Once again, all the methods except SCT show improvement with increase in cache size. The SCT method actually results in a proportionately large fraction of hits—but the large number of hops in the round trip suggests that more hits occur in the stub node to which the *server* is connected, and not the client. This is not unexpected—the *gateway* stub node connected to the transit domain for a busy server will experience a lot of traffic due to the busy server, and as such, will cache a large part of that data as well.

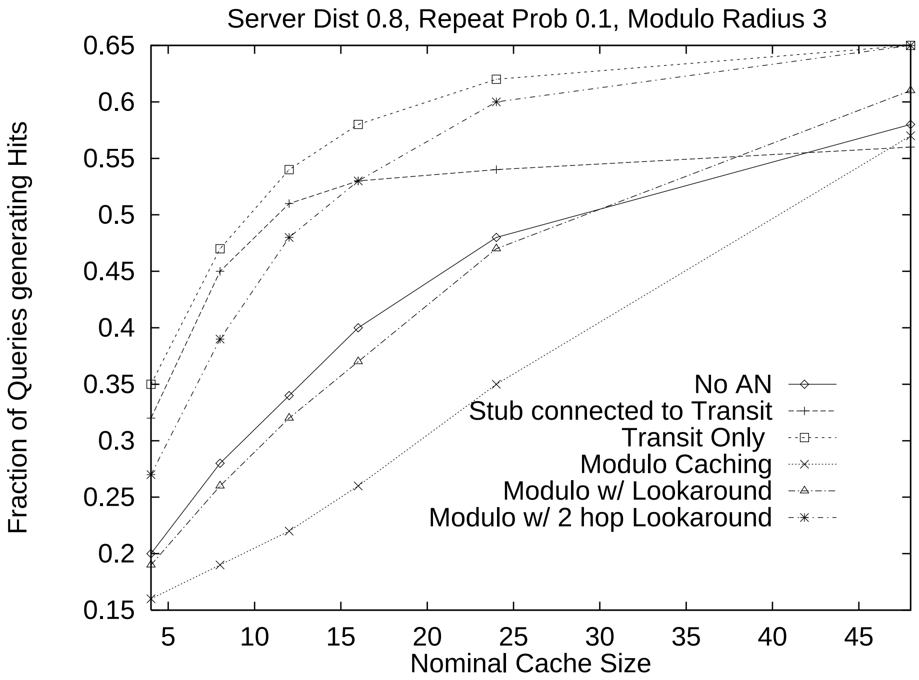

Figure 55: Cache Hits with High Access Correlation

5.5.3 Variation in Server Distribution

We have used a Zipf distribution to model server popularity (i.e. a fraction $1-\epsilon$ of the accesses are to the fraction ϵ of the nodes). However, it is not clear exactly what fraction of the nodes should be considered to be *servers*. In this experiment (Figure 56), we vary the fraction of all nodes that are servers from 0.01 to 0.5. Even when a large fraction of nodes are servers, the cache performances are not significantly affected. Thus, wide-area caching seems robust in face of widely varying server locations and distributions. It is interesting to note that round trip latencies for Transit-Only cache schemes do not improve much when the server distributions are extremely skewed, e.g. when less than 10% of the nodes are servers. The other schemes improve as the number of popular objects (which is a multiple of the number of servers) decrease, but in case of Transit-Only caches, even if all the objects are cached everywhere, the query has to reach the transit nodes before it is serviced.

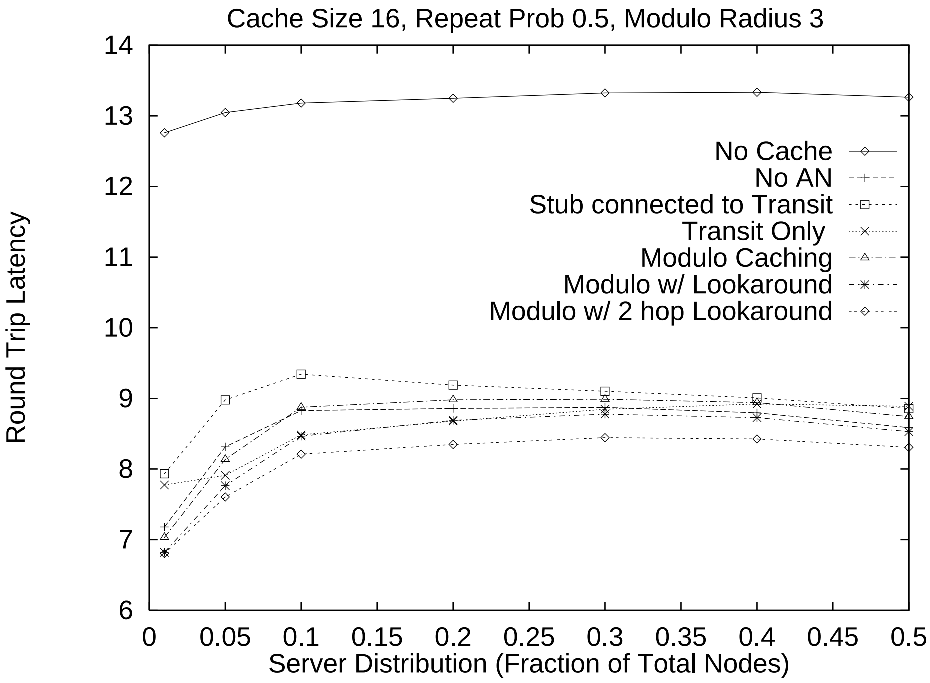

Figure 56: Latency with Varying Server Distributions

5.5.4 Variation in Topology

In Figure 57 we consider the effectiveness of the various cache policies as the underlying topologies are varied. The round trip latency is inversely proportional to the graph degree, and directly proportional to the average size of the stub domains. The two level lookaround algorithm has similar performance to the Transit-Only cache method for the base graph for this moderately correlated access pattern. As topologies are varied, the performances are comparable—with the active methods performing better as the number of stub domains increase. Although it is not presented graphically here, our experiments showed that as repeat probabilities increase, the AN mechanisms perform better than all other mechanisms in all topologies.

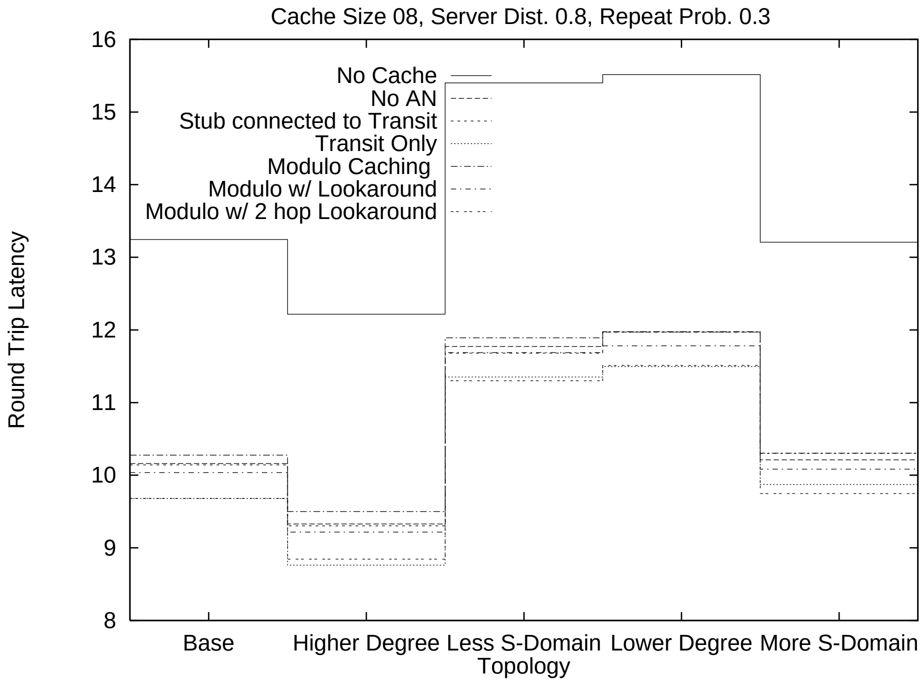

Figure 57: Topology Effect under Moderate Access Correlation

5.5.5 Spatial Access Patterns

In these experiments, we consider a different access pattern. Associated with each stub node is a set of *preferred* servers. A fraction of queries generated by the node is always directed to the preferred set. Figure 58 shows the round trip latencies as the number of servers in the preferred set is varied from 2 to 12. The probability of accessing a server in the preferred set is 0.25, and the probability of a repeat access was 0.3. The two level Lookaround and the Transit-Only cache schemes perform the best, with the Lookaround schemes being better if the number of preferred servers is small. In fact, the round trip latency for Transit-Only caching is nearly constant for any number of servers in the preferred groups since the accesses are nearly uniform at the transit domains. The performance of the Lookaround schemes deteriorate slowly as the number of servers increase because the locality in accesses are lost.

We also considered spatial access patterns in cases when the preferred set is common to the entire stub domain. As expected, the round trip latencies increase with

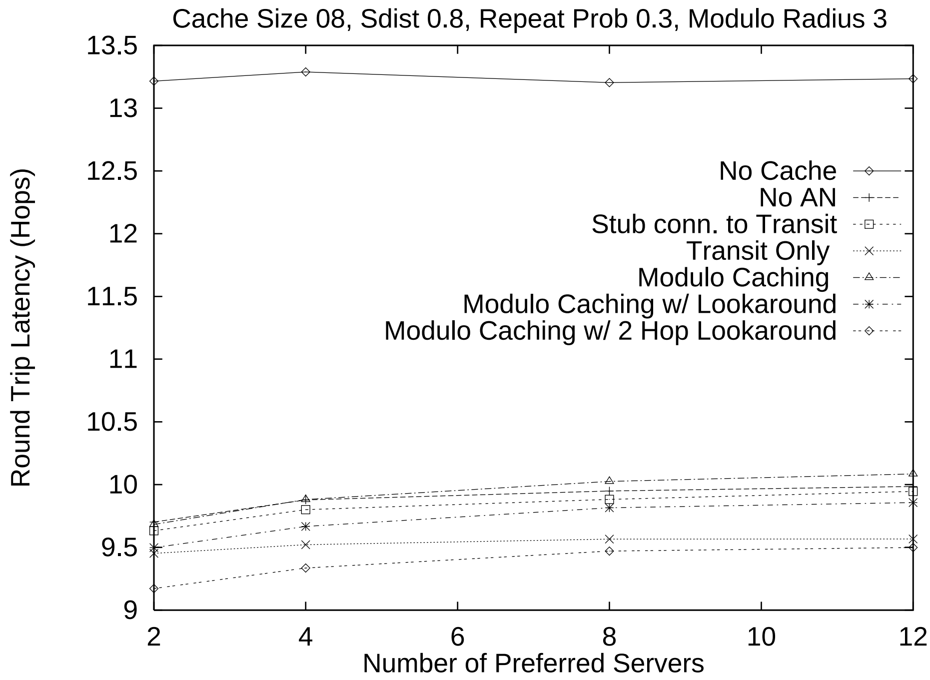

Figure 58: Preferred Node Servers

increase in the number of preferred servers. Also, caching at stub nodes connected to transit node scheme works quite well: it is within 1 hop per round trip of optimal in all cases.

5.6 Analysis of Cache Performance

In this section, we present a simple analytic model for the expected one-way latency in accessing an item in a network that has intra-network caching. The basic model makes the following assumptions: (1) the set of items is partitioned into "popular" items and "unpopular" items, and (2) only popular items are cached, (3) every cache is full, and (4) the cached items encountered from one client access to the next are random and independent. This independence assumption keeps the model tractable, though it clearly departs from reality.

5.6.1 Network Caching Performance Model

Consider an access from a client to an item. If the item is unpopular, the one-way latency is simply the average client-server path length. If the item is popular, it may be encountered at a cache in the network. Let L denote the average one-way latency, measured in hop count, when a client accesses an item. Let q denote the probability that the item is popular. Let d denote the average length of a client-server path and c denote the average length of a path from a client to a popular item (possibly encountered at a cache). Then:

$$L = (1-q)d + qc$$

Let N_s denote the average path length from a stub node to the transit node for a stub domain. Let N_t denote the average path length across transit domains. Then $d = 2N_s + N_t$, since a client-server path traverses two stub domains (client and server domains) and a path through the transit domains.

We focus the rest of the discussion on deriving an expression for c. Let H_i denote the event "a hit occurred at distance i from the client"; let M_i denote the event "a miss occurred at *every* distance up to and including i from the client". Then

$$\begin{aligned} c &= \sum_{i=1}^{d} i Pr\{H_i \cap M_{i-1}\} \\ &= \sum_{i=1}^{d} i Pr\{H_i \mid M_{i-1}\} Pr\{M_{i-1}\} \end{aligned}$$

The probabilities in this expression are affected by details of the locations of the caches, the cache policy and the client access pattern. We develop variations on the basic model to handle different instances of these details.

Let us assume the best possible conditions regarding duplication in items stored in caches along the path. Namely, we assume that the items stored in different caches are *different*. Though this is difficult to achieve in a distributed environment, it provides a lower bound on average round-trip length. We also assume that each cache is equally likely to store any popular item.

Since all items are different, the probability of a hit when accessing a popular item and examining S cache locations (potentially distributed over multiple nodes) is S/P. If we know that a miss has occurred at locations at every distance up to and including $i-1$, then we know that the item we are looking for is in a reduced set of popular items, namely reduced by the number of cache locations we have already checked. Let T_{i-1} denote the total number of items checked at distances up to and including $i-1$. Then[2] $T_{i-1} = \sum_{j=1}^{i-1} S_j$ and:

$$
\begin{aligned}
c &= \sum_{i=1}^{d} i Pr\{H_i \mid M_{i-1}\} Pr\{M_{i-1}\} \\
&= d\left(1 - \frac{T_{d-1}}{P}\right) + \sum_{i=1}^{d-1} i \frac{S_i}{P - T_{i-1}} \left(1 - \frac{T_{i-1}}{P}\right) \\
&= d\left(1 - \frac{T_{d-1}}{P}\right) + \sum_{i} i \frac{S_i}{P}
\end{aligned}
$$

The first term is the boundary condition for the request reaching the server.

5.6.2 Optimal cache partitioning

As discussed in Section 5.3, it may be beneficial to use some of the network cache's memory to store locations of *nearby* objects. In this section, we analyze the optimal partition of the network caches into object and location caches.

Assume that each object is the same size, and let the maximum number of objects in each cache be S. Let the ratio of the size of an object and the size of a location be α, i.e. α locations can be stored in the cache instead of one object. Let the average one-way length of server–client paths be d. We assume paths are symmetric, and as such, the average client–server round trip latency is $2d$. Starting from a given cache, as the level of lookaround is increased, the number of caches searched increases. For most graphs, this increase is exponential. As a simplification, we assume that the number of caches searched at a lookaround level of x is exponential in $\delta - 1$, where δ

[2] We assume that $T_i < P$ for all i. If this assumption does not hold, then at some point in the path, all popular items will have been encountered, and the hit probability will go to 1.0.

Table 9: Definitions and Notation

S	–	Max. number of objects in cache
α	–	Number of object locations that can be
	–	stored instead of one object
δ	–	Average degree of the graph
d	–	Average one-way length of server–client path
η	–	Frac. of cache used for storing objects

is the average degree of the topology graph[3]. Let η denote the fraction of the cache memory is used for caching objects; the rest is used for caching location information.

We now derive the value of fraction η such that the round trip latency is minimized. The precise fraction that minimizes the latency depends on the size of the network, the size of the caches, and the number of locations that can be stored instead of each object. Using the basic caching model developed before, the expected latency is $c = \sum_i iPr\{H_i \mid M_{i-1}\}Pr\{M_{i-1}\}$.

Using our model of memory usage, the expected latency can be calculated as follows. For the local objects (in the object cache), the latency after traveling i hops is i. If at node i, there is a hit in the location cache, then a detour has to be made. We can approximate the one-way length of the detour as follows.

We know that if η fraction of the memory is used for caching objects, then memory for $(1-\eta)S$ objects is used for storing locations. This is equal to $(1-\eta)S\alpha$ locations. As each cache stores ηS objects, the $(1-\eta)S\alpha$ locations correspond to complete location information from $\frac{(1-\eta)\alpha}{\eta}$ nodes. Under the assumption that the number of nodes increases exponentially as the level of lookaround increases, $\frac{(1-\eta)\alpha}{\eta}$ nodes imply a maximum detour of length $\log_{\delta-1}\frac{(1-\eta)\alpha}{\eta}$. Let f be the expected value of the length of this detour, and let $F = \log_{\delta-1}\frac{(1-\eta)\alpha}{\eta}$ be the maximum length of the detour. At each level x of lookaround, $(\delta-1)^x\eta S$ items are cached. Thus, the

[3]The exponent is $\delta - 1$ because after the first level of lookaround, at least one node has already been searched at the previous lookaround depth

expected one way detour length is:

$$f = \sum_{x=1}^{F} x \frac{(\delta-1)^x \eta S}{(1-\eta) S \alpha}$$

If we assume that the lookaround memory is arranged such that the content of nearest caches are searched first (i.e. caches at level 1 are searched before caches at level 2, and so on), then we can derive a simpler expression for the expected value of the detour. Under this case, on average, half of the cache will be checked before a hit (if any) occurs in the lookaround cache. Thus, the expected length of a detour in this case is:

$$f = \log_{\delta-1} \frac{(1-\eta)S\alpha}{2\eta S} = \log_{\delta-1} \frac{(1-\eta)\alpha}{2\eta}$$

As previously defined, let P be the total number of popular items, and let T_{i-1} be the total number of items checked till distance $i-1$ from the source of the request. In our case, as the maximum number of objects in a cache assumed to be equal, $S_i = S$ (for all i), and $T_{d-1} = (d-1)S$.

Thus, the expected value of the one way latency is:

$$\begin{aligned}
c &= \sum_{i}^{d} i Pr\{H_i \mid M_{i-1}\} Pr\{M_{i-1}\} \\
&= d\left(1-(d-1)\frac{S}{P}\right) + \sum_{i}^{d} i \frac{S_i}{P} \\
&= d\left(1-(d-1)\frac{\eta S + (1-\eta)S\alpha}{P}\right) + \\
&\quad \left(\frac{\eta S}{P} + \frac{(1-\eta)S\alpha}{P}\right)\frac{d(d+1)}{2} + \frac{(1-\eta)S\alpha}{P} f
\end{aligned}$$

Minimizing this expression with respect to η gives the optimal amount of cache to be used for storing location information such that the expected latency is minimized. In Figure 59, we compare the result of the analysis of lookaround caching with a simulation with similar parameters. The base graph (see Table 8) was used in

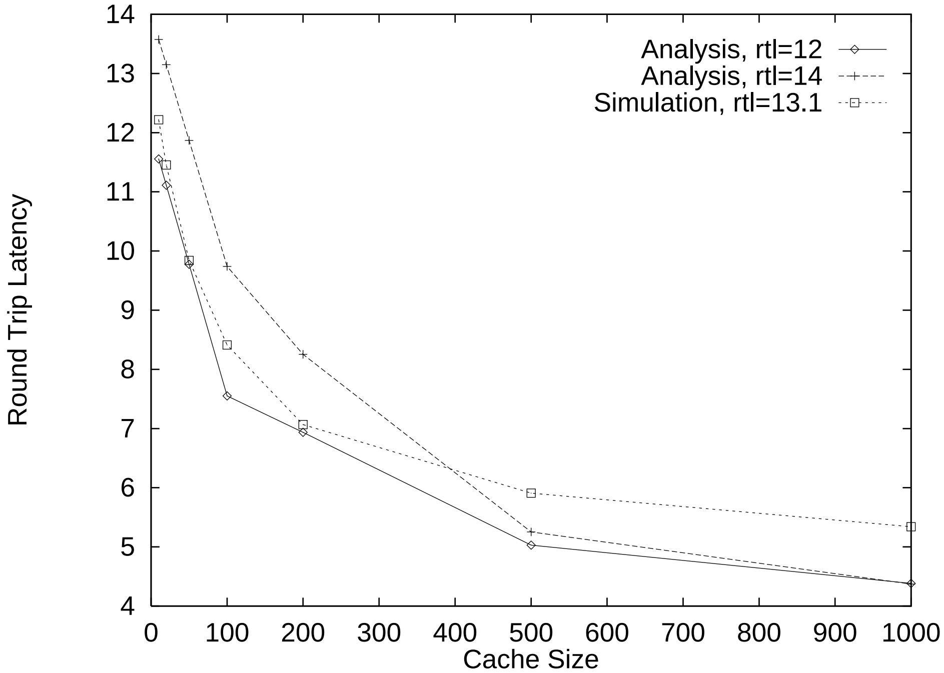

Figure 59: Comparison of analytic and simulation results — Lookaround Caches

the simulation, with α equal to 50. The lookaround was fixed to 2 levels, and the radius of caching at 3. The two analytic curves were generated by setting the one-way client–server path length to 6 and 7, respectively. This corresponds to the average length of client–server round trip lengths being 12 and 14, respectively. Thus, in general, we would expect the curve generated by simulation to be bounded (on either side) by the two curves generated by the analysis. For small cache sizes ($S < 500$), this is the case. The refinements to the basic model generate an extremely accurate measure of the cache latency. When the cache sizes are larger, the analysis overestimates the number of *unique* items that are cached.

In Figure 60, we use the expression for expected round trip latency to evaluate the benefits of lookaround caching. The x-axis shows the size of each cache, the y-axis denotes the amount of the cache memory devoted to caching local objects. The round trip latency is shown on the z-axis. We assume that nearer caches are checked earlier to evaluate the length of the detour. In this plot, we set the number

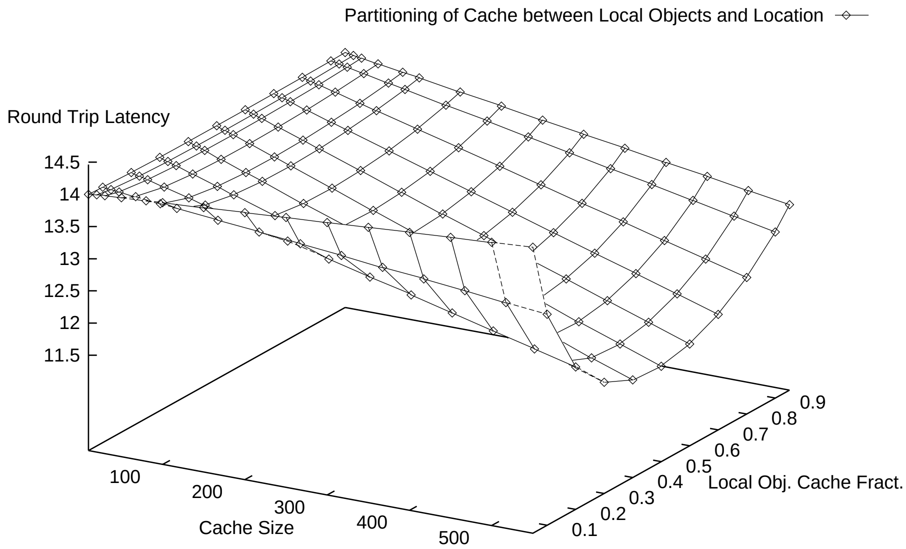

Figure 60: Optimal partitioning of cache space for local objects and location information

of popular objects in the network to 80,000, and again, α is set to 50. The average client–server round trip path length was set to 14. The plot clearly shows the benefit of lookaround caching — the *valley* in the plot is directly due to the benefits of lookaround caching. In the plot, the expected round trip latency decreases as the amount of memory devoted to the lookaround cache is increased ($.99 < \eta < 0.5$). However, after a point, as more memory is dedicated to the lookaround cache, the number of objects cached in the network decreases. The round trip latency increases sharply ($0 < \eta < 0.3$) as there are not enough objects cached in the network and each lookaround hit causes a large detour. It is interesting to note that at very small values of η, the round trip latency can increase to greater than the average latency without caching—this is again expected, as there are not enough objects cached, and each detour is greater than the remaining path length to the server. Of course, a trivial optimization that forbids detours greater than the remaining path length can easily be incorporated.

5.7 Conclusions

We have developed and evaluated mechanisms for caching objects within the network. Our active caching mechanisms allow transparent, self-organizing, location of objects where they can be of benefit to reduce access latency. These methods contrast with traditional wide-area network caching that relies on fixed and limited locations for caches.

We have used both simulation and an analytic model to evaluate performance. The simulation uses access patterns that are consistent with studies of Web access; the analytic models are simplified for tractability, though still provide reasonable estimates of access latency. The models could be extended using information about uniqueness of items and/or more complex mathematical modeling to remove simplifying assumptions.

Our results show that active caching is beneficial across a range of network topologies and access patterns, and is especially effective when access patterns exhibit significant locality characteristics.

Chapter 6

Conclusions

This thesis investigates three aspects of active networking: architectures for active networks, composition of active network services, and applications for active networks. In Section 6.1, we present a summary of our contributions to each of these areas. Based on our experience in designing and implementing active network components, in Section 6.2 we draw a set of conclusions about the advantages, the disadvantages and the future potential of active networks.

6.1 Summary of Contributions

6.1.1 Architecture

We have developed an active network architecture that can export multiple simultaneous user-network interfaces[1]. These user-network interfaces, implemented as execution environments (EEs) at active nodes, can be specialized for different purposes: e.g. some interfaces can be designed to provide higher forwarding performance at the cost of per-flow flexibility or vice-versa. The active network architecture provides a mechanism for active nodes to co-exist and co-operate with the existing network infrastructure, e.g. versions of the IP protocol can be implemented as EEs within active nodes; traditional IP routers correspond to network nodes that only support the IP EE. The architecture provides an explicit path through the node for

[1]Our active networking architecture was eventually adapted by the DARPA active networking community and has subsequently been refined by many researchers in the community.

non-active packets such that these packets do not have to incur (except for an extra packet classification step) overhead due to active processing.

The active network architecture followed our work on the Control-on-Demand (CoD) active networking environment. CoD contributed several new ideas for active networking, including flow-based processing and best-effort processing. Flow-based processing amortizes the cost of downloading, installing and executing code over the life time of end-to-end packet flows; best-effort processing allows active and non-active routers to co-exist in an internetwork. Like the DARPA architecture, the CoD design preserves a separation between a Node OS and an execution environment. CoD was implemented over Linux using IPv6, CoD "controllers" for video streams were developed and tested on a CoD testbed. In later work, we have contributed to the Odyssey project that includes the Bowman Node OS and the CANEs EE and implements the complete DARPA active networking architecture.

6.1.2 Composition

We have introduced the slot programming model for composing network services in active networks. The slot programming model provides a modular, extensible platform for composing network services. The underlying program abstraction identified by the slot programming model encapsulates uniform per-packet processing for each packet in a flow; implementation of underlying programs can potentially be highly optimized (possibly cast in hardware) as their execution do not change on a per-packet basis. Further, underlying programs provide a platform for network providers to control the level of programmability and flexibility given to users of active nodes. We have implemented the slot programming model within the CANEs EE as part of the Odyssey active network. We have developed an API for programming using slots and have designed and implemented a signaling protocol for CANEs. We have designed and implemented underlying programs for multicast and unicast forwarding within CANEs. A multicast error recovery protocol has been implemented as an injected program for CANEs[2].

6.1.3 Applications

We have introduced application-specific congestion control (ASCC) as an application for active networks. Using ASCC, applications install advice —in the form of computations and state— at network nodes. The advice specifies how the application's data stream should be handled upon congestion. We have developed application-specific discard techniques for MPEG video, implemented our schemes for unicast flows, and evaluated its performance. We have simulated extensions of ASCC for multicast video distribution. Our results show that ASCC can deliver 2-3 times more useful video frames over a wide-range of congestion scenarios. In order to reduce per-flow cost of ASCC, we have designed and simulated video discard algorithms that do not require per-flow state; instead these techniques operate on aggregate flows that consist of several individual video sub-flows.

We have introduced self-organizing network caching schemes as another application for active networks. Using active networking, we have developed techniques for co-ordinating large numbers of relatively small caches that are located inside the network. Unlike traditional caching schemes, our techniques do not require that caches be placed at certain nodes in the network. We have simulated our caching algorithms and have presented a general analytic model for in-network caching. Our results show that self-organizing caching schemes , using caches that are an order of magnitude smaller, can outperform traditional mechanisms.

[2]This work was done as part of a DARPA project in co-operation with the University of Massachusetts and The Analytic Sciences Corporation (TASC).

6.2 On Active Networking

Active networks represent a revolutionary change in data networks: they transform the network from a passive carrier of bits to a distributed programmable computational engine. In this section, we present our views on the advantages of such a change, some disadvantages that are inherent to active networks, and discuss some open problems that must be solved in order for active networks to be widely deployed.

Active networks provide an extensible network infrastructure; new protocols and services can be added to the network on the fly and network response can be tailored by users. As our ASCC and caching results show, active networks provide also provide a platform to merge application- and network-level information. Using an environment like CANEs, new network services can be composed and rapidly deployed in active networks. These capabilities translate into a network that can be tailored for its users and can, ultimately, provide better end-to-end performance.

There, however, are two issues —scale and security— that seem to be inherent as disadvantages of active networks. The issue of scale essentially is equivalent to the problem of building active nodes that can support a large number of simultaneous active flows. The security aspect covers problems of building a secure active node, securing entire active networks, and containing security breaches. Next, we examine these two issues in detail.

- Scaling properties of current routers are predicated on the fact that most packets require relatively simple and uniform processing. Active flows, on the other hand, may invoke different code; further, there is no a-priori bound on the complexity of the functions invoked by each flow. Since some of the invoked code may be loaded into the node at run-time, traditional optimization techniques (such as hardware implementations of functions) do not necessarily apply to active networks. Active processing inherits traditional overheads of per-user computations, such as processor dispatching and context switching latency. These problems are compounded at an active node because the overheads apply on finer granularity timescales — the processor is forced to switch contexts after processing a small number of packets. Due to these problems, it is currently not clear how to build an active router that can scale to beyond hundreds or possibly thousands of simultaneous active flows if each flow requires different active processing.

- The security disadvantage of active networks is not as well defined. There are three main reasons active networks are likely to be more difficult to secure than traditional networks:

Active networks provide a richer interface than traditional networks. A richer interface, especially a programmable interface, inherently provides more *opportunities* for breaches in security. As the code executed by an active router can be loaded on the fly, it is difficult (and in some cases impossible) to provide assurances about the behavior of the network.

Depending on its API, an active network executes a set of functions on behalf of an user. In many cases, these functions are loaded into the node by the user. Active nodes must authenticate code before executing them on behalf of users. Thus, a strong authentication infrastructure will be required to securely implement an active network. Building and maintaining a wide-area authentication infrastructure is a non-trivial task in itself.

If the security of an active node is violated, the programmable API of an active network provides opportunities for attacks that are not possible in traditional networks. Security violations of traditional networks can result in eavesdropping on channels or denial of service to parts of the network. However, active networks routinely accept new code to the run on behalf of users; a single compromised active router may be able to sign and upload malicious code on active routers in the entire network.

6.2.1 Open Problems

The following problems provide opportunities for future work and must be solved before active networks can be widely deployed.

- **Scheduling** As mentioned above, traditional scheduling techniques are likely to be inadequate at active nodes that support a large number of user flows. An open problem is to integrate link scheduling and processor scheduling algorithms such that nodes can provide bandwidth and delay guarantees for flows that invoke active processing. This problem must be addressed in order to construct scalable and high performance active nodes.

- **Resource allocation between competing flows** The Node OS provides access to resources at an active node; it is possible for users (or EEs) to specify contending accesses to node resources. Resource allocation algorithms that conform to node-wide policies are an open research problem. Another open research area is the specification and formulation of well-founded policies that optimize node-wide metrics.

- **Global resource management** In the Internet, each IP packet carries a time-to-live value (TTL) in the packet header. The TTL value is decremented at each IP router and packets are discarded if the TTL reaches zero; thus, the TTL in the packet header bounds the number of hops that a single IP packet can visit. This is important because packets consume network resources (e.g. network bandwidth) and in case of routing loops, the amount of resource consumed by any single packet is limited by its TTL. In an active network, packets consume network bandwidth and can trigger significant processing at routers. An open problem is to bound the amount of processing that can be initiated by a single active packet; an even more difficult challenge is to be able to allocate and manage bandwidth and processing resources in a large internetwork for long lived active flows.

6.3 Concluding Remarks

Currently, active networks are a research platform for testing hypotheses about network programmability. There are several possible futures for active networking: the open problems may prove to be too difficult to solve and active networking could forever remain an academic curiosity. It is possible that the hard problems, including policy issues, in active networking will all be solved and all networks, some day, will be active and end-user-programmable. A future that is somewhere in between these two extremes is the most plausible. In the beginning, results from active network research will be added as *features* on to the existing infrastructure. Eventually, routers will support programmability; however, instead of all users of the network, only node-providers and trusted principals will be allowed to update the code that runs in routers.

Bibliography

[1] D. Alexander, W. Arbaugh, M. Hicks, P. Kakkar, A. Keromytis, J. Moore, C. Gunter, S. Nettles, and J. Smith. The Switchware Active Network Architecture. *IEEE Network Special Issue on Active and Controllable Networks*, 12(3):29–36, 1998.

[2] D. Scott Alexander, Michael W. Hicks, Pankaj Kakkar, Angelos D. Keromytis, Marianne Shaw, Jonathan T. Moore, Carl A. Gunter, Trevor Jim, Scott M. Nettles, and Jonathan M. Smith. The Switchware Active Network Implementation. In *The 1998 ACM SIGPLAN Workshop on ML held in conjunction with the International Conference on Functional Programming (ICFP) '98.*, 1998.

[3] D. Scott Alexander, Marianne Shaw, Scott M. Nettles, and Jonathan M. Smith. Active bridging. In *Proc. of ACM SIGCOMM '97*, Cannes, France, 1997.

[4] E. Amir, S. McCanne, and H. Zhang. An application level video gateway. In *The Third ACM International Multimedia Conference and Exhibition (MULTIMEDIA '95)*, pages 255–266, New York, November 1996. ACM Press.

[5] Elan Amir. An application level video gateway. Master's thesis, University of California, Berkeley, Berkeley, California, December 1995.

[6] M. Arlitt and C. Williamson. Web server workload characterization: The search for invariants. In *Proceedings of ACM SIGMETRICS '95*, May 1995.

[7] G. Armitage and K. Adans. Packet reassembly during cell loss. *IEEE Network Magazine*, September 1993.

[8] Ken Arnold and James Gosling. *The Java Programming Language.* Addison Wesley, Reading, Massachusetts, 1996.

[9] Abhaya Asthana, Catherine Delph, H. V. Jagadish, and Paul Krzyzanowski. Towards a gigabit IP router. *Journal of High Speed Networks*, 1(4):281–288, 1993.

[10] Hari Balakrishnan, Srinivasan Seshan, Elan Amir, and Randy H. Katz. Improving TCP/IP performance over wireless networks. In *Proc. of 1st ACM Conference on Mobile Computing and Networking*, Berkeley, California, November 1995.

[11] Joe C. R. Bennett and Hui Zhang. Hierarchical packet fair queueing algorithms. In *ACM Sigcomm '96*, pages 143–156, Stanford, California, August 1996.

[12] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak an d Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, Safety and Performance in the SPIN operatin g system. *Operating Systems Review (Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles)*, 29(5):267–284, December 1995.

[13] S. Bhattacharjee, K. Calvert, and E. Zegura. Reasoning about active network protocols. Proceedings of ICNP '98, 1998.

[14] S. Bhattacharjee, K. L. Calvert, and E. W. Zegura. On active networking and congestion. Technical Report GIT-CC-96-02, College of Computing, Georgia Institute of Technology, 1996.

[15] S. Bhattacharjee, K. L. Calvert, and E. W. Zegura. Network support for multicast video distribution. Technical Report GIT-CC-98-16, College of Computing, Georgia Institute of Technology, 1998.

[16] S. Bhattacharjee and M. W. McKinnon. Performance of application-specific buffering schemes for active networks. Technical Report GIT-CC-98-17, College of Computing, Georgia Institute of Technology, 1998.

[17] S. Bhattacharjee, S. Merugu, Y. Chae, E. Zegura, and K. Calvert. Exposing the network: Support for topology-sensitive applications. Submitted to International Conference on Network Protocols (ICNP) 1999, 1999.

[18] Samrat Bhattacharjee, Ken Calvert, and Ellen Zegura. Active networking and the end-to-end argument. In *Proceedings of ICNP'97*, 1997.

[19] Samrat Bhattacharjee, Kenneth. L. Calvert, and Ellen W. Zegura. An Architecture for Active Networking. In *Proceedings of High Performance Networking 97*, 1997.

[20] Steven L. Blake, Sarah A. Rajala, and Fengmin Gong. Efficient techniques for two-layer coding of video sequences. Technical report, North Carolina State University, Raleigh, NC, November 1993.

[21] J.-C. Bolot, T. Turletti, and I. Wakeman. Scalable feedback control for multicast video distribution in the Internet. In *ACM Sigcomm '94*, 1994.

[22] F. Bonomi and K. Fendick. The rate-based flow control framework for the available bit rate ATM service. *IEEE Network Magazine*, March/April 1995.

[23] C. M. Bowman, P. Danzig, D. Hardy, U. Manber, M. Schwartz, and D. Wessels. Harvest: A scalable, customizable discovery and access system. Technical Report CU-CS-732-94, University of Colorado - Boulder, 1995.

[24] B. Braden, A. Cerpa, T. Faber, B. Lindell, G. Phillips, and J. Kann. ASP EE: An Active Execution Environment for Network Control Protocols. Available at `http://www.isi.edu/active-signal/ARP/`.

[25] R. Braden, Ed., L. Zhang, S. Berson, S. Herzog, and S. Jamin. RFC 2205: Resource ReSerVation Protocol (RSVP) — version 1 functional specification, September 1997. Status: PROPOSED STANDARD.

[26] H. Braun and K Claffy. Web traffic characterization: An assessment of the impact of caching documents from NCSA's web server. In *Electronic Proceedings of the Second World Wide Web Conference '94: Mosaic and the Web*, October 1994.

[27] K. Calvert, S. Bhattacharjee, E. Zegura, and J. Sterbenz. Directions in active networks. IEEE Communications Magazine, 1998.

[28] Kenneth L. Calvert, Matthew B. Doar, and Ellen W. Zegura. Modeling Internet Topology. *IEEE Communications Magazine*, June 1997.

[29] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[30] Kim Claffy, Greg Miller, and Kevin Thompson. The Nature of the Beast: Recent Traffic Measurements from an Internet Backbone. In *Proceedings of the Global Internet Conference*, 1998.

[31] D. Clark and D. Tennenhouse. Architectural considerations for a new generation of protocols. In *ACM SIGCOMM '90*, 1990.

[32] David D. Clark. The Design Philosophy of DARPA Internet Protocols. In *In Proceedings of ACM SIGCOMM'88*, Stanford, CA, August 1988. ACM.

[33] Richard V. Clayton. *Structuring and Destructuring Protocols*. PhD thesis, Georgia Institute of Technology, 1999.

[34] WheelGroup Corporation. Analysis of Internet Attacks - ProWatch Secure Network Security Survey. Published on the Internet, May 1997. http://www.wheelgroup.com/netrangr/PWS_survey.html.

[35] Steve Crocker. Host Software. Network Working Group, RFC 1, 7 April 1969.

[36] Sushil da Silva, Danilo Florissi, and Yechiam Yemini. Composing active services in netscript. In *Proc. of DARPA Active Networks Workshop*, page 6, Tucson, Arizona, March 1998.

[37] C. A. Dahlbom and J. S. Ryan. Common channel interoffice signaling: Signaling network. *Bell Systems Technical Journal*, 57(2):263–282, February 1978.

[38] P. Danzig, M. Schwatrz, and R. Hall. A case for caching file objects inside internetworks. In *Proceedings of ACM SIGCOMM '93*, September 1993.

[39] Peter B. Danzig and Sugih Jamin. tcplib: A library of TCP/IP traffic characteristics. *USC Networking and Distributed Systems Laboratory TR CS-SYS-91-01*, October, 1991. ftp://catarina.usc.edu/pub/jamin/tcplib.

[40] G. Deloche. ARPA Network Functional Specification. Network Working Group, RFC 8, 5 May 1969.

[41] Kenneth L. Calvert (Editor). Architectural Framework for Active Networks. DARPA AN Working Group Draft, 1998.

[42] Larry Peterson (Editor). NodeOS Interface Specification. DARPA AN NodeOS Working Group Draft, 1999.

[43] Dawson R. Engler, M. Frans Kaashoek, and Jr. James O'Toole. Exokernel: An Operating System Architecture for Application-level Resource Management. *Operating Systems Review (Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles)*, 29(5):251–266, December 1995.

[44] J. Enssle. Modelling and statistical multiplexing of VBR MPEG compressed video in ATM networks. In *Proceedings of the 4th Open Workshop on High Speed Networks, IND, Universitt Stuttgart*, page 10, Brest, France, 1994.

[45] D. Scott Alexander et. al. Active Network Encapsulation Protocol (ANEP). http://www.cis.upenn.edu/ switchware/ANEP/docs/ANEP.txt, 1997.

[46] David Black et. al. An Architecture for Differentiated Services. IETF Diffserv Working Group Internet Draft, May 1998. draft-ietf-diffserv-arch-00.txt, Expires November 1998.

[47] Larry Peterson et al. Scout Joust. `http://www.cs.arizona.edu/scout/joust.html`.

[48] T. Faber. ACC: Using Active Networking to Enhance Feedback Congestion Control Mechanisms. *IEEE Network*, 12(3):61–65, May/June 1998.

[49] D. Feldmeier. A data-labelling technique for high-performance protocol processing and its consequences. In *ACM SIGCOMM '93*, 1993.

[50] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[51] International Organisation for Standardisation. Generic coding of moving pictures and associated audio. ISO/IEC/JTC1/SC29/WG-11, March 1993.

[52] Rahul Garg. Characterization of video traffic. Technical Report TR-95-007, International Computer Science Institute, Berkeley, CA, 1995. Available at ftp://ftp.icsi.berkeley.edu/pub/techreports/1995/ tr-95-007.ps.gz.

[53] J. Gwertzman and Margo Seltzer. The case for geographical push caching. In *Hot Operating Systems*, 1995.

[54] Katie Hafner and Matthew Lyon. *Where Wizards Stay Up Late: The Origins of the Internet.* Simon and Schuster, 1998.

[55] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A packet language for active networks. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming Languages*, pages 86–93. ACM, 1998. Available at `www.cis.upenn.edu/ ~switchware/ papers/ plan.ps`.

[56] Michael Hicks, Jonathan T. Moore, D. Scott Alexander, Carl A. Gunter, and Scott M. Nettles. PLANet: An Active Internetwork. In *In Proceedings of IEEE INFOCOM '99, New York, NY.*, March 1999.

[57] Gísli Hjálmtýsson and Samrat Bhattacharjee. Control on demand: Customizing control for each application. ATT Labs Technical Memorandum, 1997.

[58] Gísli Hjálmtýsson and Samrat Bhattacharjee. Control on Demand. In *Proceedings of the First International Working Conference on Active Networks (IWAN)*, June 1999.

[59] Gísli Hjálmtýsson and Samrat Bhattacharjee. Control on Demand: An efficient approach to router programmability. *IEEE Journal on Selected Areas in Communications (JSAC): Special Issue on Service Enabling Platforms for Networked Multimedia Systems*, August 1999. Guest Editors–D. Hutchison, G. Pacifici, B. Plattner, R. Stadler and J. Sventek.

[60] Gísli Hjálmtýsson and Robert Gray. Dynamic C++ Classes: A Lightweight Mechanism to Update Code in a Running System. In *Proceedings of USENIX'98*, June 1998.

[61] Norman C. Hutchinson and Larry L. Peterson. Design of the x-kernel. In *Proceedings of the SIGCOMM '88 Symposium*, pages 65–75, Stanford, California, 16–19 August 1988. ACM Press.

[62] ITU-T. ISO DIS 10918-1 Digital compression and coding of continuous-tone still images (JPEG). Recommendation T.81.

[63] ITU-T. Video Codec for audiovisual services at p*64kb/s. Recommendation H.261.

[64] ITU-T. Specification of Signaling System Number 7. ITU Recommendation Q.709, 1993.

[65] V. Jacobson. Congestion avoidance and control. In *ACM SIGCOMM '88*, 1988.

[66] R. jain. Congestion Control in Computer Networks: Trends and Issues. *IEEE Network*, pages 24–30, May 1990.

[67] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, volume 31,5 of *Operating Systems Review*, pages 52–65, New York, October 5–8 1997. ACM Press.

[68] C. Kent and J. Mogul. Fragmentation considered harmful. In *ACM SIGCOMM '87*, August 1987.

[69] A. A. Lazar. Programming Telecommunication Networks. *IEEE Network Magazine*, September/October 1997.

[70] Ulana Legedza, David J. Wetherall, and John Guttag. Improving the performance of distributed applications using active networks. IEEE Infocom'98, 1998.

[71] W. Leland, M. Taqqu, W. Willinger, and D. Wilson. On the self-similar nature of Ethernet traffic (extended version). *IEEE/ACM Transactions on Networking*, pages 1–15, February 1994.

[72] Xue Li, Sanjoy Paul, and Mostafa H. Ammar. Layered Video Multicast with Retransmission (LVMR): Evaluation of hierarchical rate control. In *Proceedings of IEEE Infocom'98*, San Francisco, CA, March 1998.

[73] N. Likhanov, B. Tsybakov, and N. Georganas. Analysis of an ATM buffer with self-similar (fractal) input traffic. In *IEEE Infocom '95*, 1995.

[74] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.

[75] A. Lopez-Ortiz and Daniel M. German. A multicollaborative push-caching http protocol for the WWW. In *World Wide Web Conference (WWW5)*, 1995. Poster Presentation.

[76] S. McCanne, V. Jacobson, and M. Vetterli. Receiver-driven layered multicast. In *ACM Sigcomm '96*, pages 117–130, 1996.

[77] S. Merugu, S. Bhattacharjee, E. Zegura, and K. Calvert. Bowman: A Node OS for Active Networks. Submitted to INFOCOM'2000.

[78] A. Montz, D. Mosberger, S. O'Malley, L. Peterson, T. Proebsting, and J. Hartman. Scout: A communications-oriented operationg system. Technical Report 94-20, Department of Computer Science, The University of Arizona, 1994.

[79] James E. Narem Jr. An informal semantics of C-Linda V2.3.5. Technical report, Department of Computer Science, Yale University, New Haven, Conn., 20 December 1989.

[80] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of the Second Symposium on Operating System Design and Implementation*, Seattle, Wa., October 1996.

[81] Craig Partridge. *Gigabit Networking*. Addison-Wesley, Reading, MA, USA, 1994.

[82] G. Phillips, B. Braden, J. Kann, and B. Lindell. ASP EE: The ASP Execution Environment's Protocol Programming Interface (PPI). Available at `http://www.isi.edu/active-signal/ARP/`.

[83] G.P. Picco, G.C. Roman, and P. J. McCann. Expressing Code Mobility in Mobile UNITY. In M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC'97)*, Lecture Notes in Computer Science 1301. Springer-Verlag, September 1997.

[84] J. Postel. RFC 791: Internet Protocol, September 1981. Obsoletes RFC0760. See also STD0005 . Status: STANDARD.

[85] James M. Purtilo. The POLYLITH software bus. *ACM Transactions on Programming Languages and Systems*, 16(1):149–174, January 1994.

[86] K. Ramakrishnan and R. Jain. Congestion Avoidance in Computer Networks with A Connecitonless Network Layer:Part IV:a Binary Feedback Scheme for Congestion Avoidance in Computer Networks with Connectionless Network Layer. Technical report, Digital Equipment Corporation, August 1987. DEC Technical Report DEC-TR-510.

[87] K. Ramakrishnan and R. Jain. A Binary Feedback Scheme for Congestion Avoidance in Computer Networks with Connectionless Network Layer. In *Proceedings of SIGCOMM'88.*, August 1988.

[88] K. K. Ramakrishnan and S. Floyd. A proposal to add explicit congestion notification (ECN) to IPv6 and to TCP. IETF Internet Draft:draft-kksjf-ecn-00.txt, November 1997.

[89] Dennis M. Ritchie. A Stream Input-Output system. *AT&T Bell Laboratories Technical Journal*, 63(8), October 1984.

[90] A. Romanow and S. Floyd. Dynamics of TCP traffic over ATM networks. *IEEE Journal on Selected Areas in Communications*, May 1995.

[91] D. Rubenstein, S. Kasera., D. Towsley, and J. Kurose. Improving reliable multicast using active parity encoding services (apes). Technical Report CMPSCI Technical Report RT 98-79, University of Massachussetts, July 1998.

[92] Jef Rulifson. DEL. Network Working Group, RFC 5, 2 June 1969.

[93] H. Saltzer, D. P. Reed, and D. Clark. End-to-end arguments in system design. *ACM Transactions on Computing Systems*, 2(4), 1984.

[94] Steve Schwab. Personal communication via electronic mail. TIS, May 1999.

[95] Beverly Schwartz, Wenyi Zhou, Alden Jackson, W. Timothy Strayer, Dennis Rockwell, and Craig Partridge. Smart Packets for Active Networks. BBN Technologies, http://www.net-tech.bbn.com/smtpkts/smart.ps.gz, 1998.

[96] Jonathan Smith, David Farber, Carl A. Gunter, Scott Nettles, Mark Segal, William D. Sincoskie, David Feldmeier, and Scott Alexander. SwitchWare: Towards a 21st century network infrastructure. Whitepaper.

[97] William Stallings. *Cryptography and network security: principles and practice*. Prentice-Hall, Inc., Upper Saddle River, NJ 07458, USA, second edition, 1999.

[98] D. L. Tennenhouse and D. J. Wetherall. Towards an active network architecture. In *Multimedia Computing and Networking '96*, January 1996.

[99] D. Towsley, J. Kurose, and S. Pingali. A Comparison of Sender-Initiated and Receiver-Initiated Reliable Multicast Protocols. *IEEE JSAC*, 15(3), April 1997.

[100] J. E. van der Merwe, S. Rooney, I. M. Leslie, and S. A. Crosby. The Tempest - A Practical Framework for Network Programmability. *IEEE Network Magazine*, 12(3), 1998.

[101] V. Varadharajan. Verification of network security protocols. *Computers & Security*, 8:693–708, 1989.

[102] D. Wessels and K. Claffy. Internet Cache Protocol (ICP), version 2. Technical report, IETF Network Working Group, May 27 May 1997. draft-wessels-icp-v2-03.txt.

[103] D. Wetherall, J. Guttag, and D. L. Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. In *IEEE OPENARCH'98*, San Francisco, CA, April 1998.

[104] W. Willinger, M. Taqqu, R. Sherman, and D. Wilson. Self-similarity through high-variability: statistical analysis of Ethernet LAN traffic at the source level. In *ACM Sigcomm '95*, pages 100–113, 1995.

[105] L. Wu, R. Sharma, and B. Smith. Thin streams: an architecture for multicasting layered video. In *NOSSDAV'97*, 1997.

[106] Y. Yemini and S. da Silva. Towards programmable networks. In *IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, L'Aquila, Italy, October 1996.

[107] Jens Zander. SOFTNET — Packet Radio in Sweden. In *ARRL 1st Computer Networking Conference*, October 1981.

Vita

Samrat Bhattacharjee was born in Calcutta, India on 26th January, 1972. In 1994, he graduated *Summa Cum Laude* from Georgia College and State University with Bachelor's of Science degrees in Computer Science and Mathematics. He received a Doctor of Philosophy degree in Computer Science from the Georgia Institute of Technology in 1999.