Internet Algorithmics — How to Build Fast Servers and Routers

George Varghese

April 3, 2004
For Aju and Tim and Andrew, who made all this possible …
Contents

1 Introduction ......................................................... 18
  1.1 The Problem: Network Bottlenecks ......................... 20
    1.1.1 Endnode Bottlenecks ................................. 20
    1.1.2 Router Bottlenecks ................................. 22
  1.2 The Techniques: Internet Algorithmics .................... 24
    1.2.1 Warm-up Example: Scenting an Evil Packet ............ 25
    1.2.2 Strawman Solution .................................. 26
    1.2.3 Thinking Algorithmically ............................ 28
    1.2.4 Refining the Algorithm: Exploiting Hardware ......... 29
    1.2.5 Cleaning up ........................................ 30
    1.2.6 Characteristics of Internet Algorithmics .......... 32
  1.3 Book Organization ........................................... 34
  1.4 Exercise .................................................... 36

2 Models ............................................................ 37
  2.1 Protocols .................................................... 38
    2.1.1 Transport and Routing Protocols .................... 38
    2.1.2 Abstract Protocol Model ............................ 39
    2.1.3 Performance Environment and Measures ............ 41
  2.2 Hardware ..................................................... 43
2.2.1 Combinatorial Logic ........................................... 44
2.2.2 Timing and Power ........................................... 45
2.2.3 Raising the Abstraction Level of Hardware Design ........ 46
2.2.4 Memories .................................................. 49
2.2.5 Memory Subsystem Design Techniques ..................... 54
2.2.6 Component Level Design ................................... 54
2.2.7 Final Hardware Lessons .................................... 55
2.3 Network Device Architectures .................................. 57
  2.3.1 Endnode Architecture ..................................... 57
  2.3.2 Router Architecture ....................................... 60
2.4 Operating Systems ............................................. 65
  2.4.1 Uninterrupted Computation via Processes ................ 66
  2.4.2 Infinite Memory via Virtual Memory ..................... 68
  2.4.3 Simple I/O via System Calls ............................. 70
2.5 Summary ..................................................... 71
2.6 Exercises ................................................... 72

3 Strategies ......................................................... 78
  3.1 A Coin Puzzle ............................................... 78
  3.2 The Ternary CAM Update Puzzle ............................. 80
  3.3 Algorithms versus Algorithmics Redux ...................... 83
  3.4 Fifteen Implementation Principles .......................... 85
    3.4.1 Systems Principles .................................... 85
    3.4.2 Principles for Modularity with Efficiency ............. 91
    3.4.3 Principles for Speeding Up Routines ................... 94
  3.5 Design versus Implementation Principles ................... 96
  3.6 Caveats ................................................... 97
    3.6.1 Case Study 1, Reducing Page Download Times ........ 97
5.2.4 Transparantly Emulating Copy Semantics .......................... 162
5.3 Avoiding Copying using Remote DMA ................................. 165
  5.3.1 Avoiding Copying in a Cluster .................................. 165
  5.3.2 Modern Day Incarnations of RDMA ............................... 167
5.4 Broadening to File Systems ........................................... 169
  5.4.1 Shared memory ................................................... 170
  5.4.2 IO-Lite: A unified view of buffering ............................. 171
  5.4.3 Avoiding File System Copies by I/O Splicing ................... 173
5.5 Broadening beyond Copies ............................................. 174
5.6 Broadening beyond Data Manipulations ............................... 176
  5.6.1 Using Caches Effectively ....................................... 177
  5.6.2 DMA versus PIO ................................................. 182
5.7 Conclusions .......................................................... 182
5.8 Exercises ........................................................... 184

6 Control ................................................................. 187
  6.1 Why Control Overhead ............................................... 188
  6.2 Avoiding Scheduling Overhead in Networking Code ............... 191
    6.2.1 Making User Level Protocol Implementations Real .......... 193
  6.3 Avoiding Context Switching Overhead in Applications .......... 196
    6.3.1 Process per Client ........................................... 197
    6.3.2 Thread per Client ............................................ 198
    6.3.3 Event Driven Scheduler ..................................... 199
    6.3.4 Event driven server with helper processes .................. 200
    6.3.5 Task Based Structuring ...................................... 202
  6.4 Fast Select ........................................................ 203
    6.4.1 A Server Mystery ............................................... 203
    6.4.2 Existing use and implementation of select() ............... 205
6.4.3 Analysis of Select() .......................................................... 207
6.4.4 Speeding Up Select without changing the API ................... 209
6.4.5 Speeding Up Select by changing the API ......................... 210
6.5 Avoiding System Calls ..................................................... 212
   6.5.1 The VIA proposal ..................................................... 216
6.6 Reducing Interrupts ......................................................... 217
   6.6.1 Avoiding Receiver Livelock ........................................ 218
6.7 Conclusions ................................................................. 219
6.8 Exercises ................................................................. 220

7 Timers .............................................................................. 224
   7.1 Why Timers ................................................................. 224
   7.2 Model and Performance Measures .................................... 226
   7.3 Simplest Timer Schemes ............................................... 227
   7.4 Timing Wheels ............................................................. 229
   7.5 Hashed Wheels ............................................................. 231
   7.6 Hierarchical Wheels ..................................................... 233
   7.7 BSD Implementation ................................................... 235
   7.8 Obtaining Fine Granularity Timers .................................. 236
   7.9 Conclusions ............................................................... 238
   7.10 Exercises ................................................................. 239

8 Demultiplexing .................................................................. 241
   8.1 Opportunities and Challenges of Early Demultiplexing ......... 243
   8.2 Goals ........................................................................... 244
   8.3 CSPF: Pioneering Packet Filters ..................................... 245
   8.4 BPF: Enabling High Performance Monitoring .................... 246
   8.5 Path Finder: Factoring out Common Checks ...................... 248
8.6 DPF: Compilers to the Rescue ........................................ 253
8.7 Conclusions ............................................................. 257
8.8 Exercises .............................................................. 258

9 Protocol Processing .................................................... 259
  9.1 Buffer Management .................................................. 260
     9.1.1 Buffer Allocation ............................................... 261
     9.1.2 Sharing Buffers ............................................... 264
  9.2 CRCs and Checksums ............................................... 266
     9.2.1 CRCs ............................................................ 267
     9.2.2 Internet Checksums ......................................... 271
     9.2.3 Finessing Checksums ....................................... 273
  9.3 Generic Protocol Processing ..................................... 274
     9.3.1 UDP Processing ............................................... 278
  9.4 Reassembly .......................................................... 279
     9.4.1 Efficient Reassembly ....................................... 280
  9.5 Conclusions .......................................................... 283
  9.6 Exercises ............................................................ 285

10 Exact Matching ......................................................... 289
  10.1 Challenge 1: Ethernet Under Fire ............................... 290
  10.2 Challenge 2: Wire-Speed Forwarding ............................ 293
  10.3 Challenge 3: Scaling Lookups to Higher Speeds ............... 297
     10.3.1 Scaling via Hashing ....................................... 298
     10.3.2 Using Hardware Parallelism ............................... 300
  10.4 Summary ............................................................ 302
11 Prefix Matching

11.1 Introduction to Prefix Lookups .................................................. 305
   11.1.1 Prefix Notation ............................................................... 305
   11.1.2 Why variable length prefixes? ............................................. 306
   11.1.3 Lookup Model ................................................................. 307
11.2 Finessing Lookups ................................................................. 310
   11.2.1 Threaded Indices and Tag Switching .................................... 310
   11.2.2 Flow Switching ............................................................... 312
   11.2.3 Status of Tag Switching, Flow Switching, and MPLS ................. 313
11.3 Non-algorithmic Techniques for Prefix Matching .............................. 314
11.4 Unibit Tries .............................................................................. 315
11.5 Multibit Tries ........................................................................... 318
   11.5.1 Fixed Stride Tries ............................................................. 320
   11.5.2 Variable Stride Tries .......................................................... 321
   11.5.3 Incremental Update ........................................................... 323
11.6 LC Tries ................................................................................... 324
11.7 Lulea Compressed Tries ............................................................... 326
11.8 Tree Bitmap .............................................................................. 330
11.9 Binary Search on Ranges ............................................................. 333
11.10 Binary Search on Prefix Lengths ................................................. 335
11.11 Memory Allocation in Compressed Schemes ................................... 338
11.12 Lookup Chip Model .................................................................. 340
11.13 Conclusions ............................................................................ 342
11.14 Exercises ............................................................................... 343
12 Classification

12.1 Why Packet Classification ................................................. 350
12.2 Packet Classification Problem ............................................ 353
12.3 Requirements and Metrics .................................................. 355
12.4 Simple Solutions ............................................................ 356
12.5 Two Dimensional Schemes .................................................... 358
   12.5.1 Fast Searching using Set Pruning Trees .......................... 359
   12.5.2 Reducing Memory using Backtracking ............................ 361
   12.5.3 The best of both worlds: Grid-of-Tries ......................... 363
12.6 Approaches to General Rule Sets ......................................... 365
   12.6.1 Geometric View of Classification ................................ 366
   12.6.2 Beyond Two Dimensions: The Bad News ......................... 367
   12.6.3 Beyond Two Dimensions: The Good News ......................... 368
12.7 Extending Two Dimensional schemes .................................... 369
12.8 Using Divide-and-Conquer .................................................. 371
12.9 Bit Vector Linear Search ................................................... 372
12.10 Cross-Producing ............................................................ 375
12.11 Equivalence cross-producing ............................................. 377
12.12 Decision Tree Approaches ................................................. 380
12.13 Conclusions ............................................................... 384
12.14 Exercises ................................................................. 385

13 Switching .............................................................................. 388

13.1 Shared Memory Switches .................................................... 391
13.2 Router History: from Busses to Crossbars ............................. 392
13.3 The Take-a-Ticket Crossbar scheduler ................................. 393
13.4 Head of Line Blocking ........................................................ 398
13.5 Avoiding HOL Blocking by Output Queuing ........................... 400
13.6 Avoiding HOL Blocking using Parallel Iterative Matching .......................... 402
13.7 Avoiding randomization with iSLIP .......................................................... 405
  13.7.1 Extending iSLIP to Multicast and Priority ...................................... 410
  13.7.2 iSLIP Implementation notes .......................................................... 412
13.8 Scaling to Larger Switches ................................................................. 414
  13.8.1 Measuring Switch Cost .............................................................. 415
  13.8.2 Clos Networks for Medium Sized Routers ................................... 416
  13.8.3 Benes Networks for Larger Routers ............................................ 421
13.9 Scaling to Faster Switches ................................................................. 429
  13.9.1 Using Bit Slicing for Higher Speed Fabrics .................................... 429
  13.9.2 Using Short links for Higher Speed Fabrics .................................. 430
  13.9.3 Memory Scaling using Randomization .......................................... 432
13.10 Conclusions ......................................................................................... 433
13.11 Exercises ............................................................................................. 435

14 Scheduling 438

14.1 Motivation and Chapter Outline ......................................................... 440
14.2 Random Early Detection ..................................................................... 441
14.3 Token Bucket Policing ......................................................................... 444
14.4 Multiple Outbound Queues and Priority .......................................... 446
14.5 A Quick Detour into Reservation Protocols ....................................... 447
14.6 Providing Bandwidth Guarantees ....................................................... 448
  14.6.1 The Parochial Parcel Service ....................................................... 448
  14.6.2 Deficit Round Robin .................................................................... 450
  14.6.3 DRR Implementation and Extensions .......................................... 453
14.7 Schedulers that Provide Delay Guarantees ....................................... 455
14.8 Scalable Fair Queuing ......................................................................... 460
14.9 Summary .............................................................................................. 463
14.10 Exercises ............................................................................................. 464
15 Routers as Distributed Systems \hspace{1cm} 466

15.1 Internal Flow Control \hspace{1cm} 467
   15.1.1 Improving Performance \hspace{1cm} 468
   15.1.2 Rescuing Reliability \hspace{1cm} 470

15.2 Internal Striping \hspace{1cm} 473
   15.2.1 Improving Performance \hspace{1cm} 473
   15.2.2 Rescuing Reliability \hspace{1cm} 474

15.3 Asynchronous Updates \hspace{1cm} 477
   15.3.1 Improving Performance \hspace{1cm} 477
   15.3.2 Rescuing Reliability \hspace{1cm} 479

15.4 Conclusions \hspace{1cm} 480
15.5 Exercises \hspace{1cm} 481

16 Measurement \hspace{1cm} 494

16.1 Why measurement is hard \hspace{1cm} 495
   16.1.1 Why counting is hard \hspace{1cm} 496

16.2 Chapter Outline \hspace{1cm} 497

16.3 Reducing SRAM width using DRAM backing store \hspace{1cm} 498

16.4 Reducing counter width using randomized counting \hspace{1cm} 501

16.5 Reducing counters using Threshold Aggregation \hspace{1cm} 501

16.6 Reducing counters using Flow Counting \hspace{1cm} 504

16.7 Reducing processing using Sampled NetFlow \hspace{1cm} 505

16.8 Reducing reporting using Sampled Charging \hspace{1cm} 507

16.9 Correlating measurements using Trajectory Sampling \hspace{1cm} 508

16.10A Concerted Approach to Accounting \hspace{1cm} 509

16.11 Computing Traffic Matrices \hspace{1cm} 511

16.12 Sting as an example of Passive Measurement \hspace{1cm} 514

16.13 Conclusion \hspace{1cm} 515

16.14 Exercises \hspace{1cm} 517

11
17 Security

17.1 Chapter Outline .............................................. 520
17.2 Searching for Multiple Strings in Packet Payloads .......................... 520
  17.2.1 Integrated String Matching using Aho-Corasick ...................... 521
  17.2.2 Integrated String Matching using Boyer-Moore ...................... 523
17.3 Approximate String Matching ........................................ 525
17.4 IP Traceback via Probabilistic Marking ................................. 527
17.5 IP Traceback via Logging ........................................... 531
  17.5.1 Bloom Filters ............................................. 532
  17.5.2 Bloom Filter Implementation of Packet Logging ...................... 534
17.6 Detecting Worms ................................................. 535
17.7 Conclusion ....................................................... 537
17.8 Exercises ......................................................... 539

18 Endgame

18.1 What this book was about ......................................... 541
  18.1.1 Endnode Algorithmics ........................................ 541
  18.1.2 Router Algorithmics ........................................ 543
  18.1.3 Towards a synthesis ....................................... 544
18.2 What Internet Algorithmics is about ................................ 548
  18.2.1 Interdisciplinary Thinking ................................... 548
  18.2.2 Systems Thinking .......................................... 549
  18.2.3 Algorithmic Thinking ...................................... 551
18.3 Internet Algorithmics and Real Products ................................ 554
18.4 The future of Internet Algorithmics ................................ 555
  18.4.1 New abstractions .......................................... 555
  18.4.2 New connecting Disciplines .................................. 557
  18.4.3 New requirements .......................................... 558
18.5 Conclusions ....................................................... 558

540
A Detailed Models

A.1 TCP and IP ................................................................. 560
A.1.1 Transport Protocols ............................................... 560
A.1.2 Routing Protocols .................................................. 563
A.2 Hardware Models ....................................................... 565
A.2.1 From Transistors to Logic Gates ............................... 565
A.2.2 Timing Delays ....................................................... 566
A.2.3 Hardware Design Building Blocks ............................. 568
A.2.4 Memories: The Inside Scoop .................................... 568
A.2.5 Chip Design .......................................................... 570
PREFACE

I first encountered Internet Algorithmics in the spare and beautiful design of the first bridge done by Mark Kempf, whose story is told in Chapter 10. To prove itself against detractors, the bridge had to be wire speed; this forced as much careful thinking of the implementation as of the Layer 2 switching concept itself. Later, I encountered it again when DEC decided to build a wire speed OSI Router and when DEC pursued a dream of building a Gigabit speed switched network. This last effort was before its time but did produce the DEC Gigaswitch described in Chapter 13.

Later when I joined academia and began working with industry there was a groundswell of activity during the boom of the 1990's as companies rushed to build a new router. It never ceased to amaze me that each of these companies got funded and bought out for solving essentially the same problems: building a fast internal switch, a fast lookup engine, and some fast QoS processing, and writing or porting OSPF and BGP.

As 2000 rolled out, suddenly even endnode networking, long the domain of a few academic researchers, suddenly became mainstream with labels such as Infiniband, VIA, and RDMA. Of course, this is because of the new impetus to build fast servers. The techniques used are basically old ones (e.g., copy avoidance, avoiding system calls) with a few new twists.

Thus it seemed ripe to collect all these methods in a book. However, my mind revolted at the thought of stuffing together a number of tricks into a grab bag. So at the 1997 SIGCOMM tutorial, I began trying to break up the techniques more finely and systematically via a set of 15 principles. In retrospect, some of the principles seem redundant and glib. Perhaps, however, they will serve as part of a first attempt to organize a vast amount of material.

Somewhat later, I worked on a router that worked at OC-768 speeds: the team consisted of some crackerjack digital and circuit designers who had designed processors, experts who had written vast amounts of the software in core routers, and people like me who liked algorithms. Despite the varied backgrounds and ignorance of each other’s areas, the team produced some wonderful ideas by working together. In particular, I had a moment of epiphany working with a memory designer to design a memory system that made a new algorithm possible.

That experience convinced me that interdisciplinary thinking was vital. Thus I decided after much trepidation to produce the Models chapter. Being hardly a hardware expert myself, I found that I had picked up enough over the years to ask leading questions and work with hardware designers. I thought this, however crude, was worth passing on. Probably the attempt to teach algorithm designers the salient aspects of hardware, hardware designers the fundamentals of protocols and operating systems, will all be regarded as a failure. But it seems like a worthwhile effort if it at least it provides a call for further action.
After all, if RISC computing was made possible by interdisciplinary collaboration between architects and compiler writers, one can only hope for what a new generation of students — students fortified by a solid dose of common sense, systems-savvy and a mastery of protocols, hardware, and algorithms — might do to upend the future of networking . . .

Finally, a special thanks to my editors: Karen and Mary and Emily; to all my advisors who taught me so much: Wushow Chou, Arne Nilsen, Baruch Awerbuch, Nancy Lynch; to all my mentors at work: Alan Kirby, Radia Perlman, Tony Lauck, Bob Thomas, Bob Simcoe, Jon Turner; to numerous colleagues at DEC and other companies, especially to Sharad Merhotra, Bill Lynch, and Tony Li of Procket Networks who taught me so much about real routers; to my students who冒险ed in the field of Internet Algorithmics with me; to numerous reviewers of this book and especially to Craig Partridge and Jon Snader; to my family, my mother, my in-laws and my sister; and, of course, to my wife Aju and sons Tim and Andrew. A publisher had, long ago, given me a T-shirt imprinted with a design for the cover of this book. It was meant to motivate but became an albatross in my closet, and a long standing family joke. Perhaps I can wear it now.

I'd like to end this preface by acknowledging my heroes: four teachers who have influenced me. The first is Leonard Bernstein who taught me in his lectures on music that the teacher's enthusiasm for the material can be infectious. The second is George Polya who taught me in his books on problem solving that the process of discovery is as important as the final discoveries themselves. The third is Socrates who taught me through Plato that it is worth always questioning assumptions. The fourth is Jesus who has taught me that life, and indeed this book, is not a matter of merit but of grace and gift.

La Jolla
August 2002
Part I: The Rules of the Game

“Come, Watson, come!”, he cried. “The game is afoot!”

— Arthur Conan Doyle in “The Abbey Grange”

The first part of the book deals with specifying the rules of the Internet Algorithmics game. We start with a quick introduction where we define Internet Algorithmics and contrast it to algorithm design. Next, we present models of protocols, operating systems, processor architecture, and hardware design; these are the key disciplines used in the rest of the book. Then we present a set of fifteen principles that are abstracted from the specific techniques presented later in the book. This part of the book ends with a set of sample problems together with solutions obtained using the principles. Implementors pressed for time should skim the Quick Reference Guides at the start of each chapter.
Biography

The first rules of Internet Algorithmics were articulated by Dave Clark in the 1980’s. Clark hammered home the distinction between protocol specifications and protocol implementations in a series of classic documents. He proposed upcalls to implement protocols, a notion that seemed radical at the time but is commonplace today. Besides protocol implementations, of course, Clark is celebrated for being one of the architects of the TCP/IP protocols. Clark works at the Laboratory for Computer Science at MIT.
Chapter 1

Introduction

*What really makes it an invention is that someone decides not to change the solution to a known problem, but to change the question.*

— Dean Kamen.

Just as the objective of chess is to checkmate the opponent and the objective of tennis is to win matches, the objective of the Internet Algorithmics game is to battle networking implementation bottlenecks.

This book was written to answer a need for books on efficient network protocol implementations. The vast majority of networking books are on network protocols; even the implementation books are, for the most part, more detailed explanations of the protocol. While protocols form the foundation of the field, there are just a handful of important protocols left such as TCP, IP, Fiberchannel, and HTTP. On the other hand, there are vast numbers of implementations as most companies and startups “roll their own” to gain competitive advantage. This is exacerbated by the tendency to implement protocols everywhere from bridges to SAN switches to toasters.

Thus it should be clear that there are vastly more people implementing than designing protocols. *This is a textbook for implementors and networking students, covering ground from the art of building a fast server to building a fast router.* To do so, this book describes a collection of efficient implementation techniques; in fact, an initial section of each chapter contains a Quick Reference Guide for implementors that points to the most useful techniques for each topic.

However, the book goes much further and distills a fundamental way of crafting solutions to internet bottlenecks that we call *Internet Algorithmics.* This provides the reader tools to
design different implementations for specific contexts, and to deal with new bottlenecks that will undoubtedly arise in the changing world of networks.

Some new endnode bottlenecks that have emerged in recent years are as follows. TCP Offload engines are a popular way to offload servers, but pose the fresh challenge of implementing TCP in hardware at 10 Gbps and higher speeds. XML is rapidly becoming the lingua franca of the web, and XML processing is known to considerably slow down servers. Web services are being proposed as a way to generalize web pages to offer services, and the protocols that underly web services appear to be cumbersome to implement.

Some new networking device bottlenecks that have emerged in recent years are as follows. First, it seems clear that routers are going beyond being simple forwarding engines and will soon provide more sophisticated services for measurement and security that are hard to implement at high speeds. Second, the increasing virulence and prevalence of network attacks have led to a growing need for security devices for intrusion detection and prevention; designing real-time intrusion detection devices for 1 Gbps and higher speeds is a challenge. Finally, there are a number of newer proxies and gateways (e.g., Voice-over-IP gateways) that are inherently difficult to implement at high speeds.

Thus while this book concentrates on basic tasks involved in server and router processing, many of these tasks (e.g., avoiding control overhead, lookups etc.) apply to other networking devices and applications. More fundamentally, Internet Algorithmics teaches a way of thinking that should be useful for an implementor of any networking device in software or hardware, ranging from intrusion detection devices to network appliances to blade servers.

So what is Internet Algorithmics? Internet algorithmics goes beyond the design of efficient algorithms for networking tasks, though this has an important place. In particular, Internet Algorithmics recognizes the primary importance of taking an interdisciplinary systems approach to streamlining network implementations.

Internet Algorithmics is an interdisciplinary approach because it encompasses such fields as architecture and operating systems (for speeding up servers), hardware design (for speeding up network devices such as routers), and algorithm design (for designing scalable algorithms). Internet Algorithmics is also a systems approach because it is described in this book using a set of 15 principles that exploit the fact that routers and servers are systems, in which efficiencies can be gained by moving functions in time and space between subsystems.

Thus in summary this book is primarily about two things: a set of fundamental networking performance bottlenecks, and a set of fundamental techniques to address these bottlenecks. Next, we provide a quick preview of both the bottlenecks and the methods covered in the rest of the book.
1.1 The Problem: Network Bottlenecks

The main problem considered in this book is how to make networks easy to use while at the same time realizing the performance of the raw hardware. Ease of use comes from the use of powerful network abstractions such as socket interfaces and prefix-based forwarding. Unfortunately, without such abstractions exact a large performance penalty when compared to the capacity of raw transmission links such as optical fiber. To examine this performance gap in more detail we examine two fundamental categories of networking devices, endnodes and routers.

1.1.1 Endnode Bottlenecks

Endnodes are the endpoints of the network: they include personal computers and workstations, as well as large servers that provide services. Endnodes are specialized towards computation as opposed to networking, and are designed to support general-purpose computation. Thus endnode bottlenecks are the result of two forces: structure and scale.

- **Structure:** To be able to run arbitrary code, personal computers and large servers typically have an operating system that mediates between applications and the hardware. To ease software development, most large operating systems are carefully structured as layered software; to protect the Operating System from other applications, operating systems implement a set of protection mechanisms; finally, core operating systems routines such as schedulers and allocators are written using general mechanisms that target as wide a class of applications as possible. Unfortunately, the combination of layered software, protection mechanisms, and excessive generality can slow down networking software greatly even with the fastest processors.

- **Scale:** The emergence of large servers providing web and other services causes further performance problems. In particular, a large server such as a web server will typically have thousands of concurrent clients. Many operating systems use inefficient data structures and algorithms that were designed for an era when the number of connections was small.

Figure 1.1 previews the main endnode bottlenecks covered in this book together with causes and solutions. The first bottleneck occurs because conventional operation system structures cause packet data copying across protection domains; the situation is further complicated in web servers by similar copying with respect to the file system, and by other manipulations such
<table>
<thead>
<tr>
<th>Bottleneck</th>
<th>Chapter</th>
<th>Cause</th>
<th>Sample Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copying</td>
<td>5</td>
<td>Protection, structure</td>
<td>Copying many data blocks without OS Intervention (e.g., RDMA)</td>
</tr>
<tr>
<td>Context Switching</td>
<td>6</td>
<td>Complex Scheduling</td>
<td>User-level protocol implementations Event driven web servers</td>
</tr>
<tr>
<td>System calls</td>
<td>6</td>
<td>Protection, structure</td>
<td>Direct channels from applications to Drivers (e.g., VIA)</td>
</tr>
<tr>
<td>Demuxing</td>
<td>7</td>
<td>Scaling with # of endpoints</td>
<td>BPF and PathFinder</td>
</tr>
<tr>
<td>Timers</td>
<td>8</td>
<td>Scaling with # of timers</td>
<td>Timing wheels</td>
</tr>
<tr>
<td>Checksums/CRCs</td>
<td>9</td>
<td>Generality</td>
<td>Multibit computation</td>
</tr>
<tr>
<td>Protocol code</td>
<td>9</td>
<td>Generality</td>
<td>Header Prediction</td>
</tr>
</tbody>
</table>

Figure 1.1: Preview of Endnode bottlenecks, solutions to which are described in Part II of the book

as checksums that examine all the packet data. Chapter 5 describes a number of techniques to reduce these overheads while preserving the goals of system abstractions such as protection and structure. The second major overhead is the control overhead caused by switching between threads of control (or protection domains) while processing a packet, and is addressed in Chapter 6.

Networking applications use timers to deal with failure: with a large number of connections the timer overhead at a server can become large; this overhead is addressed in Chapter 7. Similarly, network messages must be demultiplexed (i.e., steered) on receipt to the right end application; techniques to address this bottleneck are addressed in Chapter 8. Finally, there are several other common protocol processing tasks such as buffer allocation and checksums that are addressed in Chapter 9.

As we said earlier, however, new abstractions such as XML and Web Services are emerging. While we do not directly address these bottlenecks in this book, many of the techniques we discuss for reducing data manipulation and control overhead apply directly to these new contexts. And, of course, the method of thinking that underlies Internet Algorithmics can be used to approach these new bottlenecks.
1.1.2 Router Bottlenecks

While we concentrate on Internet routers, many of the techniques described apply equally well to any networking device such as bridges, switches, and gateways for IP as well as for other routing protocols such as Fiberchannel and Infiniband. Thus through the rest of the book, it is often useful to think of a “router” as a “generic networking device”.

Unlike endnodes, such networking devices are special-purpose devices devoted to networking. Thus there is little structural overhead within a router, with only the use of a lightweight operating system and a clearly separated forwarding path that is often completely implemented in hardware. Instead of structure, the fundamental problems faced by routers are caused by scale and services.

- **Scale:** Network devices face two areas of scaling: *bandwidth scaling* and *population scaling*. Bandwidth scaling occurs because optical links keep getting faster as the progress from 1 Gbps to 40 Gbps links shows, and because Internet traffic keeps growing due to a diverse set of new applications. Population scaling occurs because more endpoints get added to the Internet as more enterprises go online.

- **Services:** The need for speed and scale drove much of the networking industry in the 1980’s and 1990’s as more businesses went online (e.g., Amazon), and whole new online services (e.g., Ebay) were created. But the very success of the Internet boom requires careful attention in the next decade to make the Internet more effective by providing guarantees in terms of performance, security, and reliability. After all, if manufacturers (e.g., Dell) sell more online than by other channels, it is important to provide network guarantees — delay in times of congestion, protection during attacks, and availability when failures occur. Finding ways to implement these new services at high speeds will be a major challenge for router vendors in the next decade.

Figure 1.2 previews the main router (bridge/gateway) bottlenecks covered in this book together with causes and solutions.

First, all networking devices forward packets to their destination by looking up a forwarding table. The simplest forwarding table lookup does an exact match with a destination address, and is exemplified by bridges. Chapter 10 describes fast and scalable exact match lookup schemes. Unfortunately, population scaling has made lookups far more complex for routers. To deal with with large Internet populations, routers keep a single entry called a prefix (analogous to a telephone area code) for a large group of stations. Thus routers must do a more complex
<table>
<thead>
<tr>
<th>Bottleneck</th>
<th>Chapter</th>
<th>Cause</th>
<th>Sample Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exact Lookups</td>
<td>10</td>
<td>Link speed scaling</td>
<td>parallel hashing</td>
</tr>
<tr>
<td>Prefix Lookups</td>
<td>11</td>
<td>link speed scaling, Prefix database size scaling</td>
<td>Compressed multibit tries</td>
</tr>
<tr>
<td>Packet Classification</td>
<td>12</td>
<td>Service differentiation, Link speed and size scaling</td>
<td>Decision tree algorithms, Hardware parallelism (CAMs)</td>
</tr>
<tr>
<td>Switching</td>
<td>13</td>
<td>Optical-electronic speed gap, Head of line blocking</td>
<td>Crossbar switches, Virtual Output Queues</td>
</tr>
<tr>
<td>Fair Queueing</td>
<td>14</td>
<td>Service differentiation, Link speed scaling, Memory scaling</td>
<td>Weighted fair queuing, Deficit Round Robin, DiffServ, Core Stateless</td>
</tr>
<tr>
<td>Internal Bandwidth</td>
<td>15</td>
<td>Scaling of internal bus speeds</td>
<td>Reliable striping</td>
</tr>
<tr>
<td>Measurement</td>
<td>16</td>
<td>Link speed scaling</td>
<td>Juniper’s DCU</td>
</tr>
<tr>
<td>Security</td>
<td>17</td>
<td>Scaling in number and intensity of attacks</td>
<td>Traceback with Bloom Filters, Extracting worm signatures</td>
</tr>
</tbody>
</table>

Figure 1.2: Preview of Router bottlenecks, solutions to which are described in Parts III and IV of the book

The *longest prefix match* lookup. Chapter 11 describes solutions to this problem that scale to increasing speeds and table sizes.

Many routers today offer what is sometimes called *service differentiation* where different packets can be treated differently in order to provide service and security guarantees. Unfortunately, this requires an even more complex form of lookup called *packet classification* in which the lookup is based on the destination, source, and even the services that a packet is providing. This challenging issue is tackled in Chapter 12.

Next, all networking devices can be abstractly considered as switches that shunt packets coming in from a set of input links to a set of output links. Thus a fundamental issue is that of building a high speed switch. This is hard especially in the face of the growing gap between optical and electronic speeds. The standard solution is to use parallelism via a *crossbar switch*. Unfortunately, it is non-trivial to schedule a crossbar at high speeds, and parallelism is limited by a phenomenon known as Head-of-line blocking. Worse, population scaling and optical multiplexing are forcing switch vendors to build ports with a large number of ports (e.g., 256), which exacerbates these other problems. Solutions to these problems are described
in Chapter 13.

While the previous bottlenecks are caused by scaling, the next bottleneck is caused by the need for new services. The issue of providing performance guarantees at high speeds is treated in Chapter 14, where the issue of implementing so-called QoS (Quality of Service) mechanisms is studied. Chapter 15 briefly surveys another bottleneck that is becoming increasingly a problem: the issue of bandwidth within a router. It describes sample techniques such as striping across internal busses or chip-to-chip links.

Finally, the last sections of the book take a brief look at emerging services that must, we believe, be part of a well engineered Internet of the future. First, routers of the future must build in support for measurement, because measurement is key to engineering networks to provide guarantees. While routers today provide some support for measurement in terms of counters and NetFlow records, Chapter 16 also considers more innovative measurement mechanisms that may be implemented in the future.

Finally, Chapter 17 describes security support, some of which is already being built into routers. Given the increased sophistication, virulence, and rate of network attacks, we believe that implementing security features in networking devices (whether routers or dedicated intrusion prevention/detection devices) will be essential. Further, unless the security device can keep up with high speed links, the device may miss vital information required to spot an attack.

1.2 The Techniques: Internet Algorithmics

During the course of this book, we will, as the Walrus might have said to the Carpenter, talk of many specific techniques: of interrupts, copies, and timing wheels, of Pathfinder and Sting; of why some routers are very slow, and whether servers can scale. But what underlies the assorted techniques in this book and makes it more than a recipe book is the notion of Internet Algorithmics. Internet Algorithmics recognizes the primary importance of taking a systems approach to streamlining network implementations.

While everyone recognizes that the Internet is a system consisting of routers and links, it is perhaps less obvious that every networking device from the Cisco GSR to an Apache Web Server is also a system. A system is built out of interconnected subsystems that are instantiated at various points in time. For example, a core router consists of line cards with forwarding engines and packet memories connected by a crossbar switch. The router behavior is affected by decisions at various time scales that range from manufacturing time (when default parameters
are stored in NVRAM) to route computation time (when routers conspire to compute routes) to packet forwarding time (when packets are sent to adjoining routers).

Thus one key observation in the systems approach is that one can often design an efficient subsystem by moving some of its functions in space (i.e., to other subsystems) or in time (i.e., to points in time before or after the function is apparently required). In some sense, the practitioner of Internet Algorithmics is an unscrupulous opportunist willing to change the rules at any time to make the game easier. The only constraint is that the functions provided by the overall system continue to satisfy users.

For example, the Connecticut Yankee in King Arthur’s court used a gun to deal with dueling knights who were accustomed to jousting with lances. This was unfair — but it was effective.

Considering the constraints faced by the network implementor at high speeds — increasingly complex tasks, larger systems to support, small amounts of high speed memory, and a small number of memory accesses — it may require every trick, every gun in one’s arsenal, to keep pace with the increasing speed and scale of the Internet. The designer can throw hardware at the problem, can change the system assumptions, design a new algorithm — whatever it takes to get the job done.

This book divides into four parts. The first part, of which this is the first chapter, lays a foundation for applying Internet Algorithmics to packet processing. The second chapter of the first part outlines models, and the third chapter presents general principles used in the remainder of the book.

One of the best ways to get a quick idea about what Internet Algorithmics is about is to plunge right away into a warm-up example. While the warm-up example that follows is in the context of a device within the network where new hardware can be designed, note that a major part of this book (Part 2) is about building efficient servers using only software design techniques.

1.2.1 Warm-up Example: Scenting an Evil Packet

So imagine a front-end network monitor (or intrusion detection system) on the periphery of a corporate network that wishes to flag suspicious incoming packets — packets that could contain attacks on internal computers. A common such attack is a buffer overflow attack where the attacker places machine code $C$ in a network header field $F$.

If the receiving computer allocates a buffer too small for header field $F$, and is careless about checking for overflow, the code $C$ can spill on to the receiving machine’s stack. With a
little more effort, the intruder can make the receiving machine actually execute evil code $C$. $C$ then takes over the receiver machine.

Clearly, the best way to deal with such attacks, is to fix endnode software — possibly automatically — to detect and eliminate such bugs. Since this goal has proved elusive for so many years, the approach of detecting such intrusions at the entrance to a network is still popular.

![Diagram of Intrusion Detection System and Evil Code](image)

**Figure 1.3:** Getting wind of an evil packet by noticing the frequency of unprintable characters.

Figure 1.3 shows such an attack embodied in a familiar field, a destination web URL (for Uniform Resource Locator). How might the monitor detect the presence of such a suspicious URL? A possible way is to observe that URLs containing evil code are often too long (an easy check), and often contain a large fraction of unusual (at least in URLs) characters such as #. Thus the monitor could mark such packets (containing URLs that are too long, and contain too many occurrences of such unusual characters) for more thorough examination.

It is worth stating at the outset that the security implications of this strategy need to be carefully thought out. For example, there may be several innocuous programs, such as CGI scripts, in URLs that lead to false positives. Without getting too hung up in overall architectural implications, let us assume that this was a specification handed down to a chip architect by a security architect. We now use this toy problem, suggested by Mike Fisk, to illustrate algorithmics in action.

Faced with such a specification, a chip designer may use the following design process that illustrates some of the principles of Internet Algorithmics. The process starts with a strawman design, and refines the design using techniques like designing a better algorithm, relaxing the specification, and exploiting hardware.

### 1.2.2 Strawman Solution

The overall length check is straightforward to implement, so we concentrate on checking for a prevalence of suspicious characters. The first strawman solution illustrated in Figure 1.4. The chip maintains two arrays, $T$ and $C$, with 256 elements each, one for each possible value
of a 8-bit character. The threshold array $T$ contains the acceptable percentage (as a fraction of the entire URL length) for each character. If the occurrences of a character in an actual URL fall above this fraction, the packet should be flagged. Each character can have a different threshold.

<table>
<thead>
<tr>
<th>Threshold Array</th>
<th>Count Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>#</td>
<td>3</td>
</tr>
<tr>
<td>255</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1.4: Strawman solution for detecting evil packet by counting occurrences of each character via a count array (middle) and then comparing in a final pass with an array of acceptable thresholds (left).

The count array $C$ in the middle contains the current count $C[i]$ for each possible character $i$. When the chip reads a new character "$i" in the URL, it increments $C[i]$ by 1. $C[i]$ is initialized to 0 for all values of $i$ when a new packet is encountered. The incrementing process only starts after the chip parses the HTTP header and recognizes the start of a URL.

In the HTTP protocol, the end of a URL is signified by two newline characters; thus one can only tell the length of the URL after parsing the entire URL string. Thus, after the end of the URL is encountered, the chip does a final pass over the array $C$. If $C[j] \geq L \cdot T[j]$ for any $j$, where $L$ is the length of the URL, the packet is flagged.

Assume that packets are coming into the monitor at high speed, and that we wish to finish processing a packet before the next one arrives. This requirement, called wire speed processing, is very common in networking; it prevents processing backlogs even in the worst case. To meet wire speed requirements, ideally the chip should do a small constant number of operations for every URL byte. Assume the main step of incrementing a counter can be done in the time to receive a byte.

Unfortunately, the two passes over the array, first to initialize it and second to check for threshold violations, make this design slow. Minimum packet sizes are often as small as 40 bytes and only include network headers. Adding 768 more operations (1 write and 1 read to each element of $C$, and 1 read of $T$ for each of 256 indices) can make this design infeasible.
1.2.3 Thinking Algorithmically

Intuitively, the second pass through the arrays \( C \) and \( T \) at the end seems like a waste. For example, it suffices to alarm if any character is over the threshold. So why check all characters? This suggests keeping track only of the largest character count \( c \) at the end perhaps the algorithm only needs to check whether \( c \) is over threshold with respect to the total URL length \( L \).

This does not quite work. A non-suspicious character like 'e' may well have a very high occurrence count. However, 'e' is also likely to be specified with a high threshold. Thus if we keep track only of 'e' with say a count of 20, we may not keep track of '#' with say a count of 10. If the threshold of '#' is much smaller, the algorithm may cause a false negative: the chip may fail to alarm on a packet which should be flagged.

The counterexample suggests the following fix. The chip keeps track in a register of the highest counter relativized to the threshold value. More precisely, the chip keeps track of the highest relativized counter \( Max \) corresponding to some character \( k \), such that \( C[k]/T[k] = Max \) is the highest among all characters encountered so far. If a new character \( i \) is read, the chip increments \( C[i] \). If \( C[i]/T[i] > Max \), then the chip replaces the current stored value of \( Max \) with \( C[i]/T[i] \). At the end of URL processing, the chip alarms if \( Max \geq L \).

The reason this works is as follows. If \( Max = C[k]/T[k] \geq L \), clearly the packet must be flagged because character \( k \) is over threshold. On the other hand, if \( C[k]/T[k] < L \), then for any character \( i \), it follows that \( C[i]/T[i] \leq C[k]/T[k] < L \), Thus if \( Max \) falls below threshold, then no character is above threshold. Thus there can be no false negatives. This solution is shown in Figure 1.5

![Figure 1.5: Avoiding the final loop through the threshold array by keeping track only of Max, the highest counter encountered so far relative to its threshold value.](image-url)

28
1.2.4 Refining the Algorithm: Exploiting Hardware

The new algorithm has eliminated the loop at the end but still has to deal with a divide operation while processing each byte. Divide logic is somewhat complicated and worth avoiding if possible — but how?

Returning to the specification and its intended use, it seems likely that thresholds are not meant to be exact floating point numbers. It is unlikely that the architect providing thresholds can estimate the values precisely; one is likely to approximate 2.78% as 3% without causing much difference to the security goals. So why not go further and approximate the threshold by some power of 2 less than the exact intended threshold? Thus if the threshold is 1/29, why not approximate it as 1/32?

Changing the specification in this way requires negotiation with the system architect. Assume that the architect agrees to this new proposal. Then a threshold like 1/32 can be encoded compactly as the corresponding power of 2 — i.e., 5. This threshold shift value can be stored in the threshold array instead of a fraction.

Thus when a character $j$ is encountered, the chip increments $C[j]$ as usual and then shifts $C[j]$ to the left — dividing by $1/x$ is the same as multiplying by $x$ — by the specified threshold. If the shifted value is higher than the last stored value for $Max$, the chip replaces the old value with the new value and marches on.

Thus the logic required to implement the processing of a byte is a simple shift and compare. The stored state is only a single register to store $Max$. As it stands, however, the design requires a read to the threshold array (to read the shift value), a read to the count array (to read the old count) and a write to the count array (to write back the incremented value).

Now reads to memory — 1-2 nsec even for the fastest on-chip memories but possibly even as slow as 10 nsec for slower memories — are slower than logic. Single gate delays are only in the order of picoseconds, and shift logic does not require too many gate delays. Thus the processing bottleneck is the number of memory accesses.

The chip implementation can combine the 2 Reads to memory into 1 Read by coalescing the Count and Threshold arrays into a single array as shown in Figure 1.6. The idea is to make the memory words wide enough to hold the counter (say 15 bits to handle packets of length 32K) and the threshold (depending on the precision necessary, no more than 14 bits).

Thus the two fields can easily be combined into a larger word of size 29 bits. In practice, hardware can handle much larger words sizes of up to 1000 bits. Also note that extracting the 2 fields packed into a single words, quite a chore in software, is trivial in hardware by routing wires appropriately between registers or by using multiplexers.
1.2.5 Cleaning up

There is one thorny issue postponed to this point. The terminal loop has been eliminated while leaving the initial initialization loop. To handle this, note that the chip has spare time for initialization after parsing the URL of the current packet, and before encountering the URL of the next packet.

Unfortunately, packets can be as small as 50 bytes, even with an HTTP header. Thus even assuming a slack of 40 non-URL bytes other than 10 bytes of the URL, this still does not suffice to initialize a 256 byte array without paying $256/40 = 6$ more operations per byte than during the processing of a URL. As in the URL processing loop, each initialization step requires a Read and Write of some element of the coalesced array.

A trick among lazy people is to postpone work until it is absolutely needed, in the hope that it may never be needed. Note that, strictly speaking, the chip need not initialize an $C[i]$ until character $i$ is accessed for the first time in a subsequent packet. But how can the chip tell that it is seeing character $i$ for the first time?

To implement lazy evaluation, each memory word representing an entry in the coalesced array must be expanded to include, say, a 3 bit generation number $G[i]$. The generation number can be thought of as a value of clock time measured in terms of packets encountered so far, except that it is limited to 3 bits. Thus, the chip keeps an additional register $g$, besides the extra $G[i]$ for each $i$, that is 3 bits long; $g$ is incremented mod 8 for every packet encountered. In addition, every time $C[i]$ is updated, the chip updates $G[i]$ as well to reflect the current value of $g$.

Given the generation numbers, the chip need not initialize the count array after the current
packet has been processed. However, consider the case of a packet whose generation number is $h$ that contains a character $i$ in its URL. When the chip encounters $i$ while processing packet $h$, the chip reads $C[i]$ and $G[i]$ from the count array. If $G[i] < h$, this clearly indicates that entry $i$ was last accessed by an earlier packet and has not been subsequently initialized. Thus the logic will write back the value of $C[i]$ as 1 (initialization plus increment) and set $G[i]$ to $h$. This is shown in Figure 1.7.

<table>
<thead>
<tr>
<th>Shift Count Gen</th>
<th>CurrentGen = 101</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 bit</td>
<td>1 bit</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4 bits</td>
<td>3</td>
</tr>
<tr>
<td>255</td>
<td></td>
</tr>
<tr>
<td>Evil Code</td>
<td>GET AIM://OVERFLOW # # ! # . . # . *</td>
</tr>
<tr>
<td>1) Read wide word 2) If Gen match, Write count + 1 else Write 1</td>
<td></td>
</tr>
<tr>
<td>3) If C[i] shifted by T[i] &gt; Max, replace Max</td>
<td></td>
</tr>
</tbody>
</table>

At the end, flag packet if Max > URL Length

Figure 1.7: The final solution with generation numbers to finesse an initialization loop

The careful reader will immediately object. Since the generation number is only 3 bits, once the value of $g$ wraps around, there can be aliasing. Thus if $G[i]$ is 5, and entry $i$ is not accessed until 8 more packets have gone by, $g$ will have wrapped around to 5. If the next packet contains $i$, $C[i]$ will not be initialized and the count will (wrongly) accumulate the count of $i$ in the current packet together with the count that occurred 8 packets in the past.

The chip can avoid such aliasing by doing a separate “scrubbing” loop which reads the array and initializes all counters with outdated generation numbers. For correctness, the chip must guarantee one complete scan through the array for every 8 packets processed. Given that one has a slack of (say) 40 non-URL bytes per packet, this guarantees a slack of 320 non-URL bytes after 8 packets, which suffices to initialize a 256 element array using one Read and one Write per byte, whether the byte is a URL or a non-URL byte. Clearly, the designer can gain more slack, if needed, by increasing the bits in the generation number, at the cost of slightly increased storage in the array.

The chip, then, must have two states: one for processing URL bytes, and one for processing non-URL bytes. When the URL is completely processed, the chip switches to the “Scrub” state. The chip maintains another register that points to the next array entry $s$ to be scrubbed. In
the scrub state, when a non-URL character is received, the chip reads entry $s$ in the coalesced array. If $G[s] \leq g$, $G[s]$ is reset to $g$ and $C[s]$ is initialized to 0.

Thus the use of 3 extra bits of generation number per array entry has reduced initialization processing cycles, trading processing for storage. Altogether a coalesced array entry is now only 32 bits, 15 bits for a counter, 14 bits for a threshold shift value, and 3 bits for a generation number. Note that the added initialization check needed during URL byte processing does not increase memory references (the bottleneck) but adds slightly to the processing logic. In addition, it requires two more chip registers to hold $g$ and $s$, a small additional expense.

1.2.6 Characteristics of Internet Algorithmics

The example of scenting an evil packet illustrates three important aspects of Internet Algorithmics.

a) Internet Algorithmics is Interdisciplinary: Given the high rates at which network processing must be done, a router designer would be hard pressed not to use hardware. The example exploited several features of hardware: it assumed wide words of arbitrary size were easily possible; it assumed that shifts were easier than divides; it assumed that memory references were the bottleneck; it assumed that a 256 element array contained in fast on-chip memory was feasible; it assumed that adding a few extra registers was feasible; and finally, it assumed that small changes to the logic to combine URL processing and initialization were trivial to implement.

For the reader unfamiliar with hardware design, this is a little like jumping into a game of cards without knowing the rules, and then finding oneself finessed and trumped in unexpected ways. A contention of this book is that mastery of a few relevant aspects of hardware design can help even a software designer understand at least the feasibility of different hardware designs. A further contention of this book is that such interdisciplinary thinking can help produce the best designs.

Thus Chapter 2 presents the rules of the game. It presents simple models of hardware that point out opportunities for finessing and trumping troublesome implementation issues. The chapter also presents simple models of Operating Systems. This is done because end systems such as clients and web servers require tinkering with and understanding Operating System issues to improve performance, just as routers and network devices require tinkering with hardware.

b) Internet Algorithmics recognizes the primacy of Systems Thinking: The specification was relaxed to allow approximate thresholds in powers of two, which simplified
the hardware. Relaxing specifications and moving work from one subsystem to another is an extremely common systems technique but is not encouraged by current educational practice in universities in which each area is taught in isolation.

Thus today one has separate courses in each of algorithms, operating systems, and networking. This tends to encourage "black box" instead of holistic or systems thinking. The example alluded to other systems techniques such as the use of lazy evaluation and trading memory for processing in order to scrub the count array.

Thus a feature of this book is an attempt to distill the systems principles used in algorithms into a set of 15 principles that are cataloged on the front cover of the book and are explored in detail in Chapter 3. This book attempts to explain and dissect all the network implementations described in this book in terms of these principles. The principles are also given numbers for easy reference, though for the most part we will use both the number and the name. For instance, to take a quick peek at the front cover, relaxing specifications is principle P4 and lazy evaluation is P2a.

c) Internet Algorithmics can benefit from Algorithmic Thinking: While this book stresses the primacy of systems thinking to finesse problems wherever possible, there are many situations where systems constraints prevent elimination of problems. In our example, after attempting to finesse the need for algorithmic thinking by relaxing the specification, the problem of false positives led to considering keeping track of the highest counter relative to its threshold value. As a second example, Chapter 11 shows that despite attempts to finesse Internet lookups using what is called tag switching, many routers resort to efficient algorithms for lookup.

It is worth emphasizing, however, that because the models are somewhat different from standard theoretical models, it is often insufficient to blindly reuse existing algorithms. For example, Chapter 13 discusses how the need to schedule a crossbar switch in 8 nsec leads to considering simpler maximal matching heuristics, as opposed to more complicated algorithms that produce optimal matchings in a bipartite graph.

As a second example, Chapter 11 describes how the BSD implementation of lookups blindly reused a data structure called a Patricia trie, which uses a skip count, to do IP lookups. The resulting algorithm requires complex backtracking.\footnote{The algorithm was considered to be the state of the art for many years, and was even implemented in hardware in several router designs. In fact, a patent for lookups issued to a major router company appears to be a hardware implementation of BSD Patricia tries with backtracking. Any deficiencies of the algorithm can, of course, be mitigated by fast hardware. However, it is worth considering that a simple change to the algorithm could have simplified the hardware design.} A simple modification that keeps the
actual bits that were skipped, (instead of the count) avoids the need for backtracking. But this requires some insight into the black box (i.e., the algorithm) and its application.

In summary, the uncritical use of standard algorithms can miss implementation breakthroughs because of inappropriate measures (e.g., for packet filters such as BPF, the insertion of a new classifier can afford to take more time than search), inappropriate models (e.g., ignoring the effects of cache lines in software, or parallelism in hardware), and inappropriate analysis (e.g., order of complexity results that hide constant factors crucial in ensuring wire speed forwarding).

Thus another purpose of this book is to persuade implementors that insight into algorithms and the use of fundamental algorithmic techniques such as divide-and-conquer and randomization is important to master. This leads us to the following

**Definition:** Internet Algorithmics is the use of an interdisciplinary systems approach, seasoned with algorithmic thinking, to design fast implementations of network processing tasks at servers, routers, and other networking devices.

While this book concentrates on networking, the general algorithmics approach holds for the implementation of any computer system, whether it be a database, a processor architecture, or a software application. This general philosophy is alluded to in Chapter 3 by providing illustrative examples from the field of computer system implementation. The reader only interested in networking should rest assured that the remainder of the book, other than Chapter 3, avoids further digressions beyond networking.

### 1.3 Book Organization

This book is structured into four overall parts. Each part is made as self-contained as possible to allow a detailed study of each part. Unfortunately, this length can come in the way of a reader who is in a hurry. Readers that are pressed for time can consult the index or table of contents for a particular topic (e.g., IP lookups). More importantly, in the first two pages of each chapter, there is also a *Quick Reference Guide* that lists some of the most important topics, especially for implementors. In a first reading, the Quick Reference Guide may be fastest guide to usefully skimming a chapter.

The book starts with the proposed method of thought. Thus Part I, of which this chapter is the leader, aims to familiarize the reader with the rules and philosophy of the Internet Algorithmics "game". A quick overview of Part I is described in Figure 1.8.

Chapter 2 describes simple models of protocols, operating systems, hardware design and endnode and router architectures. Chapter 3 describes in detail the 15 principles used as a
<table>
<thead>
<tr>
<th>Focus</th>
<th>Chapter</th>
<th>Motivation</th>
<th>Sample Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Models</td>
<td>2</td>
<td><strong>Understand simple models</strong> for OS, hardware, networks</td>
<td><strong>Memory technology techniques</strong> (interleaving, mixing SRAM/DRAM)</td>
</tr>
<tr>
<td>Strategies</td>
<td>3</td>
<td><strong>Assimilate systems principles</strong> for overcoming bottlenecks</td>
<td><strong>Pass hints, evaluate lazily</strong> Add state, exploit locality</td>
</tr>
<tr>
<td>Problems</td>
<td>4</td>
<td><strong>Practice applying principles on simple problems</strong></td>
<td><strong>Designing a lookup engine for a network monitor</strong></td>
</tr>
</tbody>
</table>

Figure 1.8: Preview of Internet Algorithmics. Internet Algorithmics is introduced using a set of models, strategies, and sample problems, which are described in Part I of the book.

cornerstone for the book. Chapter 4 rounds out the first part by providing 15 examples, drawn for the most part from real implementation problems, to allow the reader a first opportunity to apply algorithmics on real problems.

Part II of the book is called “Playing with Endnodes” and shows how to build fast endnode implementations such as Web Servers. It describes how to avoid copies, reduce control overhead, and to quickly process specific functions such as sequence number bookkeeping and checksums. The second part of the book requires more attention to operating systems techniques and less to hardware models. An overview of Part II has already been described in Figure 1.1.

Part III of the book is called “Playing with Routers” and shows how to build fast routers, bridges, and gateways. It covers such topics as IP lookups, packet classification, switching, and implementing Quality of Service (QoS) in routers. Increasingly, the design of fast routers requires attention to hardware design opportunities.

Finally, the book ends with Part IV, which is called “Playing with the Future”. Part IV starts by speculating on the potential need for implementing more complex tasks in the future. Examples of future needs include providing support for accounting and detecting security violations. Part IV is applicable to future routers, as well as to high speed monitoring and intrusion detection devices. The book ends with a short chapter that attempts to reach closure by distilling the unities that underly the many different topics in this book. The last chapter also briefly describes examples of the use of algorithmics in a canonical router (the Cisco GSR) and a canonical server (the Flash Web Server).

An overview of Parts III and IV has already been described in Figure 1.1.

While Parts II and III provide specific techniques for important specific problems, the main
goal of this book is to allow the reader to be able to tackle arbitrary packet processing tasks at high speeds in software or hardware.

Thus if you, dear reader, are assigned the task of speeding up XML processing in a web server (likely, given current trends) or even the task of computing the Chi-square statistic in a router (somewhat far out, but just possible because Chi-square provides a test for detecting abnormal observed frequencies), the hope is that you will blink for a few minutes . . . . but then set to work crafting a solution using the models, principles, and techniques described in this book.

1.4 Exercise

1. Implementing Chi-square The Chi-Square statistic can be used to find if the overall set of observed character frequencies are unusually different (as compared to normal random variation) from the expected character frequencies. This is a more sophisticated test, statistically speaking, than the simple threshold detector used in the warm-up example. Assume that the thresholds represent the expected frequencies. The statistic is computed by finding the sum of \((\text{ExpectedFrequency}[i] - \text{ObservedFrequency}[i])^2 / \text{ExpectedFrequency}[i]\) for all values of character \(i\). The chip should alarm if the final statistic is above a specified threshold. (For example, a value of 14.2 implies that there is only a 1.4% chance that the difference is due to chance variation.) Find a way to efficiently implement this statistic assuming once again that the length is known only at the end.
Chapter 2

Models

A rather small set of key concepts is enough. Only by learning the essence of each topic, and by carrying along the least amount of mental baggage at each step, will the student emerge with a good overall understanding of the subject.

— Carver Mead and Lynn Conway

To improve the performance of endnodes and routers, an implementor must know the rules of the game. A central difficulty is that network algorithmics encompasses four separate areas: protocols, hardware architectures, operating systems, and algorithms. Networking innovations occur when area experts work together to produce synergistic solutions. But can a logic designer understand protocol issues, and can a clever algorithm designer understand hardware tradeoffs, at least without deep study?

Useful dialog can begin with simple models that have explanatory and predictive power but without unnecessary detail. At the least such models should define terms used in the book; at the best, such models should enable a creative person outside an area to play with and and create designs that can be checked by an expert within the area. For example, a hardware chip implementor should be able to suggest software changes to the chip driver, and a theoretical computer scientist should be able to dream up hardware matching algorithms for switch arbitration. This is the goal of this chapter.

To this end, this chapter is organized as follows. Starting with a model for protocols in Section 2.1, the implementation environment is described in bottom-up order. Section 2.2 describes relevant aspects of hardware protocol implementation, surveying logic, memories, and components. Section 2.3 describes a model for endnodes and network devices such as routers. Section 2.4 describes a model for the relevant aspects of operating systems that affect
Chapter 8

Demultiplexing

*Biologically the species is the accumulation of the experiments of all its successful individuals since the beginning.*

— H. G. Wells

A protocol, like a copy center or an ice cream parlor, should be able to serve multiple clients. The clients of a protocol could be end users (as in the case of the file transfer protocol), software programs (for example, when the tool traceroute uses the Internet protocol), or even other protocols (as in the case of the email protocol SMTP that uses TCP).

Thus when a message arrives, the receiving protocol must dispatch the received message to the appropriate client. This function is called *demultiplexing.* Demultiplexing is an integral part of Data Link, Routing, and Transport protocols. It is a fundamental part of the abstract protocol model of Chapter 2.

Traditionally, demultiplexing is done layer by layer using a demultiplexing field contained in each layer header of the received message. This is called *layered demultiplexing,* and is shown in Figure 8.1. For example, working from bottom to top in the picture, a packet may arrive on the Ethernet at a workstation. The packet is examined by the Ethernet driver, which looks at a so-called *protocol type field* to decide what routing protocol (e.g., IP, IPX) is being used. Assuming the type field specifies IP, the Ethernet driver may upcall the IP software.

After IP processing, the IP software inspects the protocol ID field in the IP header to determine the transport protocol (e.g., TCP or UDP?). Assuming it is TCP, the packet will be passed to the TCP software. After doing TCP processing, the TCP software will examine the port numbers in the packet to demultiplex the packet to the right client, say to a process implementing HTTP.
Figure 8.1: Traditional layered demultiplexing has each layer demultiplex a packet to the next layer software above using a field in the layer header.

Traditional demultiplexing is fairly straightforward because each layer essentially does an exact match on some field or fields in the layer header. This can be done easily, using say hashing, as we describe in Chapter 10. Of course, the lookup costs add up at each layer.

By contrast, this chapter concentrates on early demultiplexing which is a much more challenging task at high speeds. Referring back to Figure 8.1, early demultiplexing determines the entire path of protocols taken by the received packet in one operation, when the packet first arrives. In the last example, early demultiplexing would determine in one fell swoop that the path of the web packet was Ethernet, IP, TCP, Web. A possibly better term is delayed demultiplexing. However, this book uses the more accepted name of early demultiplexing.

This chapter delineates the reasons for early demultiplexing, and then studies various implementations of early demultiplexing. The implementations range from the pioneering CMU/Stanford packet filter to the commonly used Berkeley Packet Filter to more recent proposals such as Pathfinder and DPF.

Quick Reference Guide: The Berkeley Packet Filter (BPF) is freely available. However, other demultiplexers are more efficient. The implementor who wishes to design a demultiplexing routine should consider PathFinder, described in Section 8.5. While Dynamic Packet Filter (DPF, see Section 8.6) is even faster, many implementors may find the need for dynamic code generation in DPF to be an obstacle.
8.1 Opportunities and Challenges of Early Demultiplexing

Why is early demultiplexing a good idea? The following basic motivations were discussed in Chapter 6.

- **Flexible User Level Implementations:** The original reason for early demultiplexing was to allow flexible user level implementation of protocols without excessive context switching.

- **Efficient User Level Implementations:** As time went on, implementors realized that early demultiplexing could also allow efficient user level implementations by minimizing the number of context switches. The main additional trick was to structure the protocol implementation as a shared library that can be linked to application programs.

However, there are other advantages of early demultiplexing as well such as:

- **Prioritizing Packets:** Early demultiplexing allows important packets to be prioritized and unnecessary ones to be discarded quickly. For example, Chapter 6 shows that the problem of receiver livelock can be mitigated by early demultiplexing of received packets to place packets directly on a per-socket queue. This allows the system to discard messages for slow processes during overload while allowing better behaved processes to continue receiving messages.

More generally, early demultiplexing is crucial in providing quality-of-service guarantees for traffic streams via service differentiation. If all traffic is demultiplexed into a common kernel queue, then important packets can get lost when the shared buffer fills up in periods of overload. Routers today do packet classification (Chapter 12, Chapter 14) for similar reasons. Early demultiplexing allows explicit scheduling of the processing of data flows; scheduling and accounting can be combined to prevent anomalies like priority inversion.

- **Specializing Paths:** Once the path for a packet is known, the code can be specialized to process the packet because the wider context is known. For example, rather than have each layer protocol check for packet lengths, this can be done just once in the spirit of P1, Avoiding Obvious Waste. The philosophy of paths is taken to its logical conclusion in [MP96], an operating system in which paths are first class objects.

- **Fast Dispatching:** This chapter and Chapter 6 have already described an instance of this idea using packet filters and user level protocol implementations. Early demultiplexing avoids per-layer multiplexing costs; more importantly, it avoids the control overhead that can sometimes be incurred in delayed multiplexing.
8.2 Goals

If early demultiplexing is a good idea, is it easy to implement? Early demultiplexing is particularly easy to implement if each packet carries some information in the outermost (e.g., Data Link or Network) header that identifies the final endpoint. This is an example of P14, passing information in layer headers. For example, if the network protocol is a virtual circuit protocol like ATM, the ATM Virtual Circuit Identifier (VCI) can directly identify the final recipient of the packet.

However, protocols like IP do not offer such a convenience. MPLS does offer this convenience but MPLS is generally used only between routers, as described in Chapter 11. Even using a protocol like ATM, the number of available VCIs may be limited. In lieu of a single demultiplexing field, more complex data structures are needed that we call packet filters or packet classifiers.

Such data structures take as input a complete packet header and map the input to an endpoint or path. Intuitively, the endpoint of a packet represents the receiving application process, while the path represents the sequence of protocols that need to be invoked in processing the packet prior to consumption by the endpoint. Before describing how packet filters are built, here are the goals of a good early demultiplexing algorithm.

- Safety: Many early demultiplexing algorithms are implemented in the kernel based on input from user level programs. Each user program P specifies the packets that it wishes to receive. As with Java programs, designers must ensure that incorrect or malicious users cannot affect other users.

- Speed: Since demultiplexing is done in real time, the early demultiplexing code should run quickly, particularly in the case where there is only a single filter specified.

- Composability: If N user programs specify packet filters that describe the packets they expect to receive, the implementation should ideally compose these N individual packet filters into a single composite packet filter. The composite filter should have the property that it is faster to search through the composite filter than to search each of the N individual filters individually, especially for large N.

This chapter takes a mildly biological view, describing a series of packet filter species, with each successive adaptation providing more of the goals than the previous one. Not surprisingly, the earliest species is nearly extinct, though it is noteworthy for its simplicity and historical interest.
8.3 CSPF: Pioneering Packet Filters

The CMU/Stanford (CSPF) packet filter[MRA87] was developed to allow user-level protocol implementations in the Mach operating system. In the CSPF model, application programs provide the kernel with a program describing the packets they wish to receive. The program supplied by $A$ operates on a packet header, and returns $\text{true}$ if the packet should be routed to application $A$. Like the old Texas Instrument calculators, the programming language is a stack based implementation of an expression tree model.

As shown in Figure 8.2, the leaves of the tree represent simple test predicates on packet headers. An example of a test predicate is equality comparison with a fixed value; for example, in Figure 8.2, ETHER.TYPE = ARP represents a check whether the Ethernet type field in the received packet matches the constant value specified for ARP (Address Resolution Protocol) packets. The other nodes in the tree represent boolean operations such as AND and OR.

Thus the left subtree of the expression tree in Figure 8.2 represents any ARP packet sent from source IP address $X$, while the right subtree represents any IP packet sent from source IP address $X$. Since the root represents an OR operation, the overall tree asks for all IP or ARP packets sent by a source $X$. Such an expression could be provided by a debugging tool to the kernel on behalf of a user who wished to examine IP traffic coming from source $X$.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\linewidth]{expression_tree.png}
\caption{The CMU/Stanford (CSPF) packet filter allows applications to provide programs that specify an expression tree representing the packets they wish to receive. The above tree effectively asks for all IP and ARP packets sent by IP source address $X$.}
\end{figure}

While the expression tree model provides a declarative model of a CSPF filter, CSPF filters actually use an imperative stack-based language to describe expression trees. To provide safety, CSPF provides stack instructions of limited power; to bound running times there are no jumps
or looping constructs. Safety is also achieved by checking program loads and stores in real-time to eliminate wild memory references. Thus stack references are monitored to ensure compliance with the stack range, and references to packets are vetted to ensure they stay within the length of the packet being demultiplexed.

8.4 BPF: Enabling High Performance Monitoring

CSPF guarantees security by using instructions of limited power and by doing run-time bounds checking on memory accesses. However, CSPF is not composable and has problems with speed. The next mutation in the design of packet filters occurred with the design of the Berkeley Packet Filter (BPF)[MJ93].

The BPF designers were particularly interested in using BPF as a basis for high-performance network monitoring tools such as TCAP for which speed was crucial. They noted two speed problems with the use of even a single CSPF expression tree of the kind shown in Figure 8.2

- **Architectural Mismatch**: The CSPF stack model was invented for the PDP-11, and hence is a poor match to modern RISC architectures. First, the stack must be simulated at the price of an extra memory reference for each Boolean operation to update the stack pointer. Second, RISC architectures gain efficiency from storing variables in fast registers and doing computation directly from registers.

Thus to gain efficiency in a RISC architecture, as many computations as possible should take place using a register value before it is reused. For instance, in Figure 8.2, the CSPF model will result in two separate loads from memory for each reference to the Ethernet Type field (to check equality with ARP and IP). On modern machines, it would be better to reduce memory references by storing the type field in a register and finishing all comparisons with the type field in one fell swoop.

- **Inefficient Model**: Even ignoring the extra memory references required by CSPF, the expression tree model often results in more operations than are strictly required. For example, in Figure 8.2, notice that the CSPF expression takes 4 comparisons to evaluate all the leaves. However, notice that once we know that the Ethernet Type is equal to ARP (if we are evaluating from left to right), then the extra check for whether the IP source address is equal to X is redundant.

Principle P1 seeks to avoid waste. The main problem is that in the expression tree model there is no way to “remember” packet parse state as the computation progresses. This can be fixed by a new model that builds a state machine.
CSPF had two other minor problems. It could only parse fields at fixed offsets within packet headers; thus it could not be used to access a TCP header encapsulated within an IP header, because this requires first parsing the IP header length field. CSPF also processes headers using only 16 bit fields; this doubles the number of operations required for 32-bit fields such as IP addresses.

The Berkeley Packet Filter (BPF) fixes these problems as follows. First, it replaces the stack based language by a register based language with an indirection operator that can help parse TCP headers. Fields at specified packet offsets are loaded into registers using a command like “LOAD [12]”, which loads the Ethernet type field, which happens to start at an offset of 12 bytes from the start of an Ethernet packet.

BPF can then do comparisons and jumps such as “JUMP_IF_EQUAL ETHERTYPE, IP, TARGET1, TARGET2”. This instruction compares the accumulator register to the IP Ethernet Type Field; if the comparison is true, the program jumps to line number TARGET1, else to TARGET2. BPF allows working in 8, 16, and 32 bit chunks.

More fundamentally, BPF uses a Control Flow Graph model of computation as illustrated in Figure 8.3. This is basically a state machine starting with a root, whose state is updated at each node, following which it transitions to other node states shown as arcs to other nodes. The state machine starts off by checking whether the Ethernet Type Field is that of IP; if true, it need only check whether the IP source field is X to return true. If false, it needs to check whether the Ethernet type field is ARP and whether the ARP source is X. Notice that in the left branch of the state machine we do not check whether the IP source address is X. Thus the worst case number of comparisons is 3 in Figure 8.3 compared to 4 in Figure 8.2.

The Berkeley Packet Filter is used as a basis for a number of tools including the well known Tcqdump tool by which users can obtain a readable transcript of TCP packets flowing on a link. BPF is embedded into the BSD kernel as shown in Figure 8.4.

When a packet arrives on a network link such as an Ethernet, the packet is processed by the appropriate link level driver and is normally passed to the TCP/IP protocol stack for processing. However, if BPF is active, BPF is first called. BPF checks the packet against each currently specified user filter. For each matching filter, BPF copies as many bytes as are specified by the filter to a per-filter buffer. Notice that multiple BPF applications can cause multiple copies of the same packet to be buffered. The figure also shows another common BPF application besides Tcqdump, the Reverse ARP demon (rarpd).

There are two small features of BPF that are also important for high performance. First, BPF filters packets before buffering, which avoids unnecessary waste (P1) when most of the received packets are not wanted by BPF’s applications. The waste is not just memory for
buffers but also the time required to do a copy (Chapter 5).

Second, since packets can arrive very fast and the READ system call is quite slow, BPF allows batch processing (P2c) and allows multiple packets to be returned to the monitoring application in one call. To handle this and yet allow packet boundaries to be distinguished, BPF adds a header to each packet that includes a time stamp and length. Users of TCDUMP do not have to use this interface; instead, TCDUMP offers a more user-friendly interface: interface commands are compiled to BPF instructions.

8.5 Path Finder: Factoring out Common Checks

BPF is a more refined adaptation than CSPF because it increases speed for a single filter. However, every packet must still be compared with each filter in turn. Thus the processing time grows with the number of filters. Fortunately, this is not a problem for typical BPF usage. For example, a typical TCDUMP application may only provide a few filters to BPF.

However, this is not true if early demultiplexing is used to discriminate between a large number of packet streams or paths. In particular, each TCP connection may provide a filter, and the number of concurrent TCP connections in a busy server can be large. The need to deal with this change in environment (user-level networking) led to another successful mutation called Pathfinder[BGP+94]. Pathfinder goes beyond BPF by providing composability. This allows scaling to a large number of users.
Figure 8.4: Packets arriving on a link are sent to both BPF (for potential logging) and the protocol stack (for normal protocol processing). BPF applies all currently specified filters and queues the packet to the appropriate buffer if the filter indicates a match.

To motivate the Pathfinder solution, imagine there are 500 filters, each of which is exactly the same (Ethernet Type Field is IP, IP protocol type is TCP) except that each specifies a different TCP port-pair. Doing each filter sequentially would require comparing the Ethernet Type of the packet 500 times against the (same) IP Ethernet Type field, and comparing the IP Protocol field 500 times against the (same) TCP protocol value. This is wasteful (P1).

Next, comparing the TCP port numbers in the packet to each of the 500 port-pairs specified in each of the 500 filters is not obvious waste. However, this is exactly analogous to linear search for exact matching. This suggests that integrating all the individual filters into a single composite filter can considerably reduce unnecessary comparisons when the number of individual filters is large. Specifically, this can be done using hashing (P15) to perform exact search; this can replace 500 comparisons by a few comparisons.

A data structure for this purpose is shown in Figure 8.5. The basic idea is to superimpose the CFGs for each filter in BPF so that all comparisons on the same field are placed in a single node. Finally, each node is implemented as a hash table containing all comparison values to replace linear search with hashing.

Figure 8.5 shows an example with at least 4 filters, two of which specify TCP packets with destination port numbers 2, 5; for now ignore the dashed line to TCP port 17 which will be used as an example of filter insertion in a moment. Besides the TCP filters, there are one or
more filters that specify ARP packets, and one or more filters that specify packets that use the OSI protocol.

The root node corresponds to the Ethernet type field; the hash table contains values for each possible Ethernet type field value used in the filters. Each node entry has a value and a pointer. Thus the ARP entry points to nodes that further specify what type of ARP packets must be received; the OSI entry does likewise. Finally, the Ethernet type field corresponding to IP points to a node corresponding to the IP protocol field.

In the IP protocol field node, one of the values corresponding to TCP (which has value 6) will point to the TCP node. In the TCP node, there are 3 values pointing to the three possible destination port values of 2 and 5 (recall the 17 has not been inserted yet). When a TCP packet arrives, demultiplexing proceeds as follows.

Search starts at the root and the Ethernet type field is hashed to find a matching value corresponding to IP. The pointer of this value leads to the IP node where the IP protocol type field is hashed to find a matching value corresponding to TCP. The value pointer leads to the TCP node, where the destination port value in the packet is hashed to lead to the final matching filter.
The Pathfinder data structure has a strong family resemblance to a common data structure called a trie that described in Chapter 11. Briefly, a trie is a tree in which each node contains an array of pointers to subtries; each array contains one pointer for each possible value of a fixed character alphabet.

To search the trie for a keyword, the keyword is broken into characters, and the $i$-th character is used to index into the $i$-th node on the path starting with the root. Searching in this way at node $i$ yields a pointer which leads to node $i + 1$ where search continues recursively. One can think of the Pathfinder structure as generalizing a trie by using packet header fields (e.g., Ethernet type field) as the successive characters used for search, and by using hash tables to replace the arrays at each node.

It is well known that tries provide fast insertions of new keys. Given this analogy, it is hardly surprising that Pathfinder has a fast algorithm to insert or delete a filter. For instance, consider inserting a new filter corresponding to TCP Port 17. As in a trie, the insert algorithm starts with a search for the longest matching prefix (Chapter 11) of this new filter.

This longest match corresponds to the path Ethernet Type = IP and IP Protocol = TCP. Since this path has already been created by the other two TCP filters, it need not be replicated. The insertion algorithm only has to add branches (in this case, a single branch) corresponding to the portion of the new filter beyond the longest match. Thus the hash table in the TCP node need only be updated to add a new pointer to the Port 17 filter.

More precisely, the basic atomic unit in Pathfinder is called a cell. A cell specifies a field of bits in a packet header (using an offset, length and a mask), a comparison value, and a pointer. For example, ignoring the pointer, the cell that checks whether the IP protocol field is TCP is (9, 1, 0xff, 6) — the cell specifies that the ninth byte of the IP header should be masked with all 1's and compared to the value 6 which specifies TCP.

Cells of a given user are strung together to form a pattern for that user. Multiple patterns are superimposed to form the Pathfinder trie by not recreating cells that already exist. Finally, multiple cells that specify identical bit fields but different values are coalesced using a hash table.

Besides using hash tables in place of arrays, Pathfinder also goes beyond tries by making each node contain arbitrary code. In effect, Pathfinder recognizes that a trie is a specialized state machine that can be generalized by performing arbitrary operations at each node in the trie. For instance, Pathfinder can handle fragmented packets by allowing loadable cells in addition to the comparison cells described above. This is required because for a fragmented packet only the first fragment specifies the TCP headers; what links the fragments together is a common packet ID described in the first fragment.
Pathfinder handles fragmentation by placing an additional loadable cell (together with the normal IP comparison cell specifying say a source address) that is loaded with the packet ID after the first fragment arrives. A cell is specified as loadable by not specifying the comparison value in a cell.

The loadable cell is not initially part of the Pathfinder trie but is instead an attribute of the IP cells. If the first fragment matches, the loaded cell is inserted into the Pathfinder trie and now matches subsequent fragments based on the newly loaded packet ID. After all fragments have been removed this newly added cell can be removed. Finally, Pathfinder handles the case when the later fragments arrive before the first fragment by postponing their processing till the first fragment arrives.

Although Pathfinder has been described so far as a tree, the data structure can be generalized to a Directed Acyclic Graph (DAG). A DAG allows two different filters to initially follow different paths through the Pathfinder graph and yet come together to share a common path suffix. This can be useful, for instance, when providing a filter for TCP packets for destination port 80 that can be fragmented or unfragmented. While one needs a separate path of cells to specify fragmented and unfragmented IP packets, the two paths can point to a common set of TCP cells.

Finally, Pathfinder also allows the use of OR links that lead from a cell. The idea is that each of the OR links specify a value, and each of the OR links is checked to find a value that matches and then that link is followed.

In order to prioritize packets during periods of congestion as in Chapter 6, the demultiplexing routine must complete in the minimum time it takes to receive a packet. Software implementations of Pathfinder are fast but are typically unable to keep up with line speeds. Fortunately, the Pathfinder state machine can be implemented in hardware to run at line speeds. This is analogous to the way IP lookups using tries can be made to work at line speeds (Chapter 11).

The hardware prototype described in [BGP+94] trades functionality for speed. It works in 16 bit chunks and implements only the most basic cell functions; it does, however, implement fragmentation in hardware. The limited functionality implies that the Pathfinder hardware can only be used as a cache to speed up Pathfinder software that handles the less common cases. A prototype design running at 100 MHz was projected to take 200 nsec to process a 40 byte TCP message, which is sufficient for 1.5 Gbps. The design can be scaled to higher wire speeds using faster clock rates, faster memories, and a pipelined traversal of the state machine.
8.6 DPF: Compilers to the Rescue

The Pathfinder story ends with an appeal to hardware to handle demultiplexing at high speeds. Since it is unlikely that most workstations and PCs today can afford dedicated demultiplexing hardware, it appears that implementors must choose between the flexibility afforded by early demultiplexing and the limited performance of a software classifier. Thus it is hardly surprising that high performance TCP[CJRS89], Active Messages[vCGS92] and RPC[TNML93] implementations use hand-crafted demultiplexing routines.

Dynamic Packet Filter[EK96] (DPF) attempts to have its cake (gain flexibility) and eat it (obtain performance) at the same time. DPF starts with the Pathfinder trie idea. However, it goes on to eliminate indirections and extra checks inherent in cell processing by recompiling the classifier into machine code each time a filter is added or deleted. In effect, DPF produces separate, optimized code for each cell in the trie as opposed to generic, unoptimized code that can parse any cell in the trie.

DPF is based on dynamic code generation technology[Eng96] which allows code to be generated at run-time instead of when the kernel is compiled. DPF is an application of Principle P2, shifting computation in time. Note that by run-time we mean classifier update time and not packet processing time.

This is fortunate because this implies that DPF must be able to recompile code fast enough so as not to slow down a classifier update. For example, it may take milliseconds to set up a connection which in turn requires adding a filter to identify the endpoint in the same time. By contrast, it can take a few microseconds to receive a minimum size packet at Gigabit rates. Despite this leeway, sub-millisecond compile times are still challenging.

To understand why using specialized code per cell helps, it helps to understand two generic causes of cell processing inefficiency in Pathfinder:

- Interpretation Overhead: Pathfinder code is indeed compiled into machine instructions when kernel code is compiled. However, the code does, in some sense, “interpret” a generic Pathfinder cell. To see this, consider a generic Pathfinder cell C that specifies a 4-tuple: offset, length, mask, value. When a packet P arrives, idealized machine code to check whether the cell matches the packet is as follows:

  LOAD R1, C(Offset); (* load offset specified in cell into register R1 *)
  LOAD R2, C(length); (* load length specified in cell into register R1 *)
  LOAD R3, P(R1, R2); (* load packet field specified by offset into R3 *)
LOAD R1, C(mask); (* load mask specified in cell into register R1 *)
AND R3, R1; (* mask packet field as specified in cell *)
LOAD R2, C(value); (* load value specified in cell into register R5 *)
BNE R2, R3; (* branch if masked packet field is not equal to value *)

Notice the extra instructions and extra memory references in Lines 1,2,4,6 that are used to load parameters from a generic cell in order to be used for later comparison.

- **Safety Checking Overhead:** Because packet filters written by users cannot be trusted, all implementations must perform checks to guard against errors. For example, every reference to a packet field must be checked at run time to ensure that it stays within the current packet being demultiplexed. Similarly, references need to be checked in real time for memory alignment; on many machines, a memory reference that is not aligned to a multiple of a word size can cause a trap. After these additional checks, the code fragment shown above is more complicated and contains even more instructions.

By specializing code for each cell, DPF can eliminate these two sources of overhead by exploiting information known when the cell is added to the Pathfinder graph.

- **Exterminating Interpretation Overhead:** Since DPF knows all the cell parameters when the cell is created, DPF can generate code in which the cell parameters are directly encoded into the machine code as immediate operands. For example, the code fragment above to parse a generic Pathfinder cell collapses to the more compact cell-specific code:

LOAD R3, P(offset, length); (* load packet field into R3 *)
AND R3, mask; (* mask packet field using mask in instruction *)
BNE R3, value; (* branch if field not equal to value *)

Notice the extra instructions and (more importantly) extra memory references to load parameters have disappeared because the parameters are directly placed as immediate operands within the instructions.

- **Mitigating Safety Checking Overhead:** Alignment checking can be reduced in the expected case (P11) by inferring at compile time that most references are word-aligned. This can be done by examining the complete filter. If the initial reference is word-aligned, and the current reference (offset plus length of all previous headers) is a multiple of the word length, then the reference is word-aligned. Real-time alignment checks need only be used when the compile time inference fails, for example when indirect loads are performed.
(e.g., a variable size IP header). Similarly, at compile time, the largest offset used in any
cell can be determined and a single check can be placed (before packet processing) to
ensure that the largest offset is within the length of the current packet.

Once one is on to a good thing, it pays to push it for all it is worth. DPF goes on to
exploit compile time knowledge in DPF to perform further optimizations as follows. A first
optimization is to combine small accesses to adjacent fields into a single large access. Other
optimizations are explored in the exercises.

DPF has the following potential disadvantages that are made manageable through careful
design.

- **Recompilation Time:** Recall that when a filter is added to the Pathfinder trie (Figure 8.5),
  only cells that were not present in the original trie need to be created. DPF optimizes
  this expected case (P11) by caching the code for existing cells and copying this code
directly (without recreating them from scratch) to the new classifier code block. New
code must only be emitted for the newly created cells. Similarly, when a new value is
added to a hash table (e.g., new TCP port added in Figure 8.5), unless the hash function
changes, the code is reused and only the hash table is updated.

- **Code Bloat:** One of the standard advantages of interpretation is more compact code.
  Generating specialized code per cell *appears* to create excessive amounts of code especially
  for large numbers of filters. A large code footprint can, in turn, result in degraded
  instruction cache performance. However, a careful examination shows that the number
  of distinct code blocks generated by DPF is only proportional to the number of distinct
  header fields examined by all filters. This should scale much better than the number of
  filters. Consider for example 10,000 simultaneous TCP connections for which DPF may
  only emit 3 specialized code blocks: one for the Ethernet header, one for the IP header,
  and one hash table for the TCP header.

The final performance numbers for DPF are impressive. DPF demultiplexes messages 13-
26 times faster than Pathfinder on a comparable platform[EK96]. The time to add a filter,
however, is only three times slower than Pathfinder. Dynamic code generation accounts for
only 40% of this increased insertion overhead.

In any case, the larger insertion costs appear to be a reasonable cost to pay for faster
demultiplexing. Finally, DPF demultiplexing routines appears to rival or beat hand-crafted
demultiplexing routines; for instance, a DPF routine to demultiplex IP packets takes 18 in-
structions compared to an earlier value reported in [Cla85] of 57 instructions. While the two
implementations were on different machines, the numbers provide some indication of DPF quality.

The final message of DPF is twofold. First, DPF indicates that one can obtain both performance and flexibility. Just as compiler generated code is often faster than hand-crafted code, DPF code appears to make hand-crafted demultiplexing no longer necessary. Second, DPF indicates that hardware support for demultiplexing at line rates may not be necessary. In fact, it may be difficult to allow dynamic code generation on filter creation in a hardware implementation. Software demultiplexing allows cheaper workstations; it also allows demultiplexing code to benefit from processor speed improvements.

<table>
<thead>
<tr>
<th>Technology Changes can Invalidate Design Assumptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>There are several examples of innovations in architecture and operating systems that were discarded after initial use and then returned to be used again. While this may seem like the whims of fashion (“collars are frilled again in 1995”) or reinventing the wheel (“there is nothing new under the sun”), it takes some careful understanding of current technology to know when to dust off an old idea, possibly even in a new guise.</td>
</tr>
<tr>
<td>For example, the core of the telephone network used to send voice calls using analog signals. With the advent of fiber optics and the transistor, much of the core telephone network now transmits voice signals in digital formats using the T1 and SONET hierarchies. However, with the advent of wavelength division multiplexing in optical fiber, there is at least some talk of returning to analog transmission.</td>
</tr>
<tr>
<td>Thus the good system designer must constantly monitor available technology to check whether the system design assumptions have been invalidated. The idea of using dynamic compilation was mentioned by the CSPF designers in [MRA87] but was not considered further. The CSPF designers assumed that tailoring code to specific sets of filters (by recompiling the classifier code whenever a filter was added) was too “complicated”.</td>
</tr>
<tr>
<td>Dynamic compilation at the time of the CSPF design was probably slow and also not portable across systems; the gains at that time would have also been marginal because of other bottlenecks. However, by the time DPF was being designed a number of systems including VCODE[Eng96] had designed fairly fast and portable dynamic compilation infrastructure. The other classifier implementations in DPF’s lineage had also eliminated other bottlenecks, which allowed the benefits of dynamic compilation to stand out more clearly.</td>
</tr>
</tbody>
</table>
8.7 Conclusions

While it may be trite to say that necessity is the mother of invention, it is also often true. New needs drive new innovations; the lack of a need explains why innovations did not occur earlier. The CSPF filter was implemented when the major need was to avoid a process context switch; having achieved that, improved filter performance was only a second-order effect. The BPF filter was implemented when the major need was to implement a few filters very efficiently to enable monitoring tools like TCDUMP to run at close to wire speeds. Having achieved that, scaling to a large number of filters seemed less important.

Pathfinder was implemented to support user-level networking in the x-kernel[HP91], and to allow Scout[MP96] to use paths as a first-class object that could be exploited in many ways. Having found a plausible hardware implementation, perhaps improved software performance seemed less important. DPF was implemented to provide high performance networking together with complete application level flexibility in the context of an extensible operating system[EKO95]. Figure 8.6 presents a summary of the techniques used in this chapter together with the major principles involved.

<table>
<thead>
<tr>
<th>NUM</th>
<th>PRINCIPLE</th>
<th>USED IN</th>
</tr>
</thead>
<tbody>
<tr>
<td>P9</td>
<td>Pass header specifications from user to kernel</td>
<td>CSPF</td>
</tr>
<tr>
<td>P1</td>
<td>Use CFG to avoid unnecessary tests</td>
<td>BPF</td>
</tr>
<tr>
<td>P4c</td>
<td>Use a register based specification language</td>
<td></td>
</tr>
<tr>
<td>P15</td>
<td>Factor common checks using a generalized trie</td>
<td>Pathfinder</td>
</tr>
<tr>
<td>P2</td>
<td>Specialize code when classifier is modified</td>
<td>DPF</td>
</tr>
</tbody>
</table>

Figure 8.6: Principles used in the various demultiplexing techniques discussed in this chapter.

As in the H. G. Wells quote at the start of the chapter, the DPF species does represent the accumulation of the experiments of all its successful individuals. All filter implementations borrow from CSPF the intellectual leap of separating demultiplexing from packet processing together with the notion that application demultiplexing specifications can be safely exported to the kernel. DPF and Pathfinder in turn borrow from BPF the basic notion of exploiting the underlying architecture using a register based, state-machine model. DPF borrows from Pathfinder the notion of using a generalized trie to factor out common checks.

But while evolution no doubt played a part in the algorithmic process for packet filters, perhaps creationism had the larger say …
8.8 Exercises

1. Other uses of Early Demultiplexing: Besides the uses of early demultiplexing described above consider the following other potential uses.

   - Quality of Service: Why might early demultiplexing help offer different qualities of service to different packets in an end system. Give an example.

   - Integrated Later Processing: Integrated Layer Processing (ILP) was studied in Chapter 5. Discuss why early demultiplexing may be needed for ILP.

   - Specializing Code: Once the path of a protocol is known, one can possibly specialize the code for the path just as DPF specializes the code for each node. Give an example of how path information could be exploited to create more efficient code.

2. Further DPF Optimizations: Besides the optimizations described above consider the following other optimizations that DPF exploits.

   - Atom coalescing: It often happens that a node in the DPF tree checks for two smaller field values in the same word. For example, the TCP node may check for a source port value and a destination port value. How can DPF do these checks more efficiently? What crucial assumption does this depend on, and how can DPF validate this assumption?

   - Optimizing Hash Tables: When DPF adds a classifier, it may update the hash table at the node. Unlike PathFinder, the code can be specialized to the specific set of values in each hash table. Explain why this can be used to provide a more efficient implementation for small tables and for collision handling in some cases.
Part III: Playing with Routers

*My work is a game, a very serious game.* — M.C. Escher

The third part of the book deals with *router algorithmics*. This is the application of Internet Algorithmics to building fast routers. However, many of the techniques apply to bridges, gateways, measurement devices, and firewalls. The techniques are mostly applied in a hardware setting, and much of it has to do with processing packets at wire speeds as links get faster. We study exact lookups, prefix lookups, packet classification, switching, and QoS. We also study some other chores within a router such as striping and flow control across chip-to-chip links within a router.
Biography

Pradeep Sindhu began his career at Xerox PARC designing high-speed interconnects for shared memory microprocessors. His interests in networks began with an idea for building a better internal switch for routers. This idea evolved into building a better router and then to Juniper Networks, the only company so far to pose a threat to Cisco Systems. As CTO of Juniper, Sindhu has helped steer Juniper through some key technology transitions, including a recent one to multi-chassis routers that is described in Chapter 13. Router Algorithmics can take you to some interesting places!
Chapter 10

Exact Matching

‘Challenge-and-response’ is a formula describing the free play of forces that provokes new departures in individual and social life. An effective challenge stimulates men to creative action . . .

— Arnold Toynbee

This chapter begins the third part of this book. The seven chapters that follow apply the models and principles of Chapter 2 and Chapter 3 to the problem of building fast routers. Each chapter addresses the efficient implementation of a key router function.

In the simplest model of a router forwarding path, the destination address of a packet is first looked up to determine a destination port; the packet is then switched to the destination port; finally, the packet is scheduled at the destination port to provide QoS guarantees. In addition, modern high performance routers also subject packets to internal striping (to gain throughput) and to internal credit-based flow control (to prevent loss on chip-to-chip links). The chapters are arranged to follow the same order, from lookups to switching to QoS.

Thus the first three chapters concentrate on the surprisingly difficult problem of state lookup in routers. The story begins with the simplest exact match lookups in this chapter, progresses to longest prefix lookups in Chapter 11, and culminates with the most complex classification lookups in Chapter 12.

What is an exact match lookup? Exact match lookups represent the simplest form of database query. Assume a database with a set of tuples; each tuple consists of a unique fixed length key together with some state information. A query specifies a key $K$. The goal is to return the state information associated with the tuple whose key is $K$. 

289
Now, exact match queries are easily implemented using well studied techniques such as binary search and hash tables [CLR90]. However, they are still worth studying in this book for two reasons. First, in the networking context the models and metrics for lookups are different from the usual algorithmic setting. Such differences include the fact that lookups must complete in the time to receive a packet, the use of memory references rather than processing as a measure of speed, and the potential use of hardware speedups. Exact match lookups offer the simplest opportunity to explore these differences. A second reason to study exact match lookups is because they are crucial for an important networking function called bridging\(^1\) that is often integrated within a router.

*Quick Reference Guide:* The implementor interested in fast exact match schemes should consider either parallel hashing techniques inspired by perfect hashing (Section 10.3.1), or pipelined binary search (Section 10.3.2).

Instead of jumping to the most useful algorithms, this chapter will describe an algorithmics topic (exact match lookups) by describing the history of bridges. This is done for one chapter in the book, in the hope of introducing the reader to the process of algorithmics at work in a real product that changed the face of networking. This chapter also describes some of the stimuli that lead to innovation, and introduces some of the people that produced such inventions.

Arnold Toynbee [TC72] describes history using a challenge-response theory in which civilizations either grow or fail in response to a series of challenges. Similarly, the history of bridges can be described as a series of three challenges: Ethernets under fire, wire speed Ethernet forwarding, and scaling to higher speeds. The responses to these challenges led to what is now known as 802.1 Spanning Tree bridges [IEE97].

### 10.1 Challenge 1: Ethernet Under Fire

The first challenge arose in the late 1980's. Ethernet, invented in the 1970's as a low-cost, high bandwidth interconnect for Personal Computers, was attacked as behaving poorly at large loads and being incapable of spanning large distances. Recall that if two or more nodes on an Ethernet send data at the same time, a collision occurs on the shared wire. All senders then compute a random retransmission time and retry, where the randomization is chosen to minimize the probability of further collisions.

Theoretical analyses (e.g., [BG85]) claimed that as the utilization of an Ethernet grew, the effective throughput of the Ethernet dropped to zero as the entire bandwidth was wasted.

\(^1\)A device commonly known as a LAN switch typically implements bridge functionality.
on retransmissions. A second charge against Ethernet was its small distance limit of 1.5 Km, much smaller than the limits imposed by say the IBM Token Ring.

While the limited bandwidth charge turned to be false in practice ([BMK88]), it remained a potent marketing bullet for a long time. The second limited distance charge was, and remains, a true limitation of a single Ethernet. In this embattled position around 1980, network marketing people at Digital Equipment Corporation (DEC) pleaded with their technical folks for a technical riposte to these attacks. Could not their bright engineers find a clever way to “extend” a single Ethernet such that it could become a longer Ethernet with a larger effective bandwidth?

First, it was necessary to discard some unworkable alternatives. Physical layer bit repeaters were unworkable because they did not avoid the distance and bandwidth limits of ordinary Ethernets. Extending an Ethernet using a router did, in theory, solve both problems but introduced two other problems. First, in those stone ages, routers were extremely slow and could hardly keep up with the speed of the Ethernet.

Second, there were at least six different routing protocols in use at that time including IBM's SNA, Xerox’s SNS, DECNET, Appletalk etc. Hard as it may seem to believe, the Internet protocols were then only a small player in the marketplace. Thus a router would have to be complex beast capable of routing multiple protocols (as Cisco would do a few years later), or one would have to incur the extra cost of placing multiple routers, one for each protocol. Thus the router solution was considered a non-starter.

Routers interconnect links using information in the routing header, while repeaters interconnect links based on physical layer information such as bits. However, in classical network layering there is an intermediate layer called the Data Link layer. For an Ethernet, the Data Link layer is quite simple and contains a 48-bit unique Ethernet Destination address. Why is it not possible, the DEC group argued, to consider a new form of interconnection based only on the Data Link layer? They christened this new beast a Data Link layer relay or a bridge.

Let us take an imaginary journey into the mind of Mark Kempf, an engineer in the Advanced Development Group at DEC, who invented bridges around 1980. Undoubtedly, something like the drawing shown in Figure 10.1 was drawn on the chalkboard in Mark’s cubicle in Tewksbury, MA. Figure 10.1 shows two Ethernets connected together by a bridge; the lower Ethernet line contains stations A and B, while the upper Ethernet contains station C.

The bridge should make the two Ethernets look like one big Ethernet, so that when A sends an Ethernet packet to C it magically gets to C without A having to even know there is a bridge in the middle. Perhaps Mark reasoned as follows in his path to a final solution:

2Note that Ethernet 48-bit address have no relation to 32-bit Internet addresses.
**Packet Repeater:** Suppose A sends a packet to C (on the lower Ethernet) with destination address C and source address A. Assume the bridge picks up the entire packet, buffers it, and waits for a transmission opportunity to send it on the upper Ethernet. This avoids the physical coupling between the collision resolution processes on the two Ethernets that would be caused by using a bit repeater. Thus the distance span increases to 3 Km, but the effective bandwidth is still that of one Ethernet because every frame is sent on both Ethernets.

**Filtering Repeater:** The frame repeater idea causes needless waste (P1) in Figure 10.1 when A sends a packet to B by sending the packet unnecessarily on the upper Ethernet. This waste can be avoided if the bridge has a table that maps station addresses to Ethernets. For example, in Figure 10.1 suppose the bridge has a table that maps A and B to the lower Ethernet and C to the upper Ethernet. Then on receipt of a packet from A to B on the lower Ethernet, the bridge need not forward the frame because the table indicates that the destination B is on the same Ethernet the packet was received on.

If say a fraction $p$ of traffic on each Ethernet is to destinations on the same Ethernet (locality assumption), then the overall bandwidth of the two Ethernet system becomes $(1 + p)$ times the bandwidth of a single Ethernet. This follows because the fraction $p$ can be simultaneously sent on both Ethernets, increasing overall bandwidth by this fraction. Hence both bandwidth and distance increase. The only trouble is to figure out how the mapping table is built.

**Filtering Repeater with Learning:** It is infeasible to have a manager build a mapping table for a large bridged network. Can the table be built automatically? One aspect of our principle P13 (exploit degrees of freedom) is Polya’s [Pol57] problem solving question: “Have you used all the data?” So far, the bridge has looked only at destination addresses to forward the data. Why not also look at source addresses? When receiving a frame from A to B, the
bridge can look at the source address field to realize that $A$ is on the lower Ethernet. Over time, the bridge will learn the ports through which all active stations can be reached.

Perhaps Mark rushed out after his insight shouting “Eureka”. But he still had to work out a few more issues. First, because the table is initially empty, bridges must forward a packet, perhaps unnecessarily, when the location of the destination has not been learnt. Second, to handle station movement table entries must be timed out if the source address is not seen for some time period $T$. Third, the entire idea generalizes to more than two Ethernets connected together without cycles, to bridges with more than two Ethernet attachments, and to links other than Ethernets that carry destination and source addresses. But there was a far more serious challenge that needed to be resolved.

10.2 Challenge 2: Wire-Speed Forwarding

When the idea was first proposed, some doubting Thomas at DEC noticed a potential flaw. In Figure 10.1 suppose that $A$ sends 1000 packets to $B$, and then $A$ follows this burst by sending say 10 packets to $C$. The bridge receives the thousand packets, buffers them, and begins to work on forwarding (actually discarding) them. Suppose the time that the bridge takes to look up its forwarding table is twice as long as the time it takes to receive a packet. Then after a burst of 1000 back-to-back packets arrive, a queue of 500 packets from $A$ to $B$ will remain as a backlog of packets that the bridge has not even examined.

Since the bridge has a finite amount of buffer storage for say 500 packets, when the burst from $A$ to $C$ arrives they may be dropped without examination because the bridge has no more buffer storage. This is ironic because the packets from $A$ to $B$ that are in the buffer will be dropped after examination, but the bridge has dropped packets from $A$ to $C$ that needed to be forwarded. One can change the numbers used in this example but the bottom line is as follows. If the bridge takes more time to forward a packet than the minimum packet arrival time, there are always scenarios in which packets to be forwarded will be dropped because the buffers are filled with packets that will be discarded.

The critics were quick to point out that routers did not have this problem\textsuperscript{3} because routers only dealt with packets that were addressed to the router. Thus if a router were used, the router Ethernet interface would not even pick up packets destined to $B$, avoiding this scenario.

To finesse this issue and avoid interminable arguments, Mark proposed doing an implementation that would do \textit{wire speed forwarding} between two Ethernets. In other words, the bridge

\textsuperscript{3}Oddly enough even routers have the same problem to allow routers to distinguish important packets from less important ones in times of congestion, but this was not taken seriously in the 1980’s.
would lookup the destination address in the table (for forwarding) and the source address (for learning) in the time it took for a minimum sized packet to arrive on an Ethernet. Given a 64 byte minimum packet, this left 51.2 usec to forward a packet. Since a two port bridge could receive a minimum size packet on each of its Ethertns every 51.2 usec, this actually translated into doing two lookups (destination and source) every 25.6 usec.

It is hard to appreciate today, when wire speed forwarding has become commonplace, how astonishing this goal was in the early 1980's. This is because in those days one would be fortunate to find an interconnect device (e.g., router, gateway) that worked at kilobit rates, let alone at 10 Mbit/sec. Impossible, many thought. To prove them wrong, Mark built a prototype as part of the Advanced Development Group in DEC. A schematic of his prototype, which became the basis for the first bridge, is shown in Figure 10.2.

![Figure 10.2: Implementation of the first Ethernet to Ethernet bridge](image)

The design of Figure 10.2 consists of a processor (the first bridge used a Motorola 68000), two Ethernet chips (the first bridge used AMD Lance chips), a lookup chip (this is described in more detail below), and a 4-ported shared memory. The memory could be read and written by the processor, the Ethernet chips, and the lookup engine.

The data flow through the bridge was as follows. Imagine a packet P sent on Ethernet 1. Both Ethernet chips were set in "promiscuous mode" whereby they received all packets. Thus the bits of P are captured by the upper Ethernet chip and stored in the shared memory in a receive queue. The processor eventually reads the header of P, extracts the destination address D and gives it to the lookup engine.

The lookup engine looks up D in a database also stored in the shared memory and returns the port (upper or lower Ethernet) in around 1.3 usec. If the destination is on the upper Ethernet, the packet buffer pointer is moved to a free queue, effectively discarding the packet;
otherwise, the buffer pointer is moved to the transmit queue of the lower Ethernet chip. The processor also provides the source address $S$ in packet $P$ to the lookup engine for learning.

His design paid careful attention to algorithmics in at least three areas to achieve wire speed forwarding at a surprisingly small manufacturing cost of around 1000 dollars.

- **Architectural Design:** To minimize cost, the memory was cheap DRAM with a cycle time of 100 nsec that was used for packet buffers, scratch memory, and for the lookup database. The 4-port memory (including the separate connection from the lookup engine to the memory) and the busses were carefully designed to maximize parallelism and minimize interference. For example, while the lookup engine worked on doing lookups to memory, the processor continued to do useful work. Note that the processor has to examine the receive queues of both Ethernet chips in dovetailed fashion to check for packets to be forwarded from either the top or bottom Ethernets. Careful attention was paid to memory bandwidth including the use of page mode (Chapter 2).

- **Data Copying:** The Lance chips used DMA (Chapter 5) to place packets in the memory without processor control. When a packet was to be forwarded between the two Ethernets, the processor only flips a pointer from the receive queue of one Ethernet chip to the transmit queue of the other processor.

- **Control Overhead:** As with most processors, the interrupt overhead of the 68000 was substantial. To minimize this overhead, the processor used polling, staying in a loop after a packet interrupt, servicing as many packets as arrive in order to reduce context switching overhead (Chapter 6). When the receive queues are empty, the processor moves to doing other chores such as processing control and management traffic. The first data packet arrival after such an idle period interrupts the processor but this interrupt overhead is overhead over the entire batch of packets that arrive before another idle period begins.

- **Lookups:** Very likely, Mark went through the eight cautionary questions found Chapter 3. First, to avoid any complaints, he decided to use binary search for lookup because of its determinism. Second, having a great deal of software experience before he began designing hardware, he wrote some sample 68000 code and determined that software binary search lookup was the bottleneck (Q2 in Chapter 3) and would exceed his packet processing budget of 25.6 usec.

Eliminating the destination and source lookup would allow him to achieve wire speed forwarding (Q3). Recall that each iteration of binary search reads an address from the database in memory, compares it with the address that must be looked up, and uses this
comparison to determine the next address to be read. In hardware, the comparison can be implemented using combinatorial logic (Chapter 2) and so a first order approximation of lookup time is the number of DRAM memory accesses.

As the first product aimed for a table size of $8000^4$, this required $\log_2 8000$ memory accesses of 100 nsec each, yielding a lookup time of 1.3 usec. Given that the processor does useful work during the lookup, two lookups for source and destination easily fit within a 25.6 usec budget (Q4).

To answer Q5 in Chapter 3 as to whether custom hardware is worthwhile, Mark found that the lookup chip could be cheaply and quickly implemented using a PAL (programmable array logic, see Chapter 2). To answer Q7, his initial prototype met wire speed tests constructed using logic analyzers. Finally, Q8 which asks about the sensitivity to environment changes was not relevant to a strictly worst-case design like this.

The 68000 software, written by Bob Shelley, also had to be carefully constructed to maximize parallelism. After the prototype was built, Tony Lauck, then head of DECNET, was worried that bridges would not work correctly if they were placed in cyclic topologies. For example, if two bridges are placed between the same pair of Ethernets, messages sent on one Ethernet will be forwarded at wire speed in the loop between bridges. In response, Radia Perlman, then the DEC routing architect, invented her celebrated spanning tree algorithm. The algorithm ensures that bridges compute a loop-free topology by having redundant bridges turn off appropriate bridge ports.

While you can read up on the design of the spanning tree algorithm in Radia’s book [Per92], it is interesting to note that there was initial resistance to implementing her algorithm which appeared to be “complex” compared to simple, fast bridge data forwarding. However, the spanning tree algorithm used control messages called Hellos that are not processed in real-time.

A simple back-of-the-envelope calculation by Tony Lauck related the number of instructions used to process a hello (at most 1000), the rate of hello generation (specified at that time to be once every second), and the number of instructions per second of the Motorola 68000 (around 1 million). Lauck’s vision and analysis carried the day, and the spanning tree algorithm was implemented in the final product.

Manufactured at a cost of one thousand dollars, the first bridge was initially sold at a markup of around eight, ensuring a handsome profit for DEC when sales initially climbed. Mark

---

4This allows a bridged Ethernet to only have 8000 stations. While this is probably sufficient for most customer sites, later bridge implementations raised this figure to 16K and even 64K
Kempf was awarded U.S. Patent 4,597,07 titled “Bridge circuit for interconnecting networks” in 1986. DEC made no money from patent licensing, choosing instead to promote the IEEE 802.1 bridge interconnection standards process.

Together with the idea of self-learning bridges, the spanning tree algorithm has also passed into history. Ironically, one of the first customers complained that their bridge did not work correctly; field service later determined that the customer had connected two bridge ports to the same Ethernet, and the spanning tree had (rightly) turned the bridge off! While features like autoconfigurability and provable fault-tolerance have only recently been added to Internet protocols, they were part of the bridge protocols in the 1980’s.

The success of Ethernet bridges led to proposals for several other types of bridges connecting other local area networks and even wide-area bridges. The author even remembers working with John Hart (who went on to become CTO of 3Com) and Fred Baker (who went on to become a Cisco Fellow) on building satellite bridges that could link geographically distributed sites. While some of the initial enthusiasm to extend bridges to supplplant routers was somewhat extreme, bridges found their most successful niche in cheaply interconnecting similar local area networks at wire speeds.

However, after the initial success of 10 Mbps Ethernet bridges, engineers at DEC began to worry about bridging higher speed LANs. In particular, DEC decided, perhaps unwisely, to concentrate their high speed interconnect strategy around 100 Mbps FDDI token rings [UNH01]. Thus in the early 1990’s, engineers at DEC and other companies began to worry about building a bridge to interconnect two 100 Mpbs FDDI rings. Could wire speed forwarding, and especially exact match lookups, be made ten times faster?

10.3 Challenge 3: Scaling Lookups to Higher Speeds

First, let’s understand why binary search forwarding does not scale to FDDI speeds. Binary search takes \( \log_2 N \) memory accesses to lookup a bridge database, where \( N \) is the size of the database. As bridges grew popular, marketing feedback indicated that the database size needed to be increased from 8K to 64K. Thus using binary search, each search would take 16 memory accesses. Doing a search for the source and destination addresses using 100 nsec DRAM would then take 3.2 usec.

Unlike Ethernet where small packets are padded to ensure a minimum size of 64 bytes, a minimum size packet consisting of FDDI, routing and transport protocol headers could be as small as 40 bytes. Given that a 40 byte packet can be received in 3.2 usec at 100 Mbps,
two binary search lookups would use up all of the packet processing budget for a single link, leaving no time for other chores such as inserting and removing from link chip queues.

One simple approach to meet the challenge of wire speed forwarding is to retain binary search but to use faster hardware (P5). In particular faster SRAM (Chapter 2) could be used to store the database. Given a factor of 5 to 10 decrease in memory access time using SRAM in place of DRAM, binary search will easily scale to wire speed FDDI forwarding.

However, this approach is unsatisfactory for two reasons. First, it is more expensive because SRAM is more expensive than DRAM. Second, using faster memory gets us lookups at FDDI speeds but will not work for the next speed increment (e.g., Gigabit Ethernet). What is needed is a way to reduce the number of memory accesses associated with a lookup so that bridging can scale with link technology. Of the two following approaches to bridge lookup scaling, one is based on hashing and the second is based on hardware parallelism.

10.3.1 Scaling via Hashing

In the 1990's, DEC decided to build a fast crossbar switch connecting up to 32 links, called the Gigaswitch [SKO+94]. The switch arbitration algorithms used in this switch will be described in Chapter 13. This chapter concentrates on the bridge lookup algorithms used in the Gigaswitch. The vision of the original designers, Bob Simcoe and Bob Thomas, was to have the Gigaswitch be a switch connecting point-to-point FDDI links without implementing bridge forwarding and learning. Bridge lookups were considered to be too complex at 100 Mbps speeds.

Into the development arena strode a young software designer who changed the product direction. Barry Spinney, who had implemented an Ada compiler in his last job, was determined to do hardware design at DEC. Barry suggested that the Gigaswitch be converted to a bridge interconnecting FDDI Local Area networks. To do so, he proposed designing an FDDI-to-GIGASwitch network controller (FGC) chip on the line cards that would implement a hashing based algorithm for lookups. The Gigaswitch article [SKO+94] states that each bridge lookup makes at most 4 reads from memory.

Now every student of algorithms [CLR90] knows that hashing, on average, is much faster (constant time) than binary search (logarithmic time). However, the same student also knows that hashing is much slower in the worst case, potentially taking linear time because of collisions. How, then, can the Gigaswitch hash lookups claim to take at most 4 reads to memory in the worst case even for bridge databases of size 64K, whereas binary search would require 16 memory accesses?
The Gigaswitch trick has its roots in a theoretical technique called \textit{perfect hashing} [DKea88]. The idea is to use a parameterized hash function where the hash function can be changed by varying some parameters. Then appropriate values of the parameters can be precomputed (P3a) to obtain a hash function such that the worst-case number of collisions is small and bounded.

While finding such a good hash function may take (in theory) a large amount of time, this is a good tradeoff because this new station’s addresses do not get added to local area networks at a very rapid rate. On the other hand, once the hash function has been picked, lookup can be done at wire speeds.

Specifically, the Gigaswitch hash function treats each 48-bit address as a 47-degree polynomial in the Galois field of order 2, GF(2). While this sounds impressive, this is the same arithmetic used for calculating CRCs; it is identical to ordinary polynomial arithmetic except that all additions are done mod 2. A hashed address is obtained by the equation $A(X) \times M(X) \bmod G(X)$, where $G(X)$ is the irreducible polynomial, $X^{48} + X^{36} + X^{25} + X^{10} + 1$, $M(X)$ is a non-zero, 47-degree programmable hash multiplier, and $A(X)$ is the address expressed as a 47-degree polynomial.

![Diagram](image_url)

\textbf{Figure 10.3:} Gigaswitch hashing uses a hash function with a programmable multiplier, a small balanced binary tree in every hash bucket, and a backup CAM to hold the rare case of entries that result in more than 7 collisions.

The hashed address is 48 bits. The bottom 16 bits of the hashed address is then used as an index into a 64K-entry hash table. Each hash table entry (see Figure 10.3 as applied to the destination address lookup, with $D(x)$ being used in place of $A(x)$) points to the root of a balanced binary tree of height at most 3. The hash function has the property that it suffices to use only the remaining high-order 32 bits of the hashed address to disambiguate collided keys.

Thus the binary tree is sorted by these 32 bit values, instead of the original 48-bit key. This saves 16 bits to be used for associated lookup information. Thus any search is guaranteed to take no more than 4 memory accesses, 1 to lookup the hash table, and 3 more to navigate
a height 3 binary tree.

It turns out that picking the multiplier is quite easy in practice. The coefficients of $M(x)$ are picked randomly. Having picked $M(x)$ it sometimes happens that a few buckets have more than 7 colliding addresses. In such a case, these entries are stored in a small hardware lookup database called a CAM (studied in more detail in Chapter 11).

The CAM lookup occurs in parallel with the hash lookup. Finally, in the extremely rare case when several dozen addresses are added to the CAM (say when new station addresses are learned that cause collisions), the central processor initiates a rehashing operation and distributes the new hash function to the line cards. It is perhaps ironic that rehashing occurred so rarely in practice that one might worry whether the rehashing code was adequately tested!

The Gigaswitch became a successful product, allowing up to 22 FDDI networks to be bridged together with other link technologies such as ATM. Barry Spinney was assigned U.S. patent 5,920,900 “Hash-based translation method and apparatus with multiple level collision resolution”. While techniques based on perfect hashing [DKea88] have been around for a while in the theoretical community, Barry’s contribution was to use a pragmatic version of the perfect hashing idea for high speed forwarding.

10.3.2 Using Hardware Parallelism

Techniques based on perfect hashing do not completely provide worst-case guarantees. While they do provide worst-case search times of 3-4 memory accesses, they cannot guarantee worst case update times. It is conceivable that an update takes an unpredictably long time while the software searches for a hash function with the specified bound on the number of collisions.

One can argue that exactly the same guarantees are provided every moment by millions of Ethernets around the world, and that non-deterministic update times are far preferable to non-deterministic search times. However, proving that long update times are rare in practice either requires considerable experimentation or good analysis. This makes some designers uncomfortable. It leads to a preference for search schemes that have bounded worst-case search and update times.

An alternate approach is to apply hardware parallelism (P5) to a deterministic scheme like binary search. Binary search has deterministic search and update times: its only problem is that search takes a logarithmic number of memory accesses which is too slow. We can get around this difficulty by pipelining binary search to increase lookup throughput (number of lookups per second) without improving lookup latency. This is illustrated in Figure 10.4.
The idea is to have a logarithmic number of processing stages, each with its own memory array. In Figure 10.4 the keys are the characters \( A \) through \( H \). The first array has only the root of the trie, the median element \( E \). The second array corresponds to the quartile and third quartile elements \( C \) and \( G \), which are the possible keys at the second probe of binary search, and so on. Search keys enter from the left and progress from stage to stage, carrying a pointer which identifies which key in the corresponding stage memory must be compared to the search key. The lookup throughput is nearly one per memory access because there can be multiple concurrent searches progressing through the stages in order.

Although the figure shows the elements in say Stage 2, \( C \) and \( G \) as being separated by their spacing in the original table, they can be packed together to save memory in the stages. Thus the overall memory across all stages becomes equal to the memory in a non-pipelined implementation. Indexing into each stage memory becomes slightly more tricky.

Assuming Stage \( i \) has passed a pointer \( j \) to Stage \( j + 1 \) along with search key \( S \). Stage \( j + 1 \) compares the search key \( S \) to its \( j \)-th array entry. If the answer is equal, the search is finished but continues flowing through the pipeline with no more changes. If the search key is smaller, the search key is passed to stage \( i + 1 \) with the pointer \( j0 \) (i.e., \( j \) concatenated with bit 0); if the search key is larger, the pointer passed is \( j1 \). For example, if the key searched for is \( F \), the pointer becomes 1 when entering stage 2, and becomes 10 when entering stage 3.

The author first heard of this idea from Greg Waters, who later went on to implement IP lookups for the core router company Avici. While the idea looks clever and arcane, there is a much simpler way of understanding the final solution. Computer Scientists are well aware of the notion of a binary search tree [CLR90]. Any binary search table can be converted into a fully balanced binary search tree by making the root the median element and so on, along
the lines of Figure 10.4. Any tree is trivially pipelined by height, with nodes of height $i$ being assigned to Stage $i$.

The only problem with a binary search tree, as opposed to a table, is the extra space required for pointers to children. However, it is well known that for a full binary search tree such as a heap [CLR90], the pointers can be implicit and can be calculated based on the history of comparisons — as shown above. The upshot is that a seemingly abstruse trick can be seen as the combination of three simple and well-known facts from theoretical computer science.

10.4 Summary

This chapter on exact match lookups is written as a story — the story of bridging. Three morals can be drawn from this story.

First, bridging was a direct response to the challenge of efficiently extending Ethernets without using routers or repeaters; wire-speed forwarding was a direct response to the problem of potentially losing important packets in a flood of less important packets. At the risk of sounding like a self-help book, challenges are best regarded as opportunities and not as annoyances. The mathematician Félix Klein [Bel86] used to say “You must always have a problem: you may not find what you were looking for but you will find something interesting on the way.” For example, it is clear that the main reason why bridges were invented — the lack of high performance multiprotocol routers — is not the reason why bridges are still useful today.

This brings us to the second moral. Today it is clear that bridges will never displace routers because of their lack of scalability using flat Ethernet addresses, lack of shortest cost routing, etc. However, they remain interesting today because bridges are interconnect devices with better cost-performance and flexibility than routers for interconnecting a small number of similar Local Area Networks. Thus bridges still abound in the marketplace, often referred to as “switches”. What many network vendors refer to as a “switch” is a crossbar switch like the Gigaswitch that is capable of bridging on every interface. A few new features, notably Virtual LANs (VLANs) [Per92], have been added. But the core idea remains the same.

Third, the techniques introduced by the first bridge have deeply influenced the next generation of interconnect devices, from core routers to web switches. Recall that Roger Bannister, who first broke the four minute mile barrier, was followed in a few months by several others. In the same way, the first Ethernet bridge was quickly followed by many other wire-speed bridges. Soon the idea began to flow to routers as well. Other important concepts introduced by bridges include the use of memory references as a metric, the notion of trading update time
for faster lookups, and the use of minimal hardware speedups. All these ideas carry over into the study of router lookups in the next chapter.

In conclusion, the challenge of building the first bridge stimulated creative actions that went far beyond the first bridge. While wire-speed router designs are fairly commonplace today, it is perhaps surprising that there are products being announced today that claim gigabit wire speed processing rates for such abstruse networking tasks as encryption and even XML transformations.
Part IV: Playing with the Future

*Daring ideas are like chessmen moved forward. They may be beaten, but they may start a winning game.*

— Goethe

*We didn’t lose the game; we just ran out of time.*

— Vince Lombardi

The last part of the book applies Internet Algorithmics to the emerging fields of security and measurement. As the Internet matures, we believe that good abstractions for security and measurement will be key to well-engineered networks. While the problems (e.g., detecting a DoS attack at a high speed router) seem hard, some remarkable ideas have been proposed. The final chapter tries to reach closure by distilling the underlying unities behind the many different techniques surveyed in this book.
Chapter 18

Endgame

_The end of a matter is better than its beginning._
— Ecclesiastes, The Bible

We began in the first part of the book by setting up the rules of the Internet Algorithmics game. The second part of the book dealt with server implementations and the third part with router implementations. The fourth and last part of the book dealt with current and future issues in measurement and security.

The book covers a large number of specific techniques and a variety of settings — there are techniques for fast server design versus techniques for fast routers, techniques specific to operating systems versus techniques specific to hardware. While all these topics are part of the spectrum of Internet Algorithmics, there is risk that the material can appear to degenerate into a patchwork of assorted topics that are not linked together in any coherent way.

Thus as we draw to a close, it is appropriate to try and reach closure by answering the following questions in the next four sections.

- **What was the book about?** What were the main problems and how did they arise? What are the main techniques. While endnode and router techniques appear to be different when considered superficially, are there some underlying unities between these two topics? Can these unities be exploited to suggest some cross-fertilization between these areas? (Section 18.1)

- **What is Internet Algorithmics about?** What is the underlying philosophy behind Internet Algorithmics, and how does it differ from algorithms by themselves. (Section 18.2)
• *Is Internet Algorithmics used in real systems?* Are the techniques in this book exercises in speculation, or are there real systems that use some of these techniques. Section 18.3

• *What is the future of Internet Algorithmics?* Are all the interesting problems already solved? Are the techniques studied in this book only useful to understand existing work or guide new implementations of existing tasks. Or are there always likely to be new problems that will require fresh applications of the principles and techniques described in this book? (Section 18.4)

### 18.1 What this book was about

The main problem considered in this book is bridging the performance gap from good network abstractions and fast network hardware. Abstractions — such as modular server code and prefix-based forwarding — make networks more usable, but also exact a performance penalty when compared to the capacity of raw transmission links such as optical fiber. The central question tackled in this book is whether we can have our cake and eat it too: retain the usability of the abstractions and yet achieve wire speed performance for the fastest transmission links.

To make this general assertion more concrete we review the main contents of this book in two sections: Section 18.1.1 on endnode algorithmics, and Section 18.1.2 III on router algorithmics. This initial summary is similar to that found in Chapter 1. However, we go beyond the description in Chapter 1 in Section 18.1.3 where we present the common themes in endnode and router algorithmics, and suggest how these unities can potentially be exploited.

#### 18.1.1 Endnode Algorithmics

Chapter 5 to Chapter 9 of this book concentrate on endnode algorithmics, especially for servers. Much of the problems tackled under endnode algorithmics involve getting around complexities due to software and structure — in other words, complexities of our own making as opposed to necessarily fundamental complexities. These complexities arise because of the following characteristics of endnodes:

• *Computation versus Communication:* Endnodes are about general purpose computing and must handle possible unknown and varied computational demands from database queries to weather prediction. By contrast, routers are devoted to communication.
• **Vertical versus Horizontal Integration:** Endnodes are typically horizontally integrated with one institution building boards, another writing kernel software, and another writing applications. In particular, kernels have to be designed to tolerate unknown and potentially buggy applications to run on top of them. Today, routers are typically vertically integrated where the hardware and all software is assembled by a single company.

• **Complexity of computation:** Endnode protocol functions are more complex (application, transport) compared to the corresponding functions in routers (routing, data link).

As a consequence, endnode software has three important artifacts which seem hard to avoid, each of which contributes to inefficiencies that must be worked around or minimized.

• 1. **Structure:** Because of the complexity and vastness of endnode software, code is structured and modular to ease software development. In particular, unknown applications are allowed using a standard application programming interface (API) between the core operating system and the unknown application.

• 2. **Protection:** Because of the need to accommodate unknown applications, there is a need to protect applications from each other, and to protect the operating system from applications.

• 3. **Generality:** Core routines such as buffer allocators and the scheduler are written with the most general use (and the widest variety of applications) in mind and thus are unlikely to be as efficient as special purpose routines.

In addition, since most endnodes were initially designed in an environment where the endnode communicated with only a few nodes at a time. Thus, it is little surprise that when these nodes were retrofitted as servers a fourth artifact was discovered.

• 4. **Scalability:** By scalability, we often mean scalability in terms of the number of concurrent connections. A number of operating systems use simple data structures that work well for a few concurrent connections but become major bottlenecks in a server environment where there are a large number of connections.

With this list of four endnode artifacts in mind, Figure 18.1 reviews the main endnode bottlenecks covered in this book together with causes and workarounds. This picture is a more detailed version of the corresponding figure in Chapter 1.

542
<table>
<thead>
<tr>
<th><strong>Bottleneck</strong></th>
<th><strong>Chapter</strong></th>
<th><strong>Cause</strong></th>
<th><strong>Sample Solution</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Copying</td>
<td>5</td>
<td>Protection, structure</td>
<td>Passing by reference optimized by caching (IOLite)</td>
</tr>
<tr>
<td>Context Switching</td>
<td>6</td>
<td>Complex Scheduling</td>
<td>User-level protocols, event-driven web servers</td>
</tr>
<tr>
<td>System calls</td>
<td>6</td>
<td>Protection, structure</td>
<td>Application device channels</td>
</tr>
<tr>
<td>Slow select</td>
<td>6</td>
<td>Scaling with # of clients</td>
<td>Kernel keeps state across calls</td>
</tr>
<tr>
<td>Demuxing</td>
<td>7</td>
<td>Scaling with # of classifiers</td>
<td>Generalized Tries (Pathfinder)</td>
</tr>
<tr>
<td>Timers</td>
<td>8</td>
<td>Scaling with # of timers</td>
<td>Timing wheels</td>
</tr>
<tr>
<td>Buffer alloc.</td>
<td>9</td>
<td>Generality</td>
<td>Linear buffers</td>
</tr>
<tr>
<td>Checksums/CRCs</td>
<td>10</td>
<td>Generality</td>
<td>Multibit computation</td>
</tr>
<tr>
<td>Protocol code</td>
<td>10</td>
<td>Generality</td>
<td>Header Prediction</td>
</tr>
</tbody>
</table>

Figure 18.1: Endnode bottlenecks covered in this book. Associated with each bottleneck is the chapter in which the material is reviewed, the underlying cause, and one or more sample solutions.

18.1.2 Router Algorithmics

In router algorithmics, by contrast, the bottlenecks are caused not by structuring artifacts (as in some problems in endnode algorithmics) but by the scaling problems caused by the need for global internets, together with the fast technological scaling of optical link speeds. Thus the global internet puts pressure on router algorithmics because of both population scaling and speed scaling.

For example, simple caches worked fine for route lookups till address diversity and the need for CIDR (both caused by population scaling) forced the use of fast longest matching prefix. Also, simple DRAM-based schemes sufficed for prefix lookup (e.g., using expanded tries) till increasing link speeds forced the use of limited SRAM and compressed tries. Unlike endnodes, routers do not have protection issues as they largely execute one code base. The only variability comes from different packet headers. Hence protection is less of an issue.
With the two main drivers of router algorithmics in mind, Figure 18.2 reviews the main router bottlenecks covered in this book together with causes and workarounds. This picture is a more detailed version of the corresponding figure in Chapter 1.

<table>
<thead>
<tr>
<th>Bottleneck</th>
<th>Chapter</th>
<th>Cause</th>
<th>Sample Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Switching</td>
<td>11</td>
<td>Electrical scaling of busses</td>
<td>Crossbar switches</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Scaling in bandwidth</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Head of line blocking</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Scalability in # of ports</td>
<td></td>
</tr>
<tr>
<td>Prefix Lookups</td>
<td>12</td>
<td>CIDR, link speed scaling</td>
<td>Expanded multibit tries</td>
</tr>
<tr>
<td></td>
<td></td>
<td>prefix database size scaling</td>
<td></td>
</tr>
<tr>
<td>Packet Classification</td>
<td>13</td>
<td>Service differentiation</td>
<td>Decision trees and heuristics</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Link speed and size scaling</td>
<td></td>
</tr>
<tr>
<td>Fair Queueing</td>
<td>14</td>
<td>Service differentiation in resource scheduling</td>
<td>Weighted fair queuing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Link speed scaling</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Memory scaling</td>
<td></td>
</tr>
<tr>
<td>Measurement</td>
<td>16</td>
<td>link speed scaling, # of counters</td>
<td>low order bits in SRAM + DRAM</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Security</td>
<td>17</td>
<td>Scaling in number and intensity of attacks</td>
<td>Traceback with Bloom Filters</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 18.2: Router bottlenecks covered in this book. Associated with each bottleneck is the chapter in which the material is reviewed, the underlying cause, and one or more sample solutions.

While we have talked about routers as *the* canonical switching device many of the techniques discussed in this book apply equally well to any switching device such as a bridge (Chapter 10 is devoted to lookups in bridges) or a gateway. It also applies to intrusion detection systems, firewalls, and network monitors who do not switch packets but must still work efficiently with packet streams at high speeds.

18.1.3 Towards a synthesis

In his book *The Character of Physical Law*, Richard Feynman argues that we have a need to understand the world in “various hierarchies, or levels”. Later, he goes on to say that “all the
sciences, and not just the sciences but all the efforts of intellectual kinds, are an endeavor to see the connections of the hierarchies... and in that way we are gradually understanding this tremendous world of interconnecting hierarchies.”

We have divided Internet Algorithmics into two hierarchies: endnode algorithmics and router algorithmics. What are the connections between these two hierarchies? Clearly, we have used the same set of fifteen principles to understand and derive techniques in both parts of the book. But are there other unities that can provide insight and suggest new directions?

There are differences between endnode and router algorithmics. Endnodes have large, structured, and general-purpose operating systems that require work-arounds to obtain high performance; routers, by contrast, have fairly primitive operating systems (e.g., Cisco IOS) that are bear some resemblance to a real-time operating system. Most endnodes protocol functions are implemented (today) in software, while the performance critical performance functions in a router are implemented in hardware. Endnodes compute, routers communicate. Thus routers have no file system and no complex process scheduling.

But there are similarities as well between endnode and router algorithmics. Roughly speaking:

- **Copying** in endnodes is analogous to the data movement orchestrated by *switching* in routers.
- **Demultiplexing** in endnodes is analogous to *classification* in routers.
- **Scheduling** in endnodes is analogous to *fair queuing* in routers.

Other than packet classification, where the analogy is more exact, it may seem that the other correspondences are a little stretched. However, these analogies suggest the following potentially fruitful directions:

1. **Switch-based endnode architectures:** The analogy between copying and switching, and the clean separation between I/O and computation in a router, suggests that this may also be a good idea for endnodes. More precisely, most routers have a crossbar switch that allows parallel data transfers using dedicated ASICs or processors; packets meant for internal computation are routed to a separate set of processors. While we considered this briefly in Chapter 2, we did not consider very deeply the implications for endnode operating systems.

By dedicating memory bandwidth and processing to I/O streams, the main computational processors can compute without interruptions, system calls, or kernel thread as I/O is essentially serviced and placed in clean form by a set of I/O processors (using separate memory
bandwidth that does not interfere with the main processors) for use by the computational processors when they switch computational tasks. With switch-based bus replacements such as Infiniband, and the increasing use of protocol offload engines such as TCP chips, this vision may be realizable in the near future. However, while the hardware elements are present, there is need for a fundamental restructuring of operating systems to make this possible.

2. \textit{Generalized endnode packet classification}: Although there seems to be a direct correspondence between packet classification in endnodes (Chapter 8) and packet classification in routers (Chapter 12), the endnode problem is simpler because it works only for a constrained set of classifiers where all the wildcards are at the end. Router classifiers, on the other hand, allow arbitrary classifiers, requiring more complicated algorithmic machinery or CAMs.

It seems clear that if early demultiplexing is a good idea, that there are several possible definitions of a “path” (flow in router terminology) other than a TCP connection. For example, one might want to devote resources to all traffic coming from certain subnets or to certain protocol types. Such flexibility is not allowed by current classifiers such as BPF and DPF (Chapter 8). It may be interesting to study the extra benefits provided by more general classifiers in return for the added computational burden.

3. \textit{Fair queuing in endnodes}: Fair queuing in routers was originally invented to provide more discriminating treatment to flows in times of overload and (later) to provide Quality of Service to flows in terms of say latency. Both these issues resonate in the endnode environment. For example, the problem of receiver livelock (Chapter 6) requires discriminating between flows during times of overload. The use of early demultiplexing and separate IP queues per flow in Lazy Receiver Processing seem like a first crude step toward fair queuing. Similarly, many endnodes do real-time processing such as running MPEG players just as routers have to deal with the real-time constraints of say Voice-over-IP packets.

Thus, a reasonable question is whether the work on fair schedulers in the networking community can be useful in an operating system environment. When a sending TCP is scheduling between multiple concurrent connections, could it use a scheduling algorithm such as DRR for better fairness? At a higher level, could a web server use worst-case weighted fair queuing to provide better delay bounds for certain clients? Some work following this agenda has begun to appear in the operating system community, but it is unclear that the question has been fully explored.

So far, we have suggested that endnodes could learn from router design in overall I/O architecture and operating system design. Routers can potentially learn the following from endnodes.

1. \textit{Fundamental Algorithms}: Fundamental algorithms for endnodes such as selection, buffer
allocation, CRCs, and timers are likely to be useful for routers, because the router processor is still an endnode with very similar issues.

2. More structured router Operating Systems: While the internals of router operating systems such as Cisco’s IOS and Juniper’s JunOS are hidden from public scrutiny, there is at least anecdotal evidence that there are major software engineering challenges associated with such systems as time progresses (leading to the need to be compatible with multiple past versions) and customers ask for special builds. Perhaps routers can benefit from some of design ideas behind existing operating systems that have stood the test of time.

While protection may be fundamentally unnecessary (no third party applications running on a router), how should a router operating system be structured for modularity? One approach to building modular but efficient router operating system can be found in the Router Plugins system[DDPP98] and the Click Operating System[KMea00].

3. Vertically integrated routers: The components of an endnode (applications, operating system, boxes, chips) are often built by separate companies, thus encouraging innovation. The interface between these components is standardized (e.g., the API between applications and operating system) allowing multiple companies to supply new solutions. Why should a similar vision not hold for routers some years from now when the industry matures? Currently, this is more of a business rather than a technical issue as existing vendors do not want to open up the market to competitors. However, this was true in the past for computers, and is no longer true; thus there is hope.

We are already seeing router chips being manufactured by semiconductor companies. However, a great aid to progress would be a standardized router operating system that is serious and general enough for production use by several, if not all, router companies. Such a router operating system would have to work across a range of router architectures just as operating systems span a variety of multiprocessor and disk architectures.

Once this is the case, perhaps there is even a possibility of “applications” that run on routers. This is not as far-fetched as it sounds, because there could be a variety of security and measurement programs that operate on a subset of the packets received by the router. With the appropriate API (and especially if the programs are operating on a logged copy of the router packet stream), such applications could even be farmed out to third party application developers. It is probably easy to build an environment where a third party application (working on logged packets) cannot harm the main router functions such as forwarding and routing.

\[1\] Click is somewhat biased towards endnode bus-based routers as opposed to switch-based routers with ASIC support.

547
18.2 What Internet Algorithmics is about

Chapter 1 introduced Internet Algorithmics with the following

Definition: Internet Algorithmics is the use of an interdisciplinary systems approach, seasoned with algorithmic thinking, to design fast implementations of network processing tasks.

The definition stresses the fact that Internet Algorithmics is interdisciplinary, requires systems thinking, and can sometimes benefit from algorithmic thinking. We review each of these three aspects (interdisciplinary thinking, systems thinking, algorithmic thinking) in turn:

18.2.1 Interdisciplinary Thinking

Internet Algorithmics represents the intersection of several disciplines within computer science that are often taught separately. Endnode algorithmics is a combination of Networking, Operating Systems, Computer Architecture, and Algorithms. Router algorithmics is a combination of Networking, Hardware Design, and Algorithms. Figure 18.3 provides examples of uses of these disciplines that are studied in the book.

ENDNODE ALGORITHMICS

<table>
<thead>
<tr>
<th>Discipline</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Networking</td>
<td>Header Prediction (Chapter 6)</td>
</tr>
<tr>
<td>Operating Systems</td>
<td>Application Device Channels (Chapter 6)</td>
</tr>
<tr>
<td>Computer Architecture</td>
<td>Locality Driven Receiver Processing (Chapter 5)</td>
</tr>
<tr>
<td>Algorithms</td>
<td>Timing Wheels (Chapter 8)</td>
</tr>
</tbody>
</table>

ROUTER ALGORITHMICS

<table>
<thead>
<tr>
<th>Discipline</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Networking</td>
<td>Link striping (Chapter 15)</td>
</tr>
<tr>
<td>Hardware Design</td>
<td>Switch arbiters (Chapter 2 &amp; 10)</td>
</tr>
<tr>
<td>Algorithms</td>
<td>Fast IP Lookup (Chapter 12)</td>
</tr>
</tbody>
</table>

Figure 18.3: Examples of disciplines used in this book and sample applications

For example, in Figure 18.3 techniques like header prediction (Chapter 9) require a deep networking knowledge of TCP protocols to optimize the expected case, while internal link striping (Chapter 15) requires knowing how to correctly design a striping protocol. On the other hand, Application Device Channels (Chapter 6) requires a careful understanding of the protection issues in operating systems.
Similarly, Locality Driven Receiver Processing requires understanding the architectural function and limitations of the instruction cache. Finally, in router algorithms it is crucial to understand hardware design. Arbiter like iSLIP and PIM were designed to allow scheduling decisions in a minimum packet arrival time.

Later in this chapter we argue that other disciplines such as statistics and learning theory will also be useful for Internet Algorithmics.

### 18.2.2 Systems Thinking

Systems thinking is embodied by Principles 1 through 10. Principles 1 through 5 were described earlier as systems principles. Systems unfold in space and time: in space, through various components (e.g., kernel, application) and in time, through certain key time points (e.g., application initialization time, packet arrival time). Principles 1 through 5 ask that a designer expand his vision to see the entire system and then consider moving functions in space and time to gain efficiency.

For example, Principle **P1**, avoiding obvious waste, is a cliche by itself. However, our understanding of systems, in terms of separable and modular hierarchies, often precludes the synoptic eye required to see waste across system hierarchies. For example, the number of wasted copies is only apparent when one broadens one’s view to that of a web server (see I/O lite in Chapter 5). Similarly, the opportunities for dynamic code generation in going from PathFinder to DPF (see Chapter 8) are only apparent when one considers the code required to implement a generic classifier.

Similarly, Principle **P4**, asks the designer to be aware of existing system components that can be leveraged. Fbufs (Chapter 5), leverage off the Virtual Memory subsystem, while timing wheels (Chapter 7) leverage off the existing time-of-day computation to amortize the overhead of stepping through empty buckets. Principle **P4** also asks the designer to be specially aware of the underlying hardware whether to exploit local access costs (e.g., DRAM pages, cache lines), to trade memory for speed (either by compression if the underlying memory is SRAM, or by expansion if memory is DRAM), or to exploit other hardware features (e.g., replacing multiplies by shifts in RED calculations in Chapter 14).

Principle 5 asks the designer to be even bolder and consider adding new hardware to the system; this is especially useful in a router context. While this is somewhat vague, Principles **5a** (parallelism via memory interleaving), **5b** (parallelism via wide words), and **5c** (combining DRAM and SRAM to improve overall speed and cost) appear to underlie many clever hardware designs to implement router functions. Thus memory interleaving and pipelining can be used
to speed up IP lookups (Chapter 11), wide words are used to improve the speed of the Lucent classification scheme (Chapter 12), and DRAM and SRAM can be combined to construct an efficient counter scheme (Chapter 16).

Once the designer sees the system and identifies wasted sequences of operations together with possible components to leverage, the next step is to consider moving functions in time (P2) and space (P3). Figure 18.4 shows examples of endnode algorithmic techniques that move functions between components. Figure 18.5 shows similar examples for router algorithmics.

![Application Kernel VM system Adaptor Incoming Packet](image)

<table>
<thead>
<tr>
<th>Application</th>
<th>Kernel</th>
<th>VM system</th>
<th>Adaptor</th>
<th>Incoming Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td>(receive buffer specification)</td>
<td>RDMA</td>
<td>(copying)</td>
<td>Fbufs</td>
<td>(protection)</td>
</tr>
<tr>
<td>(scheduling)</td>
<td>Event driven servers</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 18.4: Endnode algorithmics: Examples of moving functions in space

![Forwarding Route Processor Previous Router Edge Router Source](image)

<table>
<thead>
<tr>
<th>Forwarding Engine</th>
<th>Route Processor</th>
<th>Previous Router</th>
<th>Edge Router</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>(precomputing lookup table)</td>
<td>IP Lookups</td>
<td>Tag Switching, MPLS</td>
<td>(handling bits, rates)</td>
<td>DiffServ, Coreslless</td>
</tr>
<tr>
<td>Path MTU approach to fragmentation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 18.5: Router algorithmics: Examples of moving functions in space

Besides moving functions in space, moving functions in time is also a key enabler for efficient algorithms. Besides the more conventional approaches of precomputation (P2a), lazy evaluation (P2b), and batch processing (P2c), there are subtler examples of moving functions to different times at which the system is instantiated. For example, in Fbufs (Chapter 5), common VM Mappings between the application and kernel are calculated when the application first starts up. Application Device Channels (Chapter 6) have the kernel authorize buffers (on behalf of an application) to the adaptor when the application starts up. DPF (Chapter 8) specializes code when a classifier is updated. Tag Switching (Chapter 11) moves the work of computing labels from packet forwarding time to route computation time.

Finally, Principles P6 through P10 concern the use of alternate system structuring techniques to remove inefficiencies. P6 suggests considering specialized routines or alternate inter-
faces; for example, Chapter 6 suggests that event driven APIs may be more efficient than the state-based interface of the select() call. P7 suggests designing interfaces to avoid unnecessary generality; for example, in Chapter 5, Fbufs map the Fbuf pages into the same locations in all processes avoiding the need for a further mapping when moving between processes. P8 suggests avoiding being unduly influenced by reference implementations; for example, in Chapter 9, naive reference implementations of checksums have poor performance.

Principles P9 and P10 suggest keeping existing interfaces but adding extra information to interfaces (P9) or packet headers (P10). For example, efficiently reimplementing the select() call in Chapter 6 requires passing information between the protocol module and the select module. Passing information in packet headers, on the other hand, has a huge array of examples including RDMA (Chapter 5), MPLS (Chapter 11), DiffServ and Core Stateless Fair Queuing (Chapter 14).

18.2.3 Algorithmic Thinking

Algorithmic thinking refers to thinking about networking bottlenecks the way algorithm designers approach problems. Overall, algorithmic approaches are less important than other systems approaches as embodied by Principles 1 through 10. Also, it is dangerous to blindly reuse existing algorithms.

The first problem that must be confronted in using algorithmic thinking is how to frame the problem that must be solved. By changing the problem one can often find more effective solutions. Consider the following problem that we avoided in Chapter 11.

Example: Pipelining and Memory Allocation: A lookup engine is using a trie. The lookup engine must be pipelined for speed. The simplest solution is to pipeline the trie by level. The root is at the first stage, the children of the root are assigned to the second stage, the nodes at height 2 to the third stage etc. Unfortunately, the memory needs for each stage can vary as prefixes are inserted and deleted. There are the following spectrum of approaches:

- **Centralized Memory**: All the processing stages share a single memory. Memory allocation is easy, but the centralized memory becomes a bottleneck.

- **One memory per stage**: Each processing stage has its own memory, minimizing memory contention. However, since the memory is statically allocated at fabrication time, any memory unused by a stage cannot be used by another stage.

- **Dynamically allocate small 1 port memories to stages**: As suggested in Chapter 11, on-chip memory is divided into $M$ SRAMs, that are connected to stage processors via a
crossbar. As a processor requires more or less memory, crossbar connections can be changed to allocate more or less memories to each stage. This scheme requires large $M$ to avoid wasting memory, but large $M$ can lead to high capacitive loads.

- **Dynamically allocate medium size 2 port memories to stages:** The setting is identical to the last approach, except that each memory is now a 2-port memory that can be allocated to two processors. Using this it is is possible to show that $N$ memories are sufficient for $N$ processors with almost no memory wastage.

- **Dynamically change the starting point in the pipeline:** In a conventional linear pipeline, all lookups start at the first stage and leave at the last. Florin Baboescu has suggested an alternative: using a lookup table indexed on the first few bits, assign each address to a different first processor in the pipeline. Thus different addresses have different start and end processors. However, this gives considerably more flexibility in allocating memory to processors by changing the assignment of addresses to processors.

- **Pipeline by depth:** Instead of pipelining a tree by height, consider pipelining by depth. All leaves are assigned to the last stage $K$, all parents of the leaves to stage $K - 1$, etc.

These approaches represent the interplay between principles $P13$ (optimizing degrees of freedom) and $P5$ (add hardware). However, each approach results in a different algorithmic problem! Thus a far more important skill than solving a hard problem is the skill required to frame the right problems that balance overall system needs.

Principles $P11$ and $P13$ help choose the right problem to solve. The pipelining example shows that choosing the degrees of freedom ($P13$) can change the algorithmic problem solved.

Similarly, Principle $P11$, optimizing the expected case, can sometimes help decide what the right measure is to optimize. This in turn influences the choice of algorithm. For example, simple TCP Header prediction (Chapter 9) optimizes the expected case when the next packet is from the same connection and is the next data packet or ack. If this is indeed the expected case, there is no need for fancy connection lookup structures (a simple one element cache) or fancy structures to deal with sequence number bookkeeping. However, if there are several concurrent connections as in a server, a hash table may be better for connection lookup. Similarly, if packets routinely arrive out of order, then more fancy sequence number bookkeeping schemes [TVHS92] may be needed.

Principle $P12$, adding state for speed, is a simple technique used often in standard algorithmic design. However, it is quite common for just this principle by itself (without fancy additional algorithmic machinery) to help remove systems bottlenecks. For example, the major bottleneck in the `select()` call implementation is the need to repeatedly check for data in
network connections known not to be ready. By simply keeping state across calls, this key bottleneck can be removed. By contrast, the bottlenecks caused by the bitmap interface can be removed by algorithmic means, but are less important.

Having framed the appropriate problem using \textbf{P11}, \textbf{P12}, and \textbf{P13}, principles \textbf{P14} and \textbf{P15} can be used to guide the search for solutions.

Principle \textbf{P14} asks whether there are any important special cases, such as the use of finite universes, that can be leveraged to derive a more efficient algorithm. For example, the McKenney buffer stealing algorithm of Chapter 9 provides a fast heap with $O(1)$ operations for the special case when elements to the heap change by at most 1 on each call.

Finally, principle \textbf{P15} asks whether there are algorithmic methods that can be adapted to the system. It is dangerous to blindly adapt existing algorithms because of the following possibilities that can mislead the designer.

- \textit{Wrong Measures:} The measure for most systems implementations is the number of memory accesses and not the number of operations. For example, the fast `ufalloc()` operation uses a selection tree on bitmaps instead of a standard heap, leveraging off the fact that a single read can access $W$ bits, where $W$ is the size of a word. Again, the important measure in many IP lookup algorithms is search speed and not update speed.

- \textit{Asymptotic Complexity:} Asymptotic complexity hides constants which are crucial in systems. When every microsecond counts, surely constant factors are important. Thus the switch matching algorithms in Chapter 13 have much smaller constants than the best bipartite matching algorithms in the literature, and hence can be implemented.

- \textit{Incorrect cost penalties:} In timing wheels (Chapter 7), a priority heap is implemented using a bucket-sorting data structure. However, the cost of strolling through empty buckets, a severe cost in bucket sort, is unimportant because on every timer tick, the system clock must be incremented anyway. As a second example, the dynamic programming algorithm to compute optimal lookup strides for multibit tries (Chapter 11) is $O(N \times W^2)$ where $N$ is the number of prefixes and $W$ is the address width. While this appears to be quadratic, it is linear in the important term $N$ (100,000 or more) and quadratic in the address width (32 bits, and the term is smaller in practice).

In spite of all these warnings, algorithmic methods are useful in networking ranging from the use of Pathfinder like tries in Chapter 8 to the use of tries and binary search (suitably modified) in Chapter 11.
18.3 Internet Algorithmics and Real Products

Many of the algorithms used in this book are used in real products. The following is a quick survey.

Endnode algorithms: Zero-copy implementations of network stacks are quite common as are implementations of memory mapped files; however more drastic changes such as IOLite are only used more rarely such as by the iMimic server software. The RDMA specification is well developed. Event driven web servers are quite common, and many operating systems other than UNIX (such as Windows NT) have fast implementations of select() equivalents. The VIA standard avoids system calls using ideas similar to ADCs.

Most commercial systems for early demultiplexing still rely on BPF but that is because few systems require so many classifiers that they need the scalability of a Pathfinder or a DPF. Some operating systems use timing wheels, notably Linux and FreeBSD. Linux uses fast buffer manipulation operations on linear buffers. Fast IP checksum algorithms are common, and so are multibit CRC algorithms in hardware.

Router Algorithms: Binary search lookup algorithms for bridges were common in products as were hashing schemes (e.g., Gigaswitch). Multibit trie algorithms for IP lookups are very common; recently compressed versions such as the Tree bitmap algorithm have become popular. Classification is still generally done by CAMs and thus much of Chapter 12 is probably more useful for software classification.

In some chapters like the chapter on switching (Chapter 13), we provided a real product example for every switching scheme described (see Figure 13.1 for example). In fair queuing, DRR, RED, and token buckets are commonly implemented. General weighted fair queuing, virtual clock, and core stateless fair queuing are hardly ever used. Finally, much of the measurement and security chapters are devoted to ideas that are not part of any product today.

It is useful to see many of these ideas come together in a complete system. While it is hard to find details of such systems (because of commercial secrecy), the following two large systems pull together ideas in endnode and router algorithmics.

System Example 1, Flash Web Server: The Flash[PDZ99a] web server was designed at Rice University and undoubtedly served as the inspiration (and initial code base) for a company called iMimic. A version of Flash called Flash-lite uses the following ideas from endnode algorithmics:

- Fast copies: Flash-lite uses IOLite to avoid redundant copies.
• **Process scheduling:** Flash uses an event driven server with helper processes to minimize scheduling and maximize concurrency.

• **Fast select:** Flash uses an optimized implementation of the `select()` call.

• **Other optimizations:** Flash caches response headers and file mappings.

**System Example 2 Cisco 12000 GSR Router:** The Cisco GSR[Sys] is a popular gigabit router and uses the following ideas from router algorithmics:

• **Fast IP lookups:** It appears that the GSR uses a multibit tree to do IP lookups.

• **Fast switching:** The GSR uses the iSLIP algorithm for fast bipartite matching of VOQs.

• **Fair queuing:** The GSR implements a modified form of DRR called MDRR where queue is given priority (e.g., for Voice-over-IP). It also implements a sophisticated form of RED called weighted RED and token buckets. All these algorithms are implemented in hardware.

### 18.4 The future of Internet Algorithmics

The last three sections of this chapter talked of the past and the present. But are all the ideas played out? Has Internet Algorithmics already been milked to the point that there is nothing new left to do? We believe this is not the case. This is because we believe Internet Algorithmics will be enriched in the near future in three ways: new *abstractions* that require new solutions will become popular; new *connecting disciplines* will provide new approaches to existing problems; and new *requirements* will require rethinking existing solutions. We expand on each of these possibilities in turn.

#### 18.4.1 New abstractions

This book dealt with the fast implementation of the standard networking abstractions: TCP sockets at endnodes, and IP routing at routers. However, new abstractions are constantly being invented to increase user productivity. While these abstractions make life easier for users, unoptimized implementations of these abstractions can exact a severe performance penalty. But this only creates new opportunities for Internet Algorithmics. Some examples of such abstractions are:
• **TCP Offload Engines:** While the book has concentrated on software TCP implementations, movements like iSCSI have made hardware TCP Offload Engines more interesting. Doing TCP in hardware and handling worst-case performance at 10 Gbps and even 40 Gbps is very challenging. For example, to do complete offload, the chip must even handle out-of-order packets and packet fragments (see Chapter 9 without appreciable slowdown).

• **HTML and web server processing:** There have been a number of papers trying to improve web server performance that can be considered to be an application of Endnode Algorithmics. For example, persistent HTTP [Mog95] can be considered an application of P1 to the problem of connection overhead. A more speculative approach to reduce DNS lookup times in web accesses by passing hints (P10) is described in [CV01].

• **Web Services:** The notion of web services, by which a web page is used to provide a service, is getting increasingly popular. There are a number of protocols that underly Web Services, and standard implementations of these services can be slow.

• **CORBA:** The Common Object Request Broker Architecture is popular but quite slow. Gokhale and Schmidt [GS98] apply four of the principles described in this book (eliminating waste P1, optimizing the expected case P11, passing information between layers P9 and exploiting locality for good cache behavior P4a) to the problem. They show that such techniques from endnode algorithmics can improve the performance of the SunSoft Inter-Orb protocol by a factor of 2 to 4.5, depending on the data type. Similar optimizations should be possible in hardware.

• **SSL and other encryption standards:** Many web servers use the Secure Socket Layer (SSL) for secure transactions. Software implementations of SSL are quite slow. There is an increasing interest in hardware implementations of SSL.

• **XML Processing:** XML is rapidly becoming the lingua franca of the web. Parsing and converting from XML to HTML can be a bottleneck.

• **Measurement and Security abstractions:** Currently, SNMP and NetFlow allow very primitive measurement abstractions. The abstraction level can only be raised by a tool that has integrates all the raw measurement data. Perhaps in the future routers will have to implement more sophisticated abstractions to help in measurement and security analysis.

• **Sensor Networks:** A sensor network may wish to calculate new abstractions to solve such specific problems as to find high concentrations of pollutants or the direction of a forest fire.
If history is any guide, every time an existing bottleneck becomes well studied, a new abstraction appears with a new bottleneck. Thus after lookups became well understood, packet classification emerged. After classification, came TCP Offload; and now SSL and XML are clearly important. Many pundits believe that wire speed security solutions (as implemented in a router or an Intrusion Detection System) will be required by the year 1996. Thus it seems clear that future abstractions will keep presenting new challenges to Internet Algorithmics.

18.4.2 New connecting Disciplines

Earlier, we said that a key aspect of Internet Algorithmics was its interdisciplinary nature. Solutions require a knowledge of operating systems, computer architecture, hardware design, networking, and algorithms. We believe the following disciplines will also impinge on Internet Algorithmics very soon:

- **Optics**: Optics has been abstracted away as a link layer technology in this book. Currently, optics provides a way to add extra channels to existing fiber using Dense Wavelength Division Multiplexing. However, optical research has made amazing strides. There are undoubtedly exciting possibilities to rethink router design using some combination of electronics and optics.\(^2\)

- **Network Processor Architecture**: While this field is still in its infancy compared to computer architecture, there are surely more imaginative approaches than current approaches using that assign packets to one of several processors. One such approach described in [SVC03] uses a wide word state machine as a fundamental building block.

- **Learning Theory**: The fields of security and measurement are crying out for techniques to pick out interesting patterns from massive traffic data sets. Learning theory and data mining have been used for these purposes in other fields. Rather than simply reusing say standard clustering algorithms or standard techniques such as Hidden Markov Models, the real breakthroughs may belong to those who can find variations of these techniques that can be implemented at high speeds with some loss of accuracy. Similarly, OLAP tools may be useful for networking with twists to fit the networking milieu. An example of a tool that has an OLAP flavor in a uniquely network setting can be found in [ESV03].

- **Databases**: The field of databases has a great deal to teach networking in terms of systematic techniques for querying for information. Recently, an even more relevant trend

\(^2\)Electronics still appears to be required today because of the lack of optical buffers and the difficulty of optical header processing.
has been the subarea of continuous queries. Techniques developed in databases can be of great utility to algorithmics.

- **Statistics**: The field of statistics will be of even more importance in dealing with large data sets. Already, NetFlow and other tools have to resort to sampling. What inferences can safely be made from sampled data. As we have seen in Chapter 16, statistical methods are already used by ISPs to solve the traffic matrix problem from limited SNMP data.

### 18.4.3 New requirements

Much of this book has focused on processing time as the main metric to be optimized, while minimizing dollar cost. Storage was also an important consideration because of limited on-chip storage, and the expense of SRAM. However, even minimizing storage was related to speed, in order to maximize the possibility of storing the entire data structure in high-speed storage.

The future may bring new requirements. Two important such requirements are (mechanical) space and power. Space is particularly important in PoPs and hosting centers, because rack space is limited. Thus routers with small form factors are crucial. It may be that optimizing space is mostly a matter of mechanical design together with the use of higher and higher levels of integration. However, engineering routers (and individual sensors in sensor networks) for power may require attention to algorithmics.

Today power per rack is limited to a few kilo watts, and routers that need more power do so by spreading out across multiple racks. Power is a major problem in modern router design. It may be possible to rethink lookup, switching, and fair queuing algorithms in order to minimize power. Such power-conscious designs have already appeared in the computer architecture and operating systems community. It is logical to expect this trend to spread to router design.

### 18.5 Conclusions

We have tried to summarize in this chapter the major themes of this book in terms of the techniques described and the principles used. We have also tried to argue that Internet Algorithmics is used in real products, and is likely to be find further application in the future because of new abstractions, new connecting disciplines, and new requirements. While the specific techniques and problems may change, we hope the principles used remain useful.

Besides the fact that Internet Algorithmics is useful in building better and faster Internet devices, we hope this book also makes the case that Internet Algorithmics is also intellectually
stimulating. While it may lack the depth of hard problems in theoretical computer science or physics, perhaps what can be most stimulating is the breadth in terms of the disciplines it encompasses.

An endnode, for instance, may appear as a simple processing state machine at the highest level of abstraction. A more detailed inspection would see a web request packet arriving at a server interface, the interrupt firing, and the protocol code being scheduled via a software interrupt. Even within the protocol code, each line of code has to be fetched, hopefully from the i-cache and each data item has to go through the VM system (via the TLB hopefully) and the data cache. Finally, the application must get involved via a returned system call and a process scheduling operation. The request may trigger file system activity and disk activity.

A router similarly has an interesting inner life. Reflecting the macrocosmos of the Internet outside the router, is a microcosmos within the router consisting of major subsystems such as line cards and the switch fabric, together with striping and flow control across chip-to-chip links.

Internet Algorithmics seeks to understand these hidden subsystems of the Internet to make the Internet faster. This book is a first attempt to begin understanding — in Feynman’s phrase — this “tremendous world of interconnected hierarchies” within routers and endnodes. In furthering this process of understanding and streamlining these hierarchies, there are still home runs to be hit, and touchdowns to be scored. We hope you, dear reader, will join the game.