

- [Preface](#)
- [Foundation](#)
 - [Problem: Building a Network](#)
 - [Motivation](#)
 - [Requirements](#)
 - [Network Architecture](#)
 - [Plan for the Book](#)
 - [Summary](#)
 - [Open Issue: Ubiquitous Networking](#)
 - [Further Reading](#)
 - [Exercises](#)
- [Protocol Implementation](#)
 - [Problem: Protocols Have to be Implemented](#)
 - [Object-Based Protocol Implementation](#)
 - [Protocols and Sessions](#)
 - [Messages](#)
 - [Events](#)
 - [Id Map](#)
 - [Example Protocol](#)
 - [Summary](#)
 - [Open Issue: Cost of Network Software](#)
 - [Further Reading](#)
 - [Exercises](#)
- [Direct Link Networks](#)
 - [Problem: Physically Connecting Hosts](#)
 - [Hardware Building Blocks](#)
 - [Encoding \(NRZ, NRZI, Manchester, 4B/5B\)](#)
 - [Framing](#)
 - [Error Detection](#)

- [Reliable Transmission](#)
- [CSMA/CD \(Ethernet\)](#)
- [Token Rings \(FDDI\)](#)
- [Network Adaptors](#)
- [Summary](#)
- [Open Issue: Does it belong in hardware?](#)
- [Further Reading](#)
- [Exercises](#)
- [Packet Switching](#)
 - [Problem: Not All Machines Are Directly Connected](#)
 - [Switching and Forwarding](#)
 - [Routing](#)
 - [Cell Switching \(ATM\)](#)
 - [Switching Hardware](#)
 - [Summary](#)
 - [Open Issue: What Makes a Route Good?](#)
 - [Further Reading](#)
 - [Exercises](#)
- [Internetworking](#)
 - [Problem: There Is More than One Network](#)
 - [Bridges and Extended LANs](#)
 - [Simple Internetworking \(IP\)](#)
 - [Global Internet](#)
 - [Next Generation IP](#)
 - [Multicast](#)
 - [Host Names \(DNS\)](#)
 - [Summary](#)
 - [Open Issue: IP versus ATM](#)
 - [Further Reading](#)
 - [Exercises](#)
- [End-to-End Protocols](#)
 - [Problem: Getting Processes to Communicate](#)
 - [Simple Demultiplexor \(UDP\)](#)
 - [Reliable Byte-Stream \(TCP\)](#)
 - [Remote Procedure Call](#)
 - [Application Programming Interface](#)
 - [Performance](#)
 - [Summary](#)

- [Open Issue: Application-Specific Protocols](#)
 - [Further Reading](#)
 - [Exercises](#)
- [End-to-End Data](#)
 - [Problem: What do we do with the data?](#)
 - [Presentation Formatting](#)
 - [Data Compression](#)
 - [Security](#)
 - [Summary](#)
 - [Open Issue: Presentation---Layer of the 90's](#)
 - [Further Reading](#)
 - [Exercises](#)
- [Congestion Control](#)
 - [Problem: Allocating Resources](#)
 - [Issues](#)
 - [Queuing Disciplines](#)
 - [TCP Congestion Control](#)
 - [Congestion Avoidance Mechanisms](#)
 - [Virtual Clock](#)
 - [Summary](#)
 - [Open Issue: Inside versus Outside the Network](#)
 - [Further Reading](#)
 - [Exercises](#)
- [High-Speed Networking](#)
 - [Problem: What Breaks When We Go Faster?](#)
 - [Latency Issues](#)
 - [Throughput Issues](#)
 - [Integrated Services](#)
 - [Summary](#)
 - [Open Issue: Realizing the Future](#)
 - [Further Reading](#)
 - [Exercises](#)
- [Appendix: Network Management](#)
 - [SNMP Overview](#)
 - [MIB Variables](#)
- [Glossary](#)
- [References](#)
- [About this document ...](#)

Copyright 1996, Morgan Kaufmann Publishers

Preface

There is little need to motivate computer networking as one of the most exciting computer science topics in recent years; the popular media has already done a more than adequate job of that. Examples of the rapid spread of networking are plentiful: USA Today regularly publishes the address of interesting World Wide Web sites, Americans are encouraged to send their comments to NBC Nightly News via Internet mail, both the White House and Congress, are ``on line" (as is Her Majesty's Treasury), and perhaps most impressively, it is now possible to order pizza over the Internet. Surely any technology that receives this much attention must be worth understanding.

The concern, of course, is that there might be little or no intellectual substance once one gets beyond the buzzwords. Fortunately, this is not the case. When one peels away all the hype, the field of computer networking contains a wealth of interesting problems and clever solutions founded on solid underlying principles. The challenge is to identify and understand these concepts and principles. That is why we decided to write this book. Our intent is that the book should serve as the text for an introductory networking class, at either the graduate, or upper division undergraduate level. We also believe that its focus on core concepts should be appealing to professionals in industry who are retraining for network-related assignments, as well as current network practitioners who want to understand the ``whys" behind the protocols they work with every day.

Approach

It is our experience that people learning about networks for the first time often have the impression that network protocols are some sort of edict handed down from on high, and that their job is to learn as many TLAs (Three Letter Acronyms) as possible. In fact, protocols are the building blocks of a complex system developed through the application of engineering design principles. Moreover, they are constantly being refined, extended, and replaced based on real-world experience. With this in mind, our goal with this book is to do more than survey the protocols in use today. Instead, we try to motivate and explain the underlying principles of sound network design.

This goal implies several things about our approach. Perhaps most importantly, the book takes a systems-oriented view of computer networking. What we mean by this is that the book identifies the underlying building blocks, shows how they can be glued together to construct a complete network, measures the

resulting system experimentally, and use this data to quantitatively analyze various design options. This is in contrast to looking at networking through the eyes of a standardization body, a telephone company executive, or a queuing theorist.





In keeping with our systems perspective, this book explains *why* networks are designed the way they are. It does this by starting from first principles, rather than by simply marching through the set of protocols that happen to be in use today. Moreover, we do not adhere a rigidly layerist point-of-view, but instead allow ourselves to address issues that cannot be neatly packaged in any single layer. It is our experience that once someone understands the underlying concepts, any new protocol that he or she is confronted with---especially one that cannot be pigeon-holed in a existing layer---will be relatively easy to digest. Given the volatility and explosive growth of networking in the 1990's, this focus on principles rather than artifacts seems well justified.

Finally, the book does two things to help make these underlying concepts more tangible. First, it draws heavily from the lessons of the Internet. As a functioning, global network, the Internet is rich with examples that illustrate how networks are designed in practice. This book exploits the Internet experience in several ways: (1) as a roadmap that guides the organization of this book, (2) as a source of examples that illustrate the fundamental ideas underlying computer networks, and (3) as a set of design principles that color our way of looking at computer networks. Second, the book includes code segments from a working network subsystem---the *x*-kernel. The book is not an exhaustive inspection of the code, but rather, snippets of *x*-kernel code are given throughout the book to help make the discussion more concrete. The basic strategy is to provide the reader with not only an understanding of the components of a network, but also a feel for how all the pieces are put together.

How to Use This Book

The chapters of this book can be organized into three groups:

- Introductory material: Chapters 1--2.
- Core topics: Chapters 3--6.
- Advanced topics: Chapters 7--9.

For an undergraduate course, extra class-time will most likely be needed to help students digest the introductory material in the first two chapters, probably at the expense of the more advanced topics covered in Chapters 7 through 9. In contrast, the instructor for a graduate course should be able to cover the first two chapters in only a few lectures---with students studying the material more carefully on their own---thereby freeing up additional class time to cover the last three chapters in depth. Both graduate and undergraduate classes will want to cover the core material contained in the middle four chapters, although an undergraduate class might choose to skim the more advanced sections (e.g., , , , and )

For people using the book in self-study, we believe the topics we have selected cover the core of computer networking, and so recommend that the book be read sequentially, from front to back. In addition, we have included a liberal supply of references to help the reader locate supplementary material that is relevant to their specific areas of interest.

There are several other things to note about the topic selection and organization. First, the book takes a unique approach to the topic of congestion control by pulling all congestion control and resource allocation related topics together in a single place---Chapter 8. We do this because the problem of congestion control cannot be solved at any one single level, and we want the reader to consider the various design options at the same time. (This is consistent with our view that strict layering often obscures important design tradeoffs.) A more traditional treatment of congestion control is possible, however, by studying Section [\[1\]](#) in the context of Chapter 4, and Section [\[2\]](#) in the context of Chapter 6.


Second, even though Chapter 9 is the only chapter explicitly labeled *High-Speed Networking*, the topic is covered in substantial depth throughout the book. For example, Section [\[3\]](#) discusses the role played by network adaptors in delivering high throughput, [\[4\]](#) describes ATM networks, [\[5\]](#) describes hardware switches used in high-speed networks, [\[6\]](#) discusses MPEG video compression, [\[7\]](#) discusses resource reservation, and video applications are used throughout the book to motivate various design decisions. In short, high-speed networking is becoming commonplace, and we believe it no longer deserves to be treated purely as an advanced topic.

Third, example code from the *x*-kernel is used throughout the book to illustrate important concepts, with an overview of the *x*-kernel given in Chapter 2. How much time one spends studying Chapter 2 depends, of course, on how one plans to use the *x*-kernel throughout the rest of the book. We envision the code being used at one of three levels.

- The code can be *ignored*. Each of the chapters contains a complete discussion of the topic at hand; the code simply punctuates this discussion with tangible evidence that the ideas being presented can be put to practice.
- The code can be *read*. While this certainly involves some effort, the investment is relatively low (the *x*-kernel was explicitly designed to support network software and so has a high "signal-to-noise" ratio) and the payoff is real (the reader gets to see first hand how the underlying ideas are realized in practice).
- The code can be *written*. That is, because the complete code is available on-line, it can be used to provide hands-on experience. Some of the exercises at the end of each chapter suggest programming assignments that the reader can do using the *x*-kernel.

Additional information on how (and why) to use the *x*-kernel is given on page XXX.

Finally, the book has several features that we encourage reader to take advantage of.

- Each chapter begins with a *Problem Statement* that identifies the next set of issues that must be addressed in the design of a network. These problem statements both motivate the chapter that follows, and outlines individual topics covered in that chapter.
- *Out-of-band boxes* are used throughout the text to elaborate on the topic being discussed or to introduce a related advanced topic. In many cases, these boxes relate stories about how networking works in the real-world.
- Key paragraphs are *highlighted*. These paragraphs often summarize an important nugget of information that we want the reader to take away from the discussion. In many cases, this nugget is a system design principle that the reader will see applied in many different situations.
- As part of the Further Reading section at the end of each chapter, a specific *recommended reading list* is given. This list generally contains the seminal papers on the topics discussed in the chapter. We strongly recommend that advanced readers (e.g., graduate students) study the papers in this reading list to supplement the material covered in the chapter.
- Even though the book's focus is on core concepts rather than existing protocol specifications, real protocols are used to illustrate most of the important ideas. As a result, the book can be used as a source of reference for these example protocols. To help the reader find the descriptions of these protocols, the section headings given in the table of contents parenthetically identify the protocols defined in that section. For example, Section , which describes the principles of reliable end-to-end protocols, provides a detailed description of TCP, the canonical example of such a protocol.
- We conclude each chapter with an *Open Issue* section. The idea of this section is to expose the reader to various unresolved issues that are currently being debated in the research community, the commercial world, and/or society as a whole. We have found that discussing these issues helps to make the subject of networking more relevant and more exciting.

Software

As mentioned above, this book uses the *x*-kernel to illustrate how network software is implemented. The *x*-kernel was designed to support the rapid implementation of efficient network protocols. A protocol implementation can be done rapidly because the *x*-kernel provides a set of high-level abstractions that are tailored specifically to support protocol implementations. *x*-kernel protocols are efficient because these abstractions are themselves implemented using highly optimized data structures and algorithms. In fact, the world speed record for delivering network data from an ATM network into a desktop workstation---516Mbps---was held by the *x*-kernel at the time of writing.

Because the *x*-kernel has the dual goal of running fast and making it easy to implement protocols, it has also proven to be an invaluable pedagogical tool, one that is used extensively in the *Principles of Computer Networking* class at the University of Arizona. There are two aspects to the pedagogical value of the *x*-kernel. First, because the *x*-kernel defines a concise framework for implementing protocols, it can be viewed as simply codifying the essential elements of network protocols. An intimate understanding of the interfaces provided by the *x*-kernel necessarily means appreciating the mechanisms found in all protocols. Second, it provides a convenient means for students to get hands-on experience with network protocols---students are able both to implement protocols and to evaluate their performance.

On the subject of ``hands-on experience'', many of the exercises at the end of each chapter suggest programming assignments that can be done, some (but not all) of which use the *x*-kernel. Typically, these assignments take on one of the following four forms:

- Write a small program that tests or measures some narrow aspect of a network system. These assignments are independent of the *x*-kernel.
- Run and measure existing protocols currently distributed with the *x*-kernel. Sometimes these assignments involve making simple modifications to existing *x*-kernel code.
- Implement and evaluate standard protocols based on existing protocol specifications. These assignments use the *x*-kernel as a protocol implementation framework.
- Design, implement, and evaluate new protocols from scratch. These assignments use the *x*-kernel as a protocol implementation framework.

For these assignments, students typically run the *x*-kernel as a user process on Unix workstations connected by an Ethernet. Several flavors of Unix are currently supported, including SunOS, Solaris, OSF/1, IRIX, and Linux. In this case, the *x*-kernel generates packets that are delivered between machines over the Ethernet. An alternative is to run the *x*-kernel in *simulation mode*, again on top of Unix. In this case, networks with arbitrary topology, bandwidth, and delay characteristics can be simulated as a single Unix process.

The *x*-kernel source code is available by doing anonymous ftp to `cs.arizona.edu`; cd'ing to directory `xkernel` and getting file `xkernel.tar.Z`. You can also find the *x*-kernel home page on the Web at URL

<http://www.cs.arizona.edu/xkernel/www>

This page includes a link to an on-line programmer's manual, as well as instructions on how to obtain help with the *x*-kernel. Additional information about the *x*-kernel and how it can be used in conjunction with this book can be found at URL

Acknowledgements

This book would not have been possible without the help of a lot of people. We would like to thank them for their efforts.

First and foremost, we would like to thank our series editor, **Dave Clark**. His involvement in the series played no small part in our decision to write this book with Morgan-Kaufmann, and the suggestions, expertise, perspective, and encouragement he offered us throughout the writing process proved invaluable.

Several members of the Network Systems Research Group at the University of Arizona contributed ideas, examples, corrections, data, and code to this book. **Andy Bavier** fleshed out many of the exercises and examples, and offered helpful suggestions on the clarity of the early drafts. **Lawrence Brakmo** answered countless questions about TCP, and provided the data used in the figures on TCP congestion control. **David Mosberger** served as the resident expert on MPEG and JPEG, and ran many of the performance experiments reported in the book. Andy, Lawrence, and David also wrote most of the *x*-kernel code fragments that ended up in the book. **Richard Schroepel**, **Hilarie Orman**, and **Sean O'Malley** were a valuable resource on network security, and supplied the security-related performance numbers. Sean also helped us organize our thoughts on presentation formatting. Finally, countless people have contributed to the *x*-kernel over the years. We would especially like to thank (in addition to those mentioned above) **Ed Menze**, **Sandy Miller**, **Hasnain Karampurwala**, **Mats Bjorkman**, **Peter Druschel**, **Erich Nahum**, and **Norm Hutchinson**.

Several other people were kind enough to answer our questions and correct or misunderstandings. **Rich Salz** and **Franco Travostino** from OSF taught us about NDR. **Jeffrey Mogul** from Digital's Western Research Lab and **John Wroclawski** from MIT helped us understand how fast workstations can switch packets. **Gregg Townsend** from the University of Arizona explained how GIF works. **Gail Lalk** and **Tom Robe** of Bellcore answered questions about SONET.

We would also like to thank the many people that reviewed drafts of various chapters. The list is long, and includes **Chris Edmondson-Yurkanan** of the University of Texas at Austin, **Willis Marti** of Texas A&M University, **Frank Young** of Rose-Hulman Institute of Technology, **Tom Robertazzi** of SUNY Stony Brook, **Jon Crowcroft** of University College London, **Eric Crawley** of Bay Networks, **Jim Metzler** of Strategic Networks Consulting, **Dan McDonald** of the Navy Research Lab, **Mark Abbott** of IBM, **Tony Bogovic** of Bellcore, **David Hayes** of San Jose State University, **S. Ron Oliver** of California State Polytechnic University-Pomona, **Donald Carpenter** of the University of Nebraska at Kearney, and **Ski Ilnicki** of Hewlett-Packard. Their comments and suggestions were invaluable. Also, a special thanks also to Professors **Peter Druschel** of Rice University, **Izidor Gertner** of CCNY, and **Mansoor Alam** of the University of Toledo for their feedback based on class-testing the first draft of the book.

While we have done our best to correct the mistakes that the reviewers have pointed out, and to accurately describe the protocols and mechanisms that our colleagues have explained to us, we alone are responsible for any errors in this book.

We also want to thank the **Advanced Research Projects Agency** and the **National Science Foundation** for supporting our research over the past several years. That research helped to shape our perspective on networking, as well as produced tangible results like the x -kernel. Thanks also to **Bell Communications Research** which supported both some of our research and some of the writing.

We would like to thank all the people at Morgan-Kaufmann that helped shepherd us through the book-writing process, especially our sponsoring editor **Jennifer Mann**, our production editor **Julie Pabst**, and our production manager **Yonie Overton**. We cannot imagine a more enthusiastic or harder working group of people.

Finally, we wish to thank our long-suffering wives, **Lynn Peterson** and **Jody Davie** who put up with our complete obsession with the book (and consequent neglect of our families), especially in the final months of writing.

Next	Up	Previous
----------------------	--------------------	--------------------------

Next: [Foundation](#) **Up:** [Computer Networks: A Systems](#) **Previous:** [Computer Networks: A Systems](#)

Foundation

-
- [Problem: Building a Network](#)
 - [Motivation](#)
 - [Requirements](#)
 - [Network Architecture](#)
 - [Plan for the Book](#)
 - [Summary](#)
 - [Open Issue: Ubiquitous Networking](#)
 - [Further Reading](#)
 - [Exercises](#)
-

Problem: Building a Network

Suppose you want to build a computer network, one that has the potential to grow to global proportions and support applications as diverse as teleconferencing, video-on-demand, distributed computing, and digital libraries. What available technologies would serve as the underlying building blocks, and what kind of software architecture would you design to integrate these building blocks into an effective communication service? Answering this question is the overriding goal of this book---to describe the available building materials and show how they can be used to construct a network from the ground up.

Before we can understand how to design a computer network, we should first agree on exactly what a computer network is. At one time, a network referred to the set of serial lines used to attach dumb terminals to mainframe computers. To some, the term network implies the voice telephone network. To still others, the only interesting network is the cable network used to disseminate video signals. The main thing these networks have in common is that they are specialized to handle one particular kind of data (key strokes, voice, and video), and they typically connect to special-purpose devices (terminals, hand receivers, and television sets).

What distinguishes a computer network from these other types of networks? Probably the most important characteristic of a computer network is its generality. Computer networks are built primarily from general-purpose programmable hardware, and they are not optimized for a particular application like making phone calls or delivering television signals. Instead, they are able to carry many different types of data and they support a wide, and ever-growing, range of applications. This chapter looks at some typical applications of computer networks, and then discusses the requirements of which a network designer who wishes to support such applications must be aware.

Once we understand the requirements, then how do we proceed? Fortunately, we will not be building the first network. Others, most notably the community of researchers responsible for the Internet, have gone before us. We will use the wealth of experience generated from the Internet to guide our design. This experience is embodied in a *network architecture* that identifies the available hardware and software components, and shows how they can be arranged to form a complete network system.

To start us on the road towards understanding how to build a network, this chapter does three things. First, it explores the motivation for building networks, which is primarily to support network applications. Second, it looks at the requirements that different applications and different communities


of people (such as network users and network operators) place on the network. Finally, it introduces the idea of a network architecture, which lays the foundation for the rest of the book.

[Next](#) [Up](#) [Previous](#)


Next: [Motivation](#) **Up:** [Foundation](#) **Previous:** [Foundation](#)

Copyright 1996, Morgan Kaufmann Publishers

Motivation

It has become increasingly difficult to find people who have not made some use of computer networks, whether to send electronic mail (email), to transfer files, or to ``surf the net" using a World Wide Web browser like Mosaic.  However, just as the vast majority of today's computer users have never written a computer program, the majority of network users know very little of the technology that underlies computer networks. This book explains computer networking technology from a systems perspective, which is to say, it explains it in a way that will teach the reader how networks are implemented---what sort of hardware is used, how it works, and how one implements the software that turns the hardware components into a useful network. To begin, we consider some of the factors that make computer networking an exciting and worthwhile field of study.

Explosive Growth

While catchy phrases like ``information super highway" help to sell the general public on the idea of networking, computer networks like the Internet are a reality today, and have been for over 20 years. In 1989, it was estimated that the Internet connected approximately 100,000 computers. By the end of 1992, the number had reached 1 million. Today, there are over 6 million computers on the Internet, giving an estimated 30 to 50 million users access to its services. If traced back to its origins, the Internet has been doubling in size every year since 1981. Figure  plots the number of computers connected Internet since 1988.

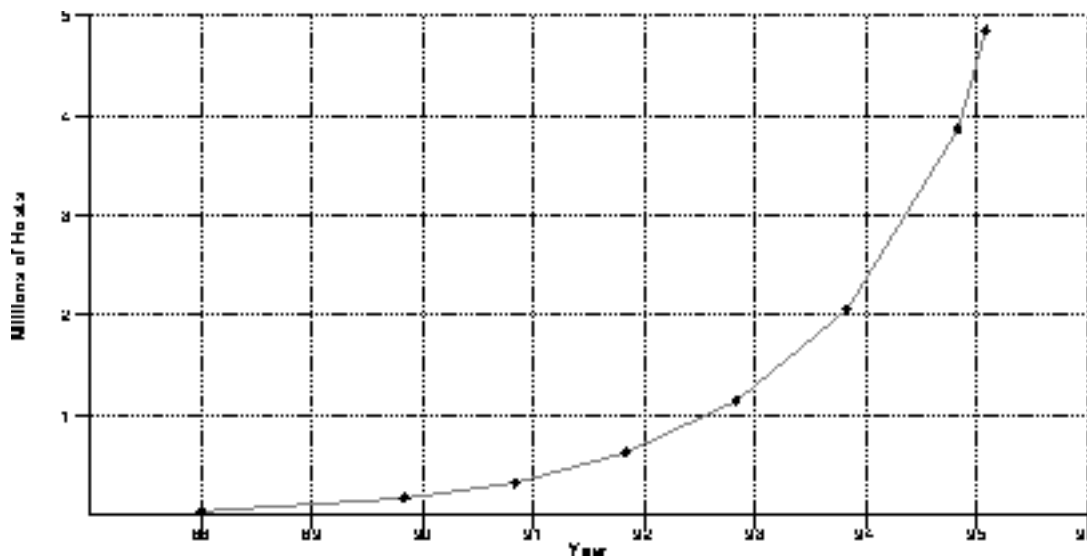


Figure: Growth in the number of computers connected to the Internet since 1988.

One of the things that has made the Internet such a runaway success is the fact that so much of its functionality is provided by software running in general purpose computers. The significance of this is that new functionality can be added readily with just a "small matter of programming". As a result, we could not even begin to list all the new applications that have appeared on the Internet in recent years. A few of these applications have been overwhelmingly popular, with the Word Wide Web being a particularly notable example.

Adding new functionality to the network is not limited to the creation of new applications, but may also include new functions "inside" the network. A good example of the latter is multicast routing, which enables multi-party teleconferencing to take place over the Internet. Our goal in this book is to teach enough about the implementation of networks that the reader will be able not only to understand the functionality of computer networks but also to expand it.

Another factor that has made computer networking so exciting in recent years is the massive increase in computing power available in commodity machines. While computer networks have always been capable in principle of transporting any sort of information such as digital voice samples, digitized images, and so on, this was not a particularly interesting thing to do if the computers sending and receiving that data were too slow to do anything useful with the information. Virtually all of today's computers are capable of playing back digitized voice at full speed and can display video at a speed and resolution that is useful for some (but by no means all) applications. Thus, today's networks have begun to support multimedia, and their support for it will only improve as computing hardware becomes inexorably faster. Dealing with multimedia applications and with the demand for ever-faster networks presents some special challenges for network designers; this book will present some of those challenges and provide some tools for dealing with them.

Network Applications

Since the reason for having computer networks in the first place is so that users can run applications over them, it will be helpful to our subsequent discussion to have some understanding of common applications. Any function that is provided in a network can ultimately be traced back to some application need. Application builders take advantage of functions provided by the network, and network designers try to add new functionality to improve the performance of current applications and to enable new ones. The following describes some of the most common applications that motivate our discussion of networking functionality.

FTP: File Transfer Protocol

The name FTP is a little confusing because it can be interpreted to stand for either File Transfer *Program*, which is an application, or the File Transfer *Protocol*, which is a protocol. The latter definition is actually the correct one but for now we want to focus on applications. The FTP application is one of the oldest on the Internet, and continues to be useful. As the name suggests, its function is to enable the transfer of files between computers over a network. The operation of one version of the application is illustrated in the following example:

```
% ftp
ftp> open ds.internic.net
Connected to ds.internic.net.
[...]
220 ds.internic.net FTP server ready.
Name (ds.internic.net:bsd):anonymous
331 Guest login ok, send ident as password.
Password:
230 Guest login ok, access restrictions apply.
ftp> cd rfc
250 CWD command successful.
ftp> get rfc-index.txt
200 PORT command successful.
150 Opening ASCII mode data connection for rfc-index.txt (245544
bytes).
226 Transfer complete.
local: rfc-index.txt remote: rfc-index.txt
251257 bytes received in 1.2e+02 seconds (2 Kbytes/s)
ftp> close
221 Goodbye.
```


The above example shows an *anonymous* FTP to a machine called `ds.internic.net`. Anonymous FTP means that a user who does not have an account on the remote machine can still connect to it, by providing the user name `anonymous`. By convention, the user's email address is given as a password.

We began the FTP application by typing `ftp`. Then we opened a connection to the remote machine and responded to its request for a user name and password. At that point, the user can perform a range of operations related to retrieving files. For a full listing of commands, the user would type `help`. What we did above was to change to a directory called `rfc` using the `cd` command and then retrieve the file `rfc-index.txt` with the `get` command. Finally, we `closed` the connection.

You can try out the above example for yourself, provided you have access to an Internet-connected machine that runs FTP. The `rfc` directory actually contains all the Internet *Request For Comments* documents (RFCs), which provide a wealth of information about the Internet, from the highly technical to the humorous.

This example allows us to define an important pair of terms: *client* and *server*. In the example, we started an FTP client program on our own machine; it connected to the FTP server on `ds.internic.net`. The client then issued a series of requests to the server, which responded. The reader may find it helpful to think of the client as the entity that issues requests for a service, just as the client of a business is the person who asks the business for some service. In the current example, the service provided is the storage and retrieval of files.

World Wide Web

The World Wide Web has been so successful and has made the Internet accessible to so many people that sometimes it seems to be synonymous with the Internet. One helpful way to think of the Web is as a set of cooperating clients and servers all of whom speak the same language: the HyperText Transfer Protocol (HTTP). Most people are exposed to the Web through a graphical client program, or Web browser, like Mosaic, so we will talk about it from that perspective here. Figure  shows the Mosaic browser in use, displaying a page of information from the University of Arizona.

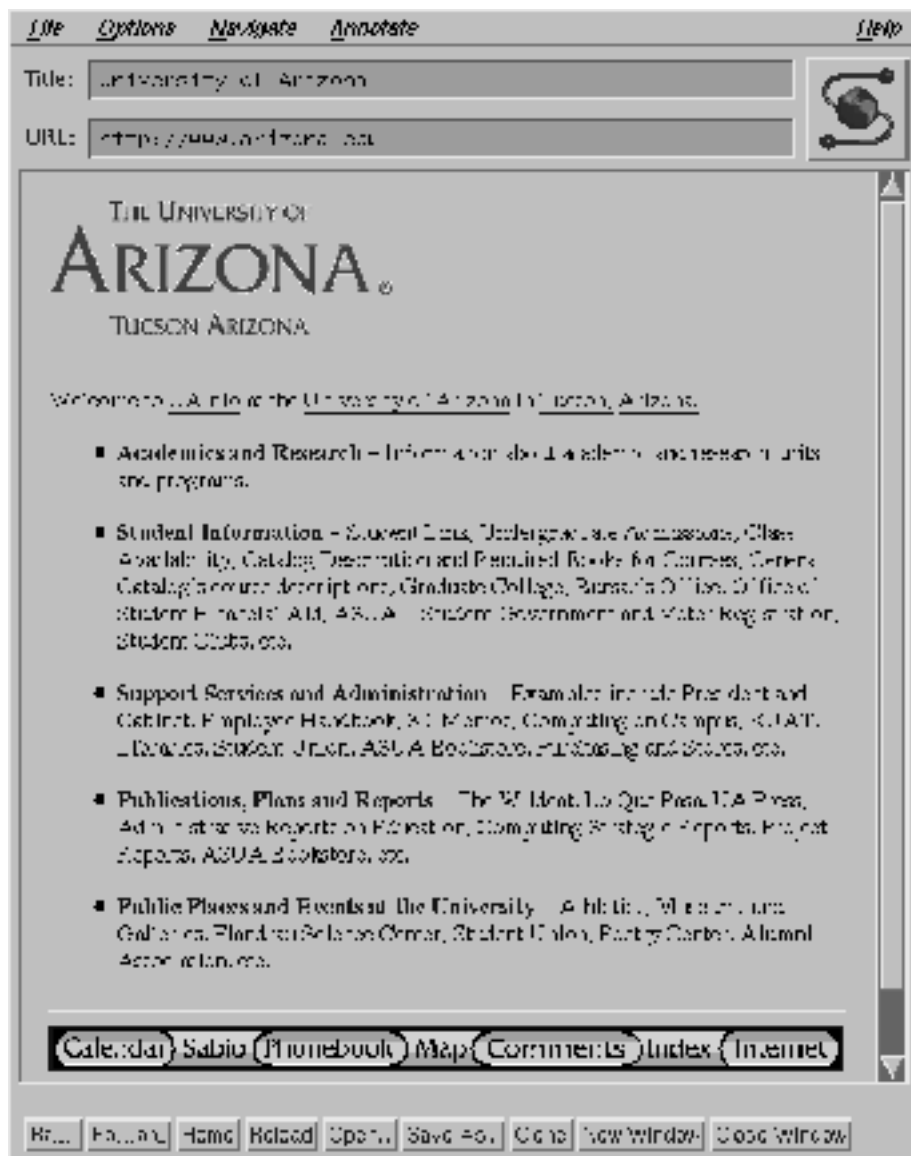


Figure: The Mosaic Web browser.

Any Web browser has a function that allows the user to 'open a URL'. URLs (Uniform Resource Locators) provide information about the location of objects on the Web; they look like the following:


`http://www.arizona.edu/index.html`

If you opened that particular URL, your Web browser would open a connection to the Web server at a machine called `www.arizona.edu` and immediately retrieve and display the file called `index.html`. Most files on the Web contain images and text, and some have audio and video clips; this multimedia flavor is one of the things that has made the Web so appealing to users. Most files also include URLs that point to other files, and your Web browser will have some way in which you can recognize URLs and ask the browser to open them. These embedded URLs are called hypertext links. When you ask your Web browser to open one of these embedded URLs (e.g. by pointing and clicking on it with a mouse), it will open a new connection and retrieve and display a new file. This is called "following a link". It thus becomes very easy to hop from one machine to another around the network, following links to all sorts of information.

It is interesting to note that much of what goes on when browsing the World Wide Web is the same as what goes on when retrieving files with FTP. Remote machines are identified, connections are made to those machines, and files are transferred. We will see that many seemingly disparate applications depend on similar underlying functionality, and that one of the network builder's jobs is to provide the right common functionality that many applications can use.

NV: Network Video

A third, rather different application is NV, which stands for Network Video. NV is used primarily for multiparty video-conferencing on the Internet. To be a sender of video, you need some special hardware, called a *frame grabber*, which takes the sequence of images output by a video camera and digitizes them so that they can be fed into a computer. The size of each image, which is called a *frame*, depends on the resolution of the picture. For example, an image one quarter the size of a standard TV image would have a resolution of 352 by 240 pixels; a pixel corresponds to one dot on a display. If each pixel is represented by 24 bits of information, as would be the case for 24-bit color, then each frame would be $(352 \times 240 \times 24)/8 = 247.5$ kilobytes (KB) big.

The rate at which images are taken is called the frame rate, and would need to be about 25--30 frames per second to provide video quality as good as television. Frame rates of 15 times per second are still tolerable, while update rates of greater than 30 times a second cannot be perceived by a human, and are therefore not very useful. NV usually runs at a slower speed, on the order of two or three frames a second, to reduce the rate at which data is sent. This is because both the computers involved in the conference and the network in between are typically unable to handle data at such high rates with today's commodity technology. In any case, digital images, once provided to the sending host by the frame grabber, are sent over the network as a stream of messages (frames) to a receiving host. The receiving host then displays the received frames on its monitor at the same rate as they were captured. Figure  shows the control panel for an NV session along with a frame of the video feed from NASA during a space shuttle mission.

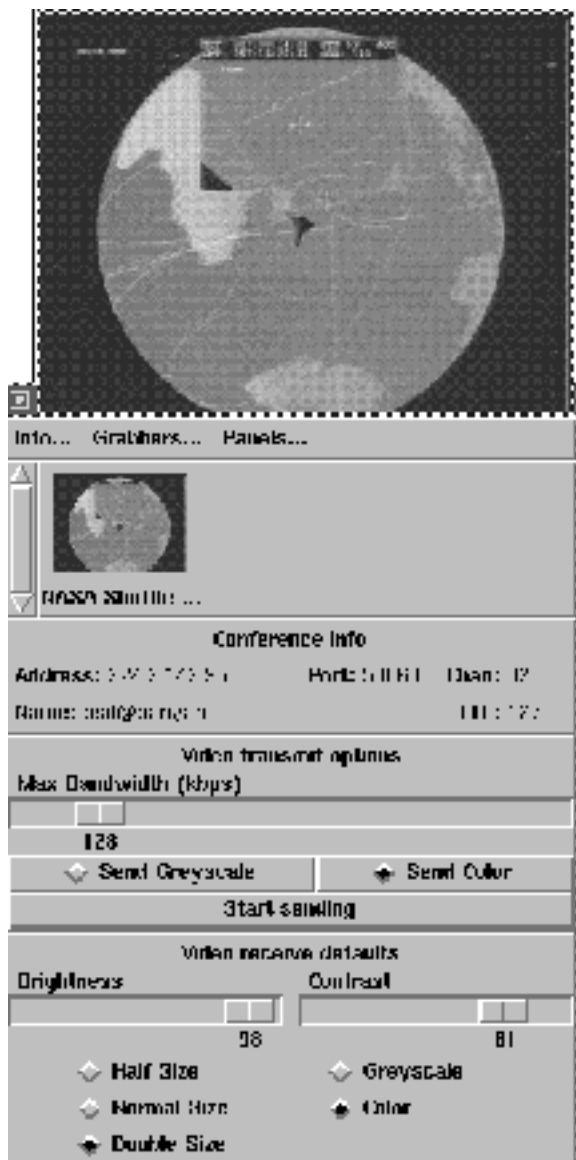


Figure: The NV video application.

[Next](#)
[Up](#)
[Previous](#)

Next: [Requirements](#)
Up: [Foundation](#)
Previous: [Problem: Building a](#)

Requirements

Before trying to understand how a network that supports applications like FTP, WWW, and NV is designed and implemented, it is important first to identify what we expect from a network. The short answer is that there is no single expectation; computer networks are designed and built under a large number of constraints and requirements. In fact, the requirements differ widely depending on your perspective:

- a *network user* would list the services that his or her application needs, for example, a guarantee that each message it sends will be delivered without error within a certain amount of time;
- a *network designer* would list the properties of a cost-effective design, for example, that network resources are efficiently utilized and fairly allocated to different users;
- a *network provider* would list the characteristics of a system that is easy to administer and manage, for example, that faults can be easily isolated and it is easy to account for usage.

This section attempts to distill these different perspectives into a high-level introduction to the major considerations that drive network design, and in doing so, identifies the challenges addressed throughout the rest of this book.

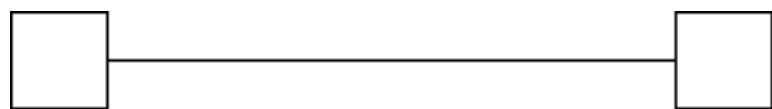
Connectivity

Starting with the obvious, a network must provide connectivity among a set of computers. Sometimes it is enough to build a limited network that connects only a few select machines. In fact, for reasons of privacy and security, many private (corporate) networks have the explicit goal of limiting the set of machines that are connected. In contrast, other networks, and the Internet is the prime example, are designed to grow in a way that allows them potentially to connect all the computers in the world. A system that is designed to support growth to an arbitrarily large size is said to *scale*. Using the Internet

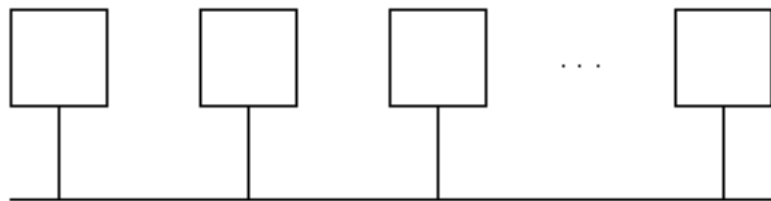
as a model, this book addresses the challenge of scalability.

Links, Nodes and Clouds

Network connectivity occurs at many different levels. At the lowest level, a network can consist of two or more computers directly connected by some physical medium, such as a coaxial cable or an optical fiber. We call such a physical medium a *link*, and we often refer to the computers it connects as *nodes*. (Sometimes a node is a more specialized piece of hardware rather than a computer, but we overlook that distinction for the purposes of this discussion.) As illustrated in Figure [1.1](#), physical links are sometimes limited to a pair of nodes (such a link is said to be *point-to-point*), while in other cases, more than two nodes may share a single physical link (such a link is said to be *multiple access*). Whether a given link supports point-to-point or multiple access connectivity depends on how the node is attached to the link. It is also the case that multiple access links are often limited in size, in terms of both the geographical distance they can cover and the number of nodes they can connect. The exception is a satellite link, which can cover a wide geographic area.



point-to-point network



multiple access network

Figure: Direct Links: Point-to-Point and Multiple Access.

If computer networks were limited to situations where all nodes are directly connected to each other over a common physical medium, then networks would either be very limited in the number of computers they connected, or the number of wires coming out of the back of each node would quickly become both unmanageable and very expensive. Fortunately, connectivity between two nodes does not necessarily imply a direct physical connection between them---indirect connectivity may be achieved among a set of cooperating nodes. Consider the following two examples of how a collection of computers can be indirectly connected.

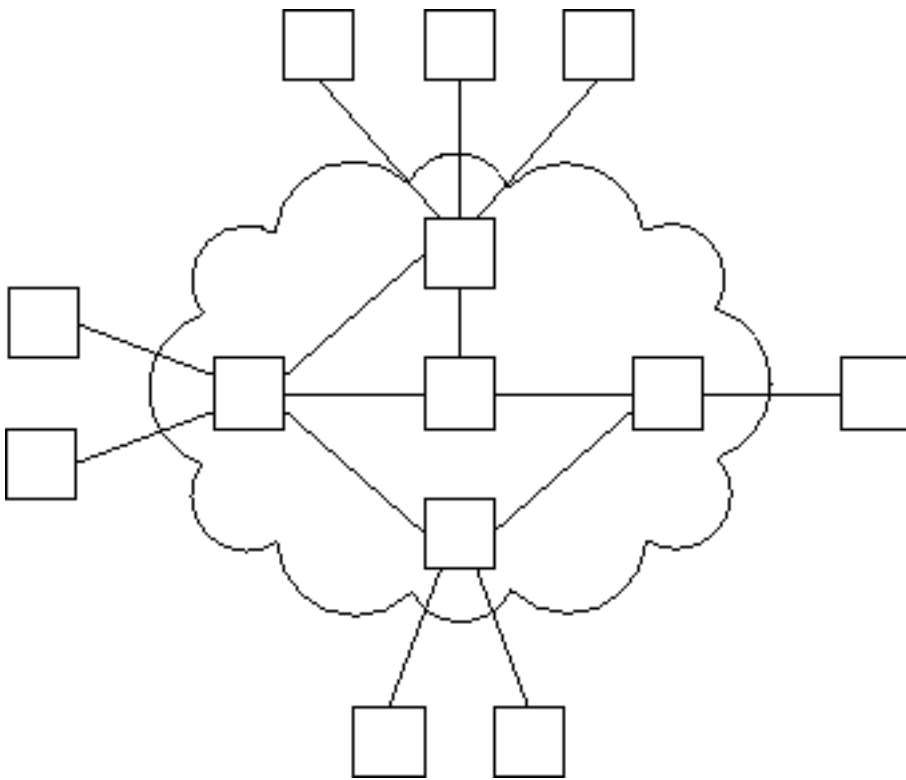






Figure: Switched network.

First, Figure  shows a set of nodes, each of which is attached to one or more point-to-point links. Those nodes that are attached to at least two links run software that forwards data received on one link out on another. If done in a systematic way, these forwarding nodes form a *switched network*. There are numerous types of switched networks, of which the two most common are *circuit-switched* and *packet-switched*. The former is most notably employed by the telephone system, while the latter is used for the overwhelming majority of computer networks, and will be the focus of this book. The important feature of packet-switched networks is that the nodes in such a network send discrete blocks of data to each other. Think of these blocks of data as corresponding to some piece of application data such as file, a piece of email, or an image. We call each block of data either a *packet* or a *message*, and for now, we use these terms interchangeably; we discuss the reason they are not always the same in Section .

Packet-switched networks typically use a strategy called *store-and-forward*. As the name suggests, each node in a store-and-forward network first receives a complete packet over some link, stores the packet in its internal memory, and then forwards the complete packet to the next node. In contrast, a circuit-switched network first establishes a dedicated circuit across a sequence of links, and then allows the source node to send a stream of bits across this circuit to a destination node. The major reason for using packet switching rather than circuit switching in a computer network is discussed in the next subsection.

The cloud in Figure  distinguishes between the nodes on the inside that *implement* the network (they are commonly called *switches* and their sole function is to store and forward packets) and the nodes on the outside of the cloud that *use* the network (they are commonly called *hosts* and they support users and run application programs). Also note that the cloud in Figure  is one of the most important icons of

computer networking. In general, we use a cloud to denote any type of network, whether it is a single point-to-point link, a multiple access link, or a switched network. Thus, whenever you see a cloud used in a figure, you can think of it as a placeholder for any of the networking technologies covered in this book.

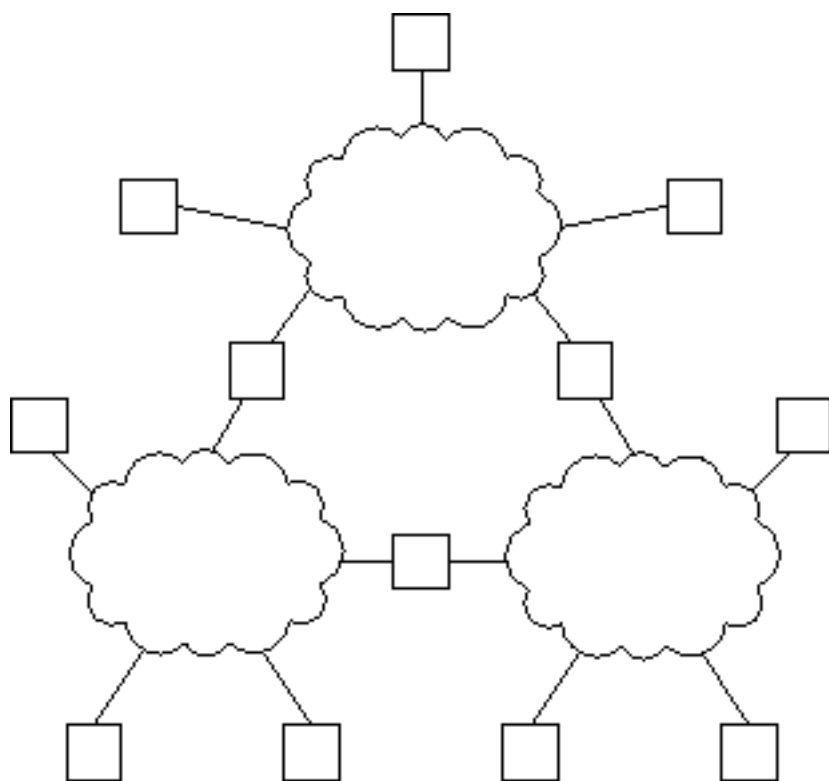



Figure: Interconnection of networks.

A second way in which a set of computers can be indirectly connected is shown in Figure . In this situation, a set of independent networks (clouds) are interconnected to form an *internetwork*, or internet for short. We adopt the Internet's convention of referring to a generic internetwork of networks as a ``little i'' internet, and the currently operational TCP/IP Internet as the ``big I'' Internet. A node that is connected to two or more networks is commonly called a *router* or *gateway*, and it plays much the same role as a switch---it forwards messages from one network to another. Note that an internet can itself be viewed as another kind of network, which means an internet can be built from an interconnection of internets. Thus, we can recursively build arbitrarily large networks by interconnecting clouds to form larger clouds.

Sidebar: DANs, LANs, MANs, and WANs

One way to characterize networks is according to their size. Two well-known examples are LANs (local-area networks) and WAN's (wide-area networks)---the former typically reach less than 1 kilometer, while the latter can be world-wide. Other networks are classified as MANs (metropolitan-area network), which as the name implies, usually span

tens of kilometers. The reason such classifications are interesting is that the size of a network often has implications on the underlying technology that can be used, with a key factor being the amount of time it takes for data to propagate from one end of the network to the other; more on this issue in later chapters.

An interesting historical note is that the term wide-area network was not applied to the first WANs because there was no other sort of network to differentiate them from. When computers were incredibly rare and expensive, there was no point thinking about how to connect all the computers in the 'local area'---there was only one computer in that area. Only as computers began to proliferate did LANs become necessary, and the term WAN was then introduced to describe the larger networks that interconnected geographically distant computers.

One of the most intriguing kinds of networks that are gaining attention today are DANs (desk-area networks). The idea of a DAN is to "open up" the computer setting on your desk, and to treat each component of that computer---e.g., its display, disk, CPU, as well as peripherals like cameras and printers---as a network-accessible device. In essence, the I/O bus is replaced by a network (a DAN) that can, in turn, be interconnected to other LANs, MAN, and WANs. Doing this provides uniform access to all the resources that might be required by a network application.

Just because a set of hosts are directly or indirectly connected to each other does not mean that we have succeeded in providing host-to-host connectivity. The final requirement is that each node must be able to say which of the other nodes on the network it wants to communicate with. This is done by assigning an *address* to each node. An address is a byte-string that identifies a node, that is, the network can use a node's address to distinguish it from the other nodes connected to the network. When a source node wants the network to deliver a message to a certain destination node, it specifies the address of the destination node. If the sending and receiving nodes are not directly connected, then the switches and routers of the network also use this address to decide how to forward the message towards the destination. The process of determining systematically how to forward messages towards the destination node based on its address is called *routing*.

This brief introduction to addressing and routing has presumed that the source node wants to send a message to a single destination node. While this is the most common scenario, it is also possible that the source node might want to *broadcast* a message to all the nodes on the network. As a slight refinement to broadcast, a source node might want to send a message to some subset of the other nodes, but not all of them. This situation is called *multicast*. Thus, in addition to node-specific (*unicast*) addresses, another requirement of a network is that it support multicast and broadcast addresses.

The main thing to take away from this discussion is that we can define a network recursively as

consisting of two or more nodes connected by a physical link, or by two or more networks connected by one or more nodes. In other words, a network can be constructed from a nesting of networks, where at the bottom-most level, a network is implemented by some physical medium. One of the key challenges in providing network connectivity is to define an address for each node that is reachable on the network (including support for broadcast and multicast), and to use this address to route messages towards the appropriate destination node(s).

Cost-Effective Resource Sharing

As stated above, this book focuses on packet-switched networks. This section explains the key requirement of computer networks---in short, efficiency---that leads us to packet switching as the strategy of choice.

Given a collection of nodes indirectly connected by a nesting of networks, it is possible for any pair of hosts to send messages to each other across a sequence of links and nodes. Of course, we want to do more than support just one pair of communicating hosts---we want to provide all pairs of hosts with the ability to exchange messages. The question, then, is how do all the hosts that want to communicate share the network, especially if they want to use it at the same time? As if that problem isn't hard enough, how do several hosts share the same *link* when they all want to use it at the same time?

To understand how hosts share a network, we need to introduce a fundamental concept--- *multiplexing*---which is a way of saying that a system resource is shared among multiple users. At an intuitive level, multiplexing can be explained by analogy to a timesharing computer system, where a single physical CPU is shared (multiplexed) among multiple jobs, each of which believes it has its own private processor. Similarly, data being sent by multiple users can be multiplexed over the physical links that make up a network.

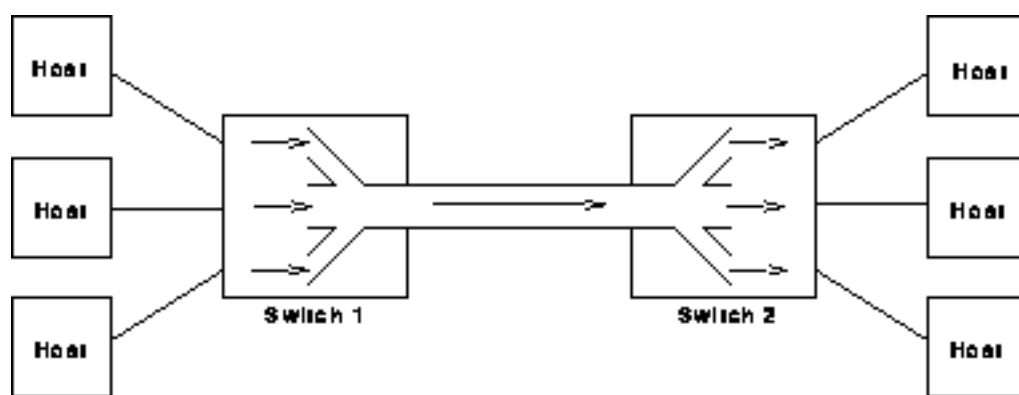



Figure: Multiplexing multiple logical flows over a single physical link.

To see how this might work, consider the simple network illustrated in Figure , where the three hosts on the left side of the network are sending data to the three hosts on the right side of the network by

sharing a switched network that contains only one physical link. (For simplicity, assume the top host on the left is communicating with the top host on the right, and so on.) In this situation, three flows of data---corresponding to the three pairs of hosts---are multiplexed onto a single physical link by Switch 1, and then *demultiplexed* back into separate flows by Switch 2. Note that we are being purposely vague about exactly what a ``flow of data" corresponds to. For the purposes of this discussion, assume each host on the left has a large supply of data that it wants to send to its counterpart on the right.

There are several different methods for multiplexing multiple flows onto one physical link. One method, which is commonly used in the telephone network, is *synchronous time-division multiplexing* (STDM). The idea of STDM is to divide time into equal-sized quanta, and in a round-robin fashion, give each flow a chance to send its data over the physical link. In other words, during time quantum 1, data from the first flow is transmitted; during time quantum 2, data from the second flow is transmitted; and so on. This process continues until all the flows have had a turn, at which time, the first flow gets to go again, and the process repeats. Another common method is *frequency-division multiplexing* (FDM). The idea of FDM is to transmit each flow over the physical link at a different frequency, much in the same way that the signals for different TV stations are transmitted at a different frequency on a physical cable TV link.

Although simple to understand, both STDM and FDM are limited in two ways. First, if one of the flows (host pairs) does not have any data to send, then its share of the physical link---i.e., its time quantum or its frequency---remains idle, even if one of the other flows has data to transmit. For computer communication, the amount of time that a link is idle can be very large---for example, consider the amount of time you spend reading a Web page (leaving the link idle) compared to how much time you spend fetching the page. Second, both STDM and FDM are limited to situations where the maximum number of flows is fixed, and known ahead of time. It is not practical to resize the quantum or add additional quanta in the case of STDM, or to add new frequencies in the case of FDM.

The form of multiplexing that we make most use of in this book is called *statistical multiplexing*. Although the name is not all that helpful in understanding it, statistical multiplexing is really quite simple; it involves two key ideas. First, it is like STDM in that the physical link is shared in time---first data from one flow is transmitted over the physical link, then data from another flow is transmitted, and so on. Unlike STDM, however, data is transmitted from each flow on demand rather than during a pre-determined time slot. Thus, if only one flow has data to send, it gets to transmit that data without waiting for its quantum to come around, and thus, without having to watch the quanta assigned to the other flows go by unused. It is this avoidance of idle time that gives packet switching its efficiency.

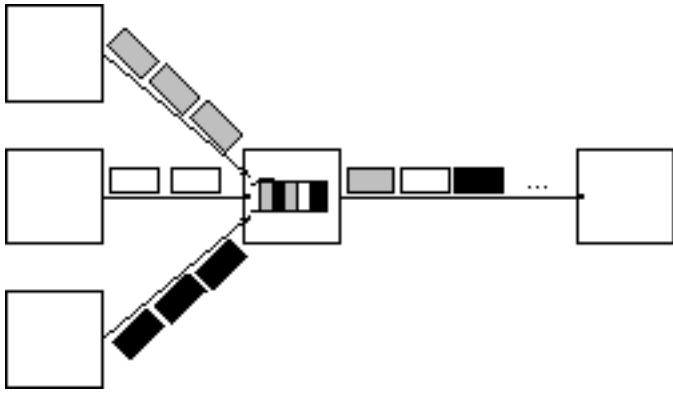





Figure: A switch multiplexing packets from multiple sources onto one shared link.

As defined so far, however, statistical multiplexing has no mechanism to ensure that all the flows eventually get their turn to transmit over the physical link. That is, once a flow begins sending data, we need some way to limit the transmission, so that the other flows can have a turn. To account for this need, the second aspect of statistical multiplexing is to define an upper bound on the size of the block of data that each flow is permitted to transmit at a given time. This limited-sized block of data is typically referred to as a *packet*, so as to distinguish it from the arbitrarily large *message* that an application program might want to transmit. The fact that a packet-switched network limits the maximum size of packets means that a host may not be able to send a message in one packet---the source may need to *fragment* the message into several packets, with the receiver *reassembling* the packets back into the original message.

In other words, each flow sends a sequence of packets over the physical link, with a decision made on a packet-by-packet basis as to which flow's packet to send next. Notice that if only one flow has data to send, then it can send a sequence of packets back-to-back. However, should more than one of the flows have data to send, then their packets are interleaved on the link. Figure  depicts a switch multiplexing packets from multiple sources onto a single shared link.

The decision as to which packet to send next on a shared link can be made in a number of different ways. For example, in a network consisting of switches interconnected by links like the one in Figure , the decision would be made by the switch that transmits packets onto the shared link. (As we will see later, not all packet-switched networks actually involve switches, and they use other mechanisms to determine whose packet goes onto the link next.) Each switch in a packet-switched network makes this decision independently, on a packet-by-packet basis. One of the issues that faces a network designer is how to make this decision in a fair manner. For example, a switch could choose to service the different flows in a round-robin manner, just as in STDM. However, statistical multiplexing does not require this. In fact, another equally valid choice would be to service each flow's packets on a first-in-first-out (FIFO) basis.

Also, notice in Figure  that since the switch has to multiplex three incoming packet streams onto one outgoing link, it is possible that the it will receive packets faster than the shared link will accommodate. In this case, the switch is forced to buffer these packets in its memory. Should a switch receive packets

faster than it can send them for an extended period of time, then the switch will eventually run out of buffer space, and some packets will have to be dropped. When a switch is operating in this state, it is said to be *congested*.

The bottom line is that statistical multiplexing defines a cost-effective way for multiple users (e.g., host-to-host flows of data) to share network resources (links and nodes) in a fine-grain manner. It defines the packet as the granularity with which the links of the network are allocated to different flows, with each switch able to schedule the use of the physical links it is connected to on a per-packet basis. Fairly allocating link capacity to different flows and dealing with congestion when it occurs are the key challenges of statistical multiplexing.

Functionality

While the previous section outlined the challenges involved in providing cost-effective connectivity among a collection of hosts, it is overly simplistic to view a computer network as simply delivering packets among a collection of computers. It is more accurate to think of a network as providing the means for a set of application processes distributed over those computers to communicate. In other words, the next requirement is that the application programs running on the hosts connected to the network must be able to communicate in a meaningful way.

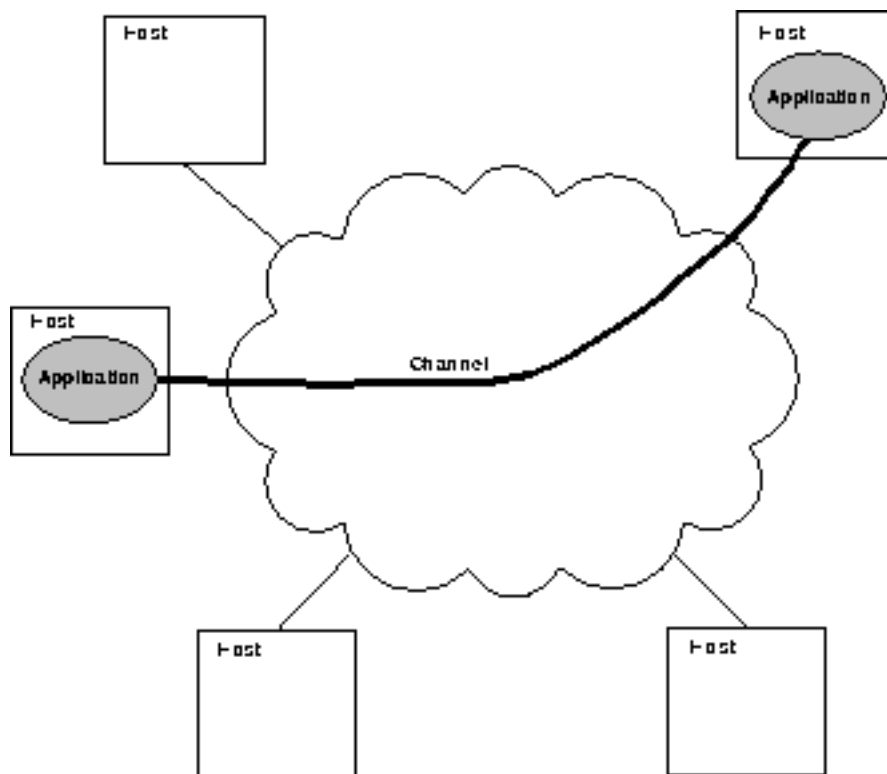



Figure: Processes communicating over an abstract channel.

When two application programs need to communicate with each other, there are a lot of complicated


things that need to happen beyond simply sending a message from one host to another. One option would be for application designers to build all that complicated functionality into each application program. However, since many applications need common services, it is much more logical to implement those common services once, and then to let the application designer build the application on top of those services. The challenge for a network designer is to identify the right set of common services. The goal is to hide the complexity of the network from the application without overly constraining the application designer.

Intuitively, we view the network as providing logical *channels* over which application-level processes can communicate with each other; each channel provides the set of services required by that application. In other words, just as we use a cloud to abstractly represent connectivity among a set of computers, we now think of a channel as connecting one process to another. Figure  shows a pair of application-level processes communicating over a logical channel that is, in turn, implemented on top of a cloud that connects a set of hosts. We can think of the channel as being like a pipe connecting two applications, so that a sending application can put data in one end and expect that data to be delivered by the network to the application at the other end of the pipe.

The challenge is to recognize what functionality channels should provide to application programs. For example, does the application require a guarantee that messages sent over the channel are delivered, or is it acceptable if some messages fail to arrive? As another example, is it necessary that messages arrive at the recipient process in the same order in which they are sent, or does the recipient not care about the order in which messages arrive? As a third example, does the network need to ensure that no third parties are able to eavesdrop on the channel, or is privacy not a concern? In general, a network provides a variety of different channel types, with each application selecting the type that best meets its needs. The rest of this section illustrates the thinking involved in defining useful channels.

Identifying Common Communication Patterns

Designing abstract channels involves first understanding the communication needs of a representative collection of applications, then extracting their common communication requirements, and finally incorporating the functionality that meets these requirements in the network.

To illustrate how one might define an abstract channel, consider the example applications introduced in Section . One of the earliest applications supported on any network is a file access program like FTP. Although many details vary---for example, whether whole files are transferred across the network, or only single blocks of the file are read/written at a given time---the communication component of remote file access is characterized by a pair of processes, one that requests a file be read or written, and a second process that honors this request. Recall that the process that requests access to the file is called the client, and the process that supports access to the file is called the server.

Reading a file involves the client sending a small request message to a server, and the server responding

with a large message that contains the data in the file. Writing works in the opposite way---the client sends a large message containing the data to be written to the server, and the server responds with a small message confirming that the write to disk has taken place.

A more recent application class that is gaining in popularity, but which in many respects behaves like remote file access, is digital library programs. As exemplified by the World Wide Web, the kind of data being retrieved from a digital library may vary---e.g., it may be a small text file, a very large digital image, or some sort of multimedia object---but the communication pattern looks very similar to the file transfer described above: a client process makes a request and a server process responds by returning the requested data.

A third class of applications that differs substantially from these first two involves the playing of video over the network. While an entire video file could be first fetched from a remote machine using a file access application, and then played at the local machine, this would entail waiting for the last second of the video file to be delivered before starting to look at the start of it. Consider instead the case where the sender and receiver are, respectively, the source and the sink for the video stream. That is, the source generates a sequence of video frames, each of which is sent across the network as a message and then displayed at the destination as it is received.

Video, in and of itself, is not an application. It is a type of data. Video data might be used as part of a video-on-demand application, or a teleconferencing application like NV. Although similar, these two applications have a major difference. In the case of video-on-demand, there are no serious timing constraints; if it takes 10 seconds from the time the user starts the video until the first frame is displayed, then the service is still deemed satisfactory. In contrast, a teleconferencing system has very tight timing constraints. Just as when using the telephone, the interactions among the participants must be timely. When a person at one end gestures, then the corresponding video frame must be displayed at the other end as quickly as possible. Added delay makes the system unusable. It is also the case that interactive video implies that frames are flowing in both directions, while a video-on-demand application is mostly likely sending frames in only one direction.

Using these four applications as a representative sample, one might decide to provide the following two types of channels: *request/reply* channels and *stream* channels. The request/reply channel would be used by the file transfer and digital library applications. It would guarantee that every message sent by one side is received by the other side, and that only one copy of each message is delivered. The request/reply channel might also protect the privacy and integrity of the data that flows over it, so that unauthorized parties cannot read or modify the data being exchanged between the client and server processes.

The stream channel could be used by both the video-on-demand and teleconferencing applications, provided it is parameterized to support both one-way and two-way traffic, and to support different delay properties. The stream-based channel might not need to guarantee that all messages are delivered, since a video application can operate adequately even if some frames are not received. It would, however, need to ensure that those messages that are delivered arrive in the same order in which they were sent.

This is because a video application does not want to display frames out of sequence. Like the request/reply channel, the message stream channel might want to ensure the privacy and integrity of the video data. Finally, the stream channel might support multicast, so that multiple parties can participate in the teleconference or view the video.


While it is common for a network designer to strive for the smallest number of abstract channel types that can serve the largest number of applications, there is a danger in trying to get away with too few channel abstractions. Simply stated, if you have a hammer, then everything looks like a nail. For example, if all you have are stream and request/reply channels, then it is tempting to use them for the next application that comes along, even if neither provides exactly the semantics needed by the application. Thus, network designers will probably be inventing new types of channels---and adding options to existing channels---for as long as application programmers are inventing new applications.

Reliability

As suggested by the examples just considered, reliable message delivery is one of the most important functions that a network can provide. It is difficult to determine how to provide this reliability, however, without first understanding how networks can fail. The first thing to recognize is that computer networks do not exist in a perfect world. Machines crash and later reboot, fibers are cut, electrical interference corrupts bits in the data being transmitted, and if these sorts of physical problems aren't enough to worry about, the software that manages the hardware sometimes forwards packets into oblivion. Thus, a major requirement of a network is to mask (hide) certain kinds of failures, so as to make the network appear more reliable than it really is to the application programs using it.

There are three general classes of failures that network designers have to worry about. First, as a packet is transmitted over a physical link, *bit errors* may be introduced into the data; i.e., a 1 is turned into a 0, or vice versa. Sometimes single bits are corrupted, but more often than not, a *burst error* occurs---several consecutive bits are corrupted. Bit errors typically occur because outside forces, such as lightning strikes and power surges, interfere with the transmission of data. The good news is that such bit errors are fairly rare, affecting on average only one of every 10^6 to 10^7 bits on a typical copper-based cable, and one of every 10^{12} to 10^{14} bits on a typical optical fiber. As we will see, there are techniques that detect these bit errors with high probability. Once detected, it is sometimes possible to correct for such errors---if we know which bit or bits are corrupted, we can simply flip them---while other times the damage is so bad that it is necessary to discard the entire packet. In this case, the sender may be expected to *retransmit* the packet.

The second class of failures is at the packet, rather than the bit, level. That is, a complete packet is lost by the network. One reason this can happen is that the packet contains an uncorrectable bit error, and therefore has to be discarded. A more likely reason, however, is that the software running on one of the nodes that handles the packet---e.g., a switch that it is forwarding it from one link to another---makes a mistake. For example, it might incorrectly forward a packet out on the wrong link, so that the packet never finds its way to the ultimate destination. Even more commonly, the forwarding node is so

overloaded that it has no place to store the packet, and therefore, is forced to drop it. This is the problem of congestion mentioned in Section . As we will see, one of the main difficulties in dealing with lost packets is distinguishing between a packet that is indeed lost, and one that is merely late in arriving at the destination.

The third class of failures is at the node and link level, that is, a physical link is cut or the computer it is connected to crashes. This can be caused by software that crashes, a power failure, or a reckless backhoe operator. While such failures can eventually be corrected, they can have a dramatic effect on the network for an extended period of time. However, they need not totally disable the network. In a packet-switched network, for example, it is sometimes possible to "route around" a failed node or link. One of the difficulties in dealing with this third class of failure is distinguishing between a failed computer and one that is merely slow, or in the case of a link, between one that has been cut and one that is very flaky and therefore introducing a high number of bit errors.

The thing to take away from this discussion is that defining useful channels involves both understanding the applications' requirements and recognizing the limitations of the underlying technology. The challenge is to fill in the gap between what the application expects and what the underlying technology can provide.


Performance

Like any computer system, computer networks are expected to exhibit high performance, and often more importantly, high performance per unit cost. Computations distributed over multiple machines use networks to exchange data. The effectiveness of these computations often depends directly on the efficiency with which the network delivers that data. While the old programming adage "first get it right and then make it fast" is valid in many settings, in networking, it is usually necessary to "design for performance". It is therefore important to understand the various factors that impact network performance.

Bandwidth and Latency

Network performance is measured in two fundamental ways: *bandwidth* (also called *throughput*) and *latency* (also called *delay*). The bandwidth of a network is given by the number of bits that can be transmitted over the network in a certain period of time. For example, a network might have a bandwidth of 10 million bits/second (Mbps), meaning that it is able to deliver 10 million bits every second. It is sometimes useful to think of bandwidth in terms of how long it takes to transmit each bit of data. On a 10Mbps network, for example, it takes 0.1 microsecond (us) to transmit each bit.

While you can talk about the bandwidth of the network as a whole, sometimes you want to be more

precise, focusing for example on the bandwidth of a single physical link or a logical process-to-process channel. At the physical level, bandwidth is constantly improving, and no end is in sight. Intuitively, if you think of a second of time as a distance you could measure on a ruler, and bandwidth as how many bits fit in that distance, then you can think of each bit as a pulse of some width. For example, each bit on a 1Mbps link is 1us wide, while each bit in a 2Mbps link is 0.5us wide, as illustrated in Figure . The more sophisticated the transmitting and receiving technology, the narrower each bit can become, and thus, the higher the bandwidth. For logical process-to-process channels, bandwidth is also influenced by other factors, including how many times the software that implements the channel has to handle, and possibly transform, each bit of data.

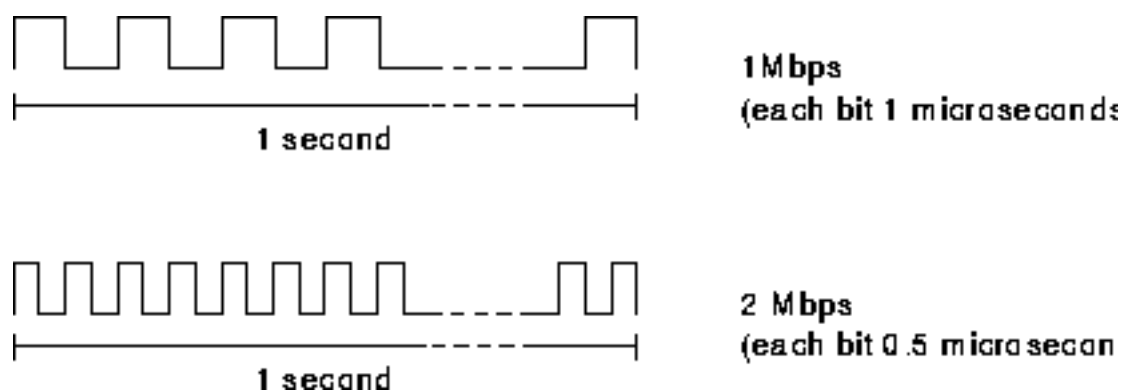


Figure: Bits transmitted at a particular bandwidth can be viewed as having some width.

Sidebar: Bandwidth and Throughput

Bandwidth and throughput are two of the most confusing terms used in networking. While we could try to give you a precise definition of each, it is important that you know how other people might use them, and to be aware that they are often used interchangeably.

First of all, bandwidth can be applied to an analog channel, such as a voice grade telephone line. Such a line supports signals in the range of 300Hz and 3300Hz and is said to have a bandwidth of $3300\text{Hz} - 300\text{Hz} = 3000\text{Hz}$. If you see the word bandwidth used in a situation where it is measured in Hertz (Hz), then it probably refers to the range of analog signals that can be accommodated.

When we talk about the bandwidth of a digital link, we normally refer to the number of bits per second that can be transmitted on the link. So, we might say that the bandwidth of an Ethernet is 10Mbps. A useful distinction might be drawn, however, between the bandwidth that is available on the link and the number of bits per second that we can actually transmit over the link in practice. We tend to use the word *throughput* to refer to the *measured performance* of a system. Thus, because of various inefficiencies of implementation, a pair of nodes connected by a link with a bandwidth of 10Mbps might

achieve a throughput of only 2Mbps. This would mean that an application on one host could send data to the other host at 2 Mbps.

Finally, we often talk about the bandwidth *requirements* of an application. This is the number of bits per second that it needs to transmit over the network to perform acceptably. For some applications, this might be 'whatever I can get', for others it might be some fixed number (preferably no more than the available link bandwidth), and for others it might be a number that varies with time. More on this topic later in this section.

The second performance metric---latency---corresponds to how long it takes a single bit to propagate from one end of a network to the other. (As with bandwidth, we could be focused on the latency of a single link or channel.) Latency is measured strictly in terms of time. For example, a transcontinental network might have a latency of 24 milliseconds (ms), that is, it takes a single bit 24ms to travel from one end of the country to the other. There are many situations in which it is more important to know how long it takes to send a bit from one end of a network to the other and back, rather than the one-way latency. We call this the *round-trip time* (RTT) of the network.

We often think of latency as having three components. First, there is the speed-of-light propagation delay. This arises because nothing, including a bit on a wire, can travel faster than the speed of light. If you know the distance between two points, you can calculate the speed-of-light latency, although you have to be careful because light travels across different mediums at different speeds: it travels at 3.0×10^8 meters/second in a vacuum, 2.3×10^8 meters/second in a cable, and 2.0×10^8 meters/second in a fiber. Second, there is the amount of time it takes to transmit a unit of data. This is a function of the network bandwidth and the size of the packet in which the data is carried. Third, there may be queuing delays inside the network, since packet switches generally need to store packets for some time before forwarding them on an outbound link, as discussed in Section [4.1](#). So, we could define the total latency as

```
Latency = Propagation + Transmit + Queue
Propagation = Distance / SpeedOfLight
Transmit = Size / Bandwidth
```

where `Distance` is the length of the wire over which the data will travel, `SpeedOfLight` is the effective speed of light over that wire, `Size` is the size of the packet, and `Bandwidth` is the bandwidth at which the packet is transmitted.

While there is eternal optimism that network bandwidth will continue to improve, observe that latency is fundamentally limited by the speed of light. We can make `Transmit + Queue` small by using high bandwidth links, but, in the words of Scotty from Star Trek, "You cannae change the laws of physics."

Thus, the 24ms latency mentioned above corresponds to the time it takes light to travel through a fiber across the approximately 3,000-mile width of the U.S.

Bandwidth and latency combine to define the performance characteristics of a given link or channel. Their relative importance, however, depends on the application. For some applications, latency dominates bandwidth. For example, a client that sends a one-byte message to a server and receives a one-byte message in return is latency bound. Assuming no serious computation is involved in preparing the response, the application will perform much differently on a cross-country channel with a 100ms RTT than it will on an across-the-room channel with a 1ms RTT. Whether the channel is 1Mbps or 100Mbps is relatively insignificant, however, since the former implies the time to transmit a byte (T_{transmit}) is 8us and the latter implies $T_{\text{transmit}} = 0.08\text{us}$.

In contrast, consider a digital library program that is being asked to fetch a 25 megabyte (MB) image---the more bandwidth that is available, the faster it will be able to return the image to the user. Here, the bandwidth of the channel dominates performance. To see this, suppose the channel has a bandwidth of 10Mbps. It will take 20 seconds to transmit the image, making it relatively unimportant if the image is on the other side of a 1ms channel or a 100ms channel; the difference between a 20.001 second response time and a 20.1 second response time is negligible.

Perceived Latency vs. RTT

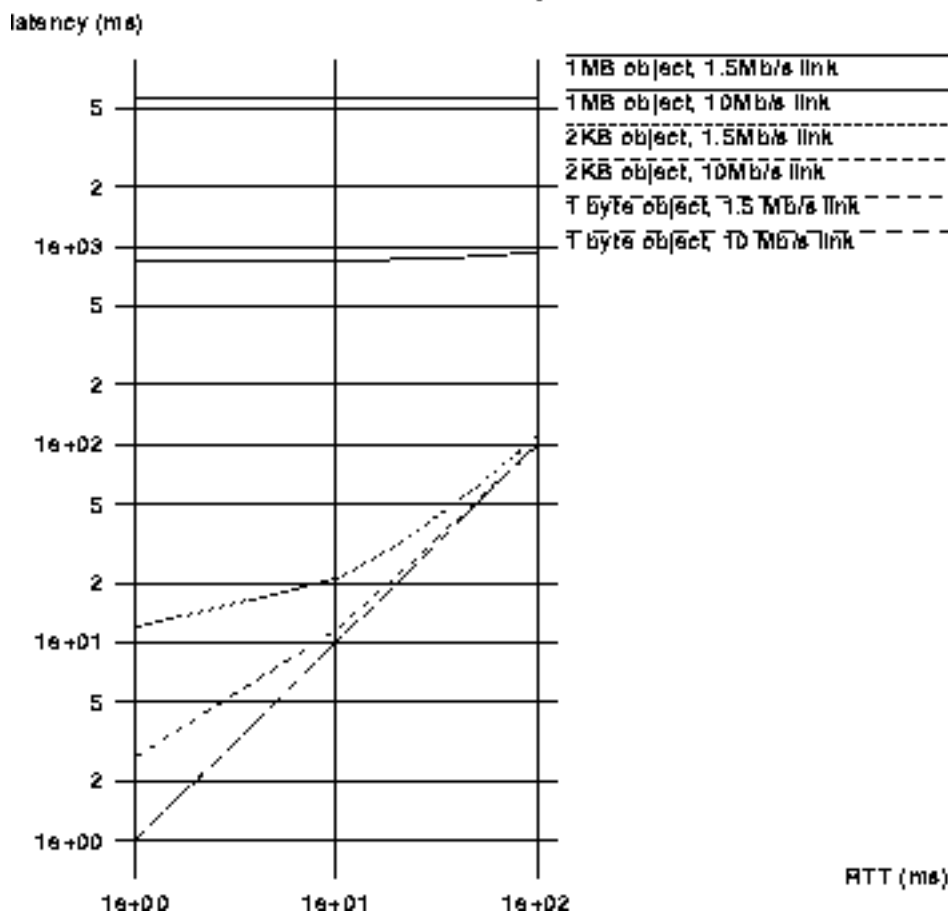



Figure: Perceived latency (response time) vs. round-trip time for various object

Figure  gives you a sense of how latency or bandwidth can dominate performance in different circumstances. The graph shows how long it takes to move objects of various sizes (1 byte, 2KB, 1 MB) across networks with RTTs ranging from 1 to 100 ms and link speeds of 1.5 or 10 Mbps. We use logarithmic scales to show relative performances. For a one byte object (say a keystroke), latency remains almost exactly equal to the RTT, so that you cannot distinguish between a 1.5 Mbps network and a 10 Mbps network. For a 2KB object (say an email message), the link speed makes quite a difference on a 1ms RTT network, but a negligible difference on a 100ms RTT network. And for a 1MB object (say a digital image) the RTT makes no difference---it is the link speed that dominates performance across the full range of RTT.

Note that throughout this book we use the terms "latency" and "delay" in a generic way, that is, to denote how long it takes to perform a particular function like deliver a message or move an object. When we are referring to the specific amount of time it takes a signal to propagate from one end of a link to another, we use the term "propagation delay". Also, we make it clear in the context whether we are referring to the one-way latency or round-trip time.

Sidebar: How Big is a Mega?

There are several pitfalls one needs to be aware of when working with the common "units" of networking---MB, Mbps, KB, and Kbps. The first is to distinguish between bits and bytes. Throughout this book, we always use a little "b" for bits and a capital "B" for bytes. The second is to be sure you are using the appropriate definition of mega (M) and kilo (K). Mega, for example, can mean either 2^{20} or 10^6 . Similarly kilo can be either 2^{10} or 10^3 . What is worse, in networking we typically use both definitions. Here's why.

Network bandwidth, which is often specified in terms of Mbps, is typically governed by the speed of the clock that paces the transmission of the bits. A clock that is running at 10 Megahertz (MHz) is used to transmit bits at 10Mbps. Because the mega in MHz means 10^6 hertz, Mbps is usually also defined in terms of 10^6 bits-per-second. (Similarly with Kbps.) On the other hand, when we talk about a message that we want to transmit, we often give its size measured in kilobytes. Because messages are stored in the computer's memory, and memory is typically measured in powers of two, the K in KB is usually taken to mean 2^{10} . (Similarly for MB.) When you put the two together, it is not uncommon to talk about sending a 32KB message over a 10Mbps channel, which should be interpreted to mean $32 \times 2^{10} \times 8$ bits are being transmitted at a rate of 10×10^6 bits-per-second. This is the interpretation used throughout the book, unless explicitly stated otherwise.

The good news is that many times we are satisfied with a back-of-the-envelope

calculation, in which case, it is perfectly reasonable to pretend that a byte has 10 bits in it (making it easy to convert between bits and bytes) and that 10^6 is really equal to 2^{20} (making it easy to convert between the two definitions of mega). Notice that the first approximation introduces a 20% error, while the latter introduces only a 5% error.

To help you in your quick-and-dirty calculations, 100ms is a reasonable number to use for a cross-country round-trip time (RTT) and 1ms is a good approximation of an RTT across a local area network. In the case of the former, we increase the 48ms round-trip time implied by the speed of light over a fiber to 100ms because there are, as we have said, other sources of delay, such as the processing time in the switches inside the network. You can also be sure that the path taken by the fiber between two points will not be a straight line.

As an aside, computers are becoming so fast that when we connect them to networks, it is sometimes useful to think, at least figuratively, in terms of *instructions-per-mile*. Consider what happens when a computer that is able to execute 200 million instructions per second sends a message out on a channel with a 100ms RTT. (To make the math easier, assume the message covers a distance of 5,000 miles.) If that computer sits idle the full 100ms waiting for a reply message, then it has forfeited the ability to execute 20 million instructions, or 4,000 instructions-per-mile. It had better have been worth going over the network to justify this waste.

Delay Bandwidth Product


It is also useful to talk about the product of these two metrics, often called the *delay × bandwidth* product. Intuitively, if we think of a channel between a pair of processes as a hollow pipe (see Figure ) , where the latency corresponds to the length of the pipe and the bandwidth gives the diameter of the pipe, then the delay × bandwidth product gives the volume of the pipe---the number of bits it holds. Said another way, if latency (measured in time) corresponds to the length of the pipe, then given the width of each bit (also measured in time), you can calculate how many bits fit in the pipe. For example, a transcontinental channel with a one-way latency of 50ms and a bandwidth of 45Mbps is able to hold $50 \times 10^{-3} \text{ sec} \times 45 \times 10^6 \text{ bits/sec} = 2.25 \times 10^6 \text{ bits}$, or approximately 280KB of data. In other words, this example channel (pipe) holds as many bytes as the memory of a personal computer of the early 1980s.



Figure: Network as a pipe.

The delay \times bandwidth product is important when constructing high performance networks because it corresponds to how many bits the sender must transmit before the first bit arrives at the receiver. If the sender is expecting the receiver to somehow signal that bits are starting to arrive, and it takes another channel latency for this signal to propagate back to the sender---i.e., we are interested in the channel's RTT rather than just its one-way latency---then the sender can send up to two delay \times bandwidth's worth of data before hearing from the receiver that all is well. Also, if the receiver says ``stop sending'', it might receive that much data before the sender manages to respond. In our above example, that corresponds to 5.5×10^6 bits (560KB) of data.

Note that most of the time we are interested in RTT scenario, which we simply refer to as the ``delay \times bandwidth product'', without explicitly saying that this product is multiplied by two. Again, whether the ``delay'' in ``delay \times bandwidth'' means one-way latency or RTT is made clear in the context.


Application Performance Needs

The discussion in this section has taken a network-centric view of performance, that is, we have talked in terms of what a given link or channel will support. The unstated assumption is that application programs have simple needs---they want as much bandwidth as the network can provide. This is certainly true of the aforementioned digital library program that is retrieving a 25 MB image---the more bandwidth available, the faster it will be able to return the image to the user. However, some applications are able to state an upper limit on how much bandwidth they need. For example, a video application that needs to transmit a 128KB video frame 30 times a second might request a throughput rate of 32Mbps. The ability of the network to provide more bandwidth is of no interest to such an application because it has only so much data to transmit in a given period of time.

Unfortunately, the situation is not as simple as this example suggests. Because the difference between any two adjacent frames in a video stream is often small, it is possible to compress the video by transmitting only the differences between adjacent frames. This compressed video does not flow at a constant rate, but varies with time according to factors such the amount of action and detail in the picture, and the compression algorithm being used. Therefore, it is possible to say what the *average* bandwidth requirement will be, but the instantaneous rate may be more or less.


The key issue is the time interval over which the average is computed. Suppose this example video application can be compressed down to the point that it needs only 2Mbps, on average. If it transmits 1 megabit in a 1 second interval and 3 megabits in the following 1 second interval, then over the 2 second interval it is transmitting at an average rate of 2Mbps, but this will be of little consolation to a channel that was engineered to support no more than 2 megabits in any one second. Clearly, just knowing the average bandwidth needs of an application will not always suffice.

Generally, however, it is possible to put an upper bound on how big a *burst* applications like this are likely to transmit. A burst might be described by some *peak* rate that is maintained for some period of

time. Alternatively, it could be described as a number of bytes that can be sent at the peak rate before reverting to the average rate or some lower rate. If this peak rate is higher than the available channel capacity, then the excess data will have to be buffered somewhere, to be transmitted later. Knowing how big a burst might be sent allows the network designer to allocate sufficient buffer capacity to hold the burst. We will return to the subject of describing bursty traffic accurately in Chapter .

Analogous to the way an application's bandwidth needs can be something other than "all it can get", an application's delay requirements may be more complex than simply "as little delay as possible". In the case of delay, it sometimes doesn't matter so much whether the one-way latency of the network is 100ms or 500ms, what matters is how much the latency varies from packet to packet. The variation in latency is called *jitter*.

Consider the situation where the source sends a packet once every 33ms, as would be the case for a video application transmitting frames 30 times a second. If the packets arrive at the destination spaced out exactly 33ms apart, then we can deduce that the delay experienced by each packet in the network was exactly the same. If the spacing between when packets arrive at the destination---this is sometimes called the *inter-packet gap*---is variable, however, then the delay experienced by the sequence of packets must have also been variable, and the network is said to have introduced jitter into the packet stream. Such variation is not generally introduced in a single physical link, but it can happen when packets experience different queuing delays in a multi-hop packet-switched network. This queuing delay corresponds to the Queue component of latency defined earlier in this section, which varies with time.

To understand the relevance of jitter, suppose the packets being transmitted over the network contain video frames, and in order to display these frames on the screen, the receiver needs to receive a new one every 33ms. If a frame arrives early, then it can simply be saved by the receiver until it is time to display it. Unfortunately, if a frame arrives late, then the receiver will not have the frame it needs in time to update the screen, and the video quality will suffer; it will not be smooth. Note that it is not necessary to eliminate jitter, only to know how bad it is. This is because if the receiver knows the upper and lower bounds on the latency that a packet can experience, then it can delay the time at which it starts playing back the video (i.e., displays the first frame) long enough to ensure that in the future it will always have a frame to display when it needs it. We return to the topic of jitter in Chapter .

Next: [Network Architecture](#) **Up:** [Foundation](#) **Previous:** [Motivation](#)

Network Architecture

In case you didn't notice, the previous section establishes a pretty substantial set of requirements on network design---a computer network must provide general, cost-effective, fair, robust, and high-performance connectivity among a large number of computers. As if this weren't enough, networks do not remain fixed at any single point in time, but must evolve to accommodate changes in both the underlying technologies upon which they are based, as well as changes in the demands placed on them by application programs. Designing a network to meet these requirements is no small task.


To help deal with this complexity, network designers have developed general blueprints---they are generally called a *network architecture*---that guide the design and implementation of networks. This section defines more carefully what we mean by a network architecture by introducing the central ideas that are common to all network architectures. It also introduces two of the most widely referenced architectures---the OSI architecture and the Internet architecture.

Layering and Protocols

When the system gets complex, the system designer introduces another level of abstraction. The idea of an abstraction is to define a unifying model that captures some important aspect of the system, encapsulate this model in an object that provides an interface that can be manipulated by other components of the system, and hide the details of how the object is implemented from the users of the object. The challenge is to identify abstractions that simultaneously provide a service that proves useful in a large number of situations, and can be efficiently implemented in the underlying system. This of course is exactly what we were doing when we introduced the idea of a channel in the previous section: providing an abstraction for applications that hides the complexity of the network from application writers.

Application Programs
Process-to-Process Channels
Host-to-Host Connectivity
Hardware

Figure: Example layered network system.

Abstractions naturally lead to layering, especially in network systems. The general idea is that one starts with the services offered by the underlying hardware, and then adds a sequence of layers, each providing a higher (more abstract) level of service. The services provided at the high layers are implemented in terms of the services provided by the low layers. Drawing on the discussion of requirements given the previous section, for example, one might imagine a network as having two layers of abstraction sandwiched between the application program and the underlying hardware, as illustrated in Figure . The layer immediately above the hardware in this case might provide host-to-host connectivity, abstracting away the fact that there may be an arbitrarily complex network topology between any two hosts. The next layer up builds on the available host-to-host communication service and provides support for process-to-process channels, abstracting away the fact that the network occasionally loses messages, for example.

Application Programs	
Request/Reply Channel	Message Stream Channel
Host-to-Host Connectivity	
Hardware	

Figure: Layered system with alternative abstractions available at a given layer.

Layering provides two nice features. First, it decomposes the problem of building a network into more manageable components. Rather than implementing a monolithic piece of software that does everything you will ever want, you can implement several layers, each of which solves one part of the problem. Second, it provides a more modular design. If you decide that you want to add some new service, you may only need to modify the functionality at one layer, re-using the functions provided at all the other

layers.

Thinking of a system as a linear sequence of layers is an over simplification, however. Many times there are multiple abstractions provided at any given level of the system, each providing a different service to the higher layers, but building on the same low-level abstractions. To see this, consider the two types of channels discussed in Section 1.1: one provides a request/reply service and one supports a message-stream service. These two channels might be alternative offerings at some level of a multi-level networking system, as illustrated in Figure 1.2.

Using this discussion of layering as a foundation, we are now ready to to discuss the architecture of a network more precisely. For starters, the abstract objects that make up the layers of a network system are called *protocols*. That is, a protocol provides a communication service that higher-level objects (such as application processes, or perhaps higher level protocols) use to exchange messages. For example, one could imagine a network that supports a request/reply protocol and a message-stream protocol, corresponding to the request/reply and message-stream channels discussed above.

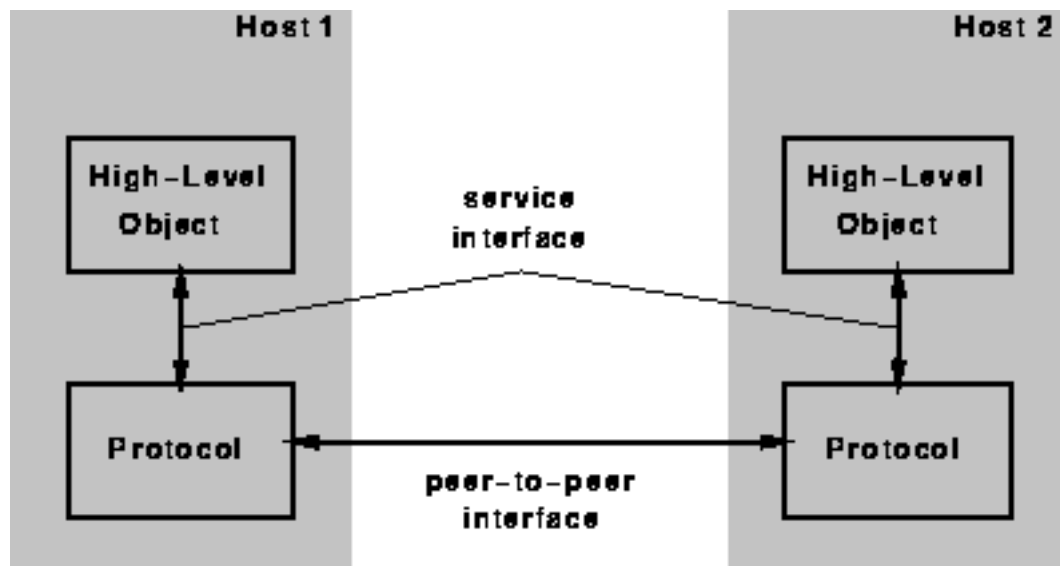



Figure: Service and peer interfaces.

Each protocol defines two different interfaces. First, it defines a *service* interface to the other objects on the same computer that want to use its communication services. This service interface defines the operations that local objects can perform on the protocol. For example, request/reply protocol would support operations by which an application can send and receive messages. Second, a protocol defines a *peer* interface to its counterpart (peer) on another machine. This second interface defines the form and meaning of messages exchanged between protocol peers to implement the communication service. This would determine the way in which a request/reply protocol on one machine communicates with its peer on another machine. In other words, a protocol defines a communication service that it exports locally, along with a set of rules governing the messages that the protocol exchanges with its peer(s) to implement this service. This situation is illustrated in Figure 1.2.

Except at the hardware level where peers directly communicate with each other over a link, peer-to-peer communication is indirect---each protocol communicates with its peer by passing messages to some lower level protocol, which in turn delivers the message to *its* peer. What is more, there are potentially multiple protocols at any given level, each providing a different communication service. We therefore represent the suite of protocols that make up a network system with a *protocol graph*. The nodes of the graph correspond to protocols and the edges represent a *depends on* relation. For example, Figure  illustrates a protocol graph for the hypothetical layered system we have been discussing---protocols RRP (Request/Reply Protocol) and MSP (Message-Stream Protocol) implement two different types of process-to-process channels, and both depend on HHP (Host-to-Host Protocol), which provides a host-to-host connectivity service.

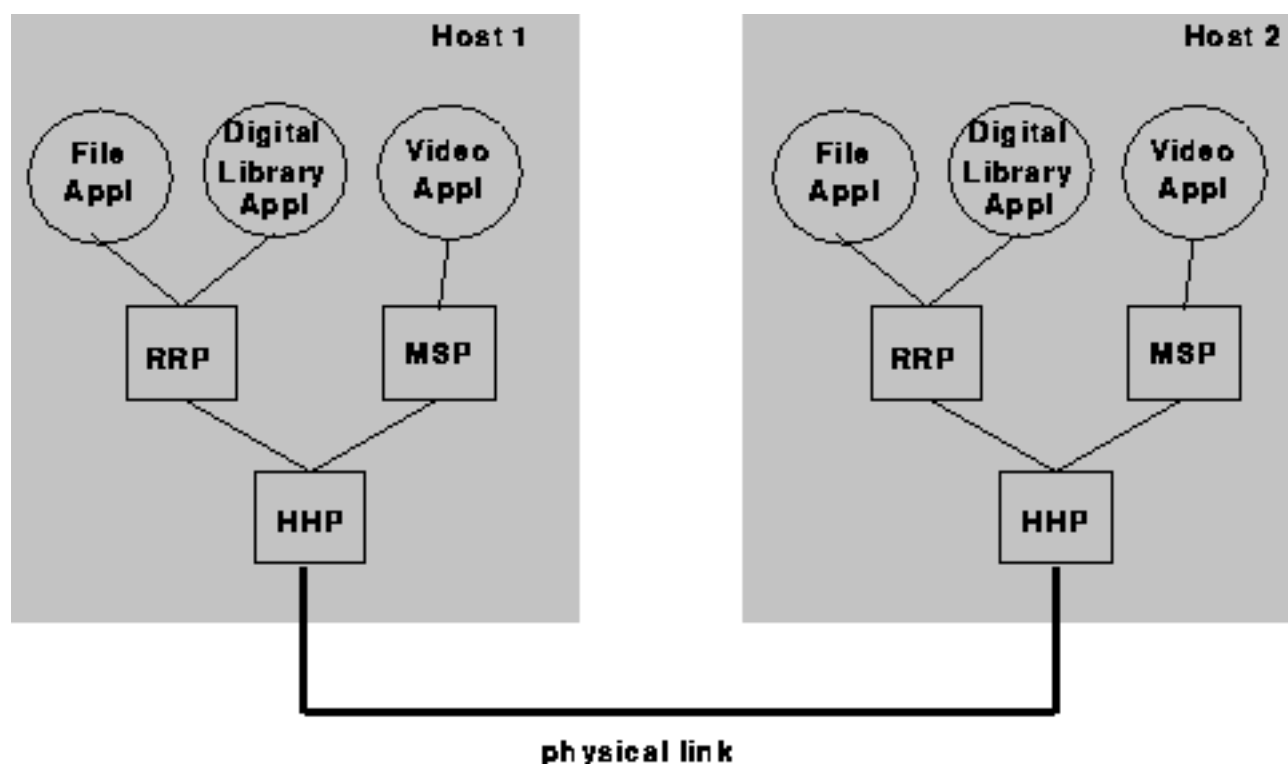


Figure: Example protocol graph.


In this example, suppose the file access program on Host 1 wants to send a message to its peer on Host 2 using the communication service offered by protocol RRP. In this case, the file application asks RRP to send the message on its behalf. To communicate with its peer, RRP then invokes the services of HHP, which in turn directly transmits the message to its peer on the other machine. Once the message has arrived at protocol HHP on Host 2, HHP passes the message up to RRP, which in turn delivers the message to the file application. In this particular case, the application is said to employ the services of the *protocol stack* RRP/HHP.

Note that the term protocol is actually overloaded. Sometimes it refers to the abstract interfaces---that is, the operations defined by the service interface and the form and meaning of messages exchanged between peers---and sometimes it refers to the module that actually implements these two interfaces. To distinguish between the interfaces and the module that implements these interfaces, we generally refer to

the former as a *protocol specification*. Specifications are generally expressed using a combination of prose, pseudo-code, state transition diagrams, pictures of packet formats, and other abstract notations. It should be the case that a given protocol can be implemented in different ways by different programmers, as long as each adheres to the specification. The challenge is ensuring that two different implementations of the same specification can successfully exchange messages. Two or more protocol modules that do accurately implement a protocol specification are said to *interoperate* with each other.

One can imagine many different protocols and protocol graphs that satisfy the communication requirements of a collection of applications. Fortunately, there exist standardization bodies, such as the International Standards Organization (ISO) and the Internet Engineering Task Force (IETF), that establish policies for a particular protocol graph. We call the set of rules governing the form and content of a protocol graph a *network architecture*. Although beyond the scope of this book, standardization bodies like the ISO and the IETF have established well-defined procedures for introducing, validating, and finally approving protocols in their respective architectures. We briefly describe the architectures defined by the ISO and the IETF in a moment, but first, there are two additional things we need to explain about the mechanics of a protocol graph.

Encapsulation

Consider what happens in Figure  when one of the application program sends a message to its peer by passing the message to protocol RRP. From RRP's perspective, the message it is given by the application is an uninterpreted string of bytes. RRP does not care that these bytes represent an array of integers, an email message, a digital image, or whatever; it is simply charged with sending them to *its* peer. However, RRP must communicate control information to its peer, instructing it how to handle the message when it is received. RRP does this by attaching a *header* to the message. Generally speaking, a header is a small data structure---on the order of a few bytes to a few dozen bytes---that is used among peers to communicate with each other. As the name suggests, headers are usually attached to the front of a message. In some cases, however, this peer-to-peer control information is sent at the end of the message, in which case it is called a *trailer*. The exact format for the header attached by RRP is defined by its protocol specification. The rest of the message---i.e., the data being transmitted on behalf of the application---is called the message's *body*. We say that the application's data is *encapsulated* in the new message created by protocol RRP.

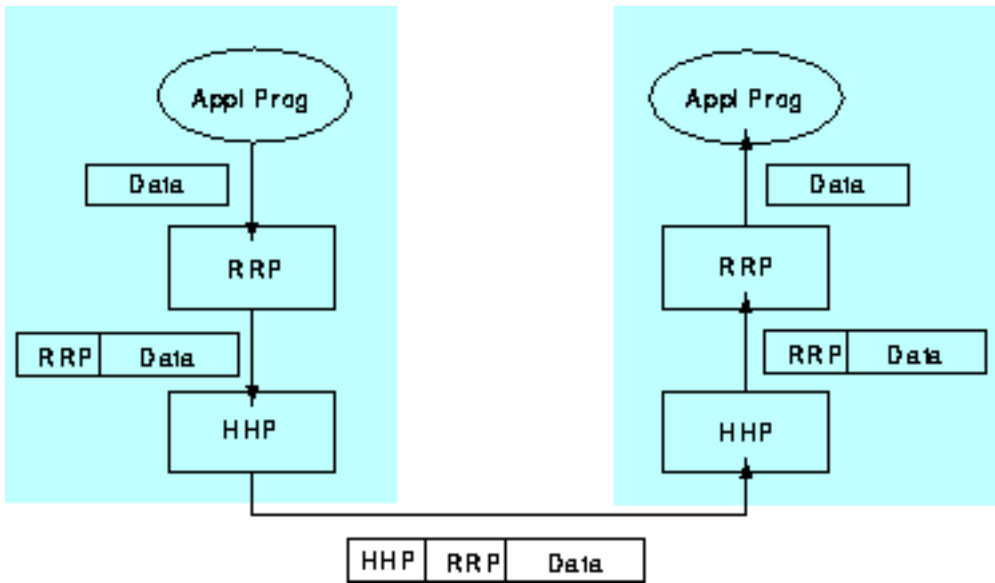





Figure: High-level messages are encapsulated inside of low-level messages.

This process of encapsulation is then repeated at each level of the protocol graph, for example, HHP encapsulates RRP's message by attaching a header of its own. If we now assume that HHP sends the message to its peer over some physical link, then when the message arrives at the destination host, it is processed in the opposite order---HHP first strips its header off the front of the message, interprets it (i. e., takes whatever action is appropriate given the contents of the header), and passes the body of the message up to RRP, which removes the header that its peer attached, takes whatever action is indicated by that header, and passes the body of the message up to the application program. The message passed up from RRP to the application on Host 2 is exactly the same message as the application passed down to RRP on Host 1; the application does not see any of the headers that have been attached to it to implement the lower level communication services. This whole process is illustrated in Figure .

Note that when we say a low-level protocol does not interpret the message it is given by some high-level protocol, we mean that it does not know how to extract any meaning from the data contained in the message. It is sometimes the case, however, that the low-level protocol applies some simple transformation to the data it is given, such as to compress or encrypt it. In this case, the protocol is transforming the entire body of the message, including both the original application's data and all the headers attached to that data by higher level protocols.

Multiplexing and Demultiplexing




Recall from Section  that a fundamental idea of packet switching is to multiplex multiple flows of data over a single physical link. This same idea applies up and down the protocol graph, not just to switching nodes. In Figure , for example, one can think of RRP as implementing a logical communication channel, with messages from two different applications multiplexed over this channel at

the source host, and then demultiplexed back to the appropriate application at the destination host.

Practically speaking, all this means is that the header that RRP attaches to its messages contains an identifier that records the application to which the message belongs. We call this identifier RRP's *demultiplexing key*, or *demux key* for short. At the source host, RRP includes the appropriate demux key in its header. When the message is delivered to RRP on the destination host, it strips its header, examines the demux key, and demultiplexes the message to the correct application.

RRP is not unique in its support for multiplexing; nearly every protocol implements this mechanism. For example, HHP has its own demux key to determine which messages to pass up to RRP and which to pass up to MSP. However, there is no uniform agreement among protocols---even those within a single network architecture---on exactly what constitutes a demux key. Some protocols use an 8-bit field (meaning they can support only 256 high-level protocols), and others use 16 or 32-bit fields. Also, some protocols have a single demultiplexing field in their header, while others have a pair of demultiplexing fields. In the former case, the same demux key is used on both sides of the communication, while in the latter case, each side uses a different key to identify the high-level protocol (or application program) to which the message is to be delivered.

OSI Architecture

The ISO was one of the first organizations to formally define a common way to connect computers. Their architecture, called the *Open Systems Interconnection* (OSI) architecture and illustrated in Figure , defines a partitioning of network functionality into seven layers, where one or more protocols implement the functionality assigned to a given layer. In this sense, the schematic given in Figure  is not a protocol graph, per se, but rather a *reference model* for a protocol graph. The ISO, usually in conjunction with a second standards organization known as the *International Telecommunications Union* (ITU),  publish a series of protocol specifications based on the OSI architecture. This series is sometimes called the ``X dot" series since the protocols are given names like X.25, X.400, X.500, and so on. There have been several networks based on these standards, including the public X.25 network and private networks like Tymnet.

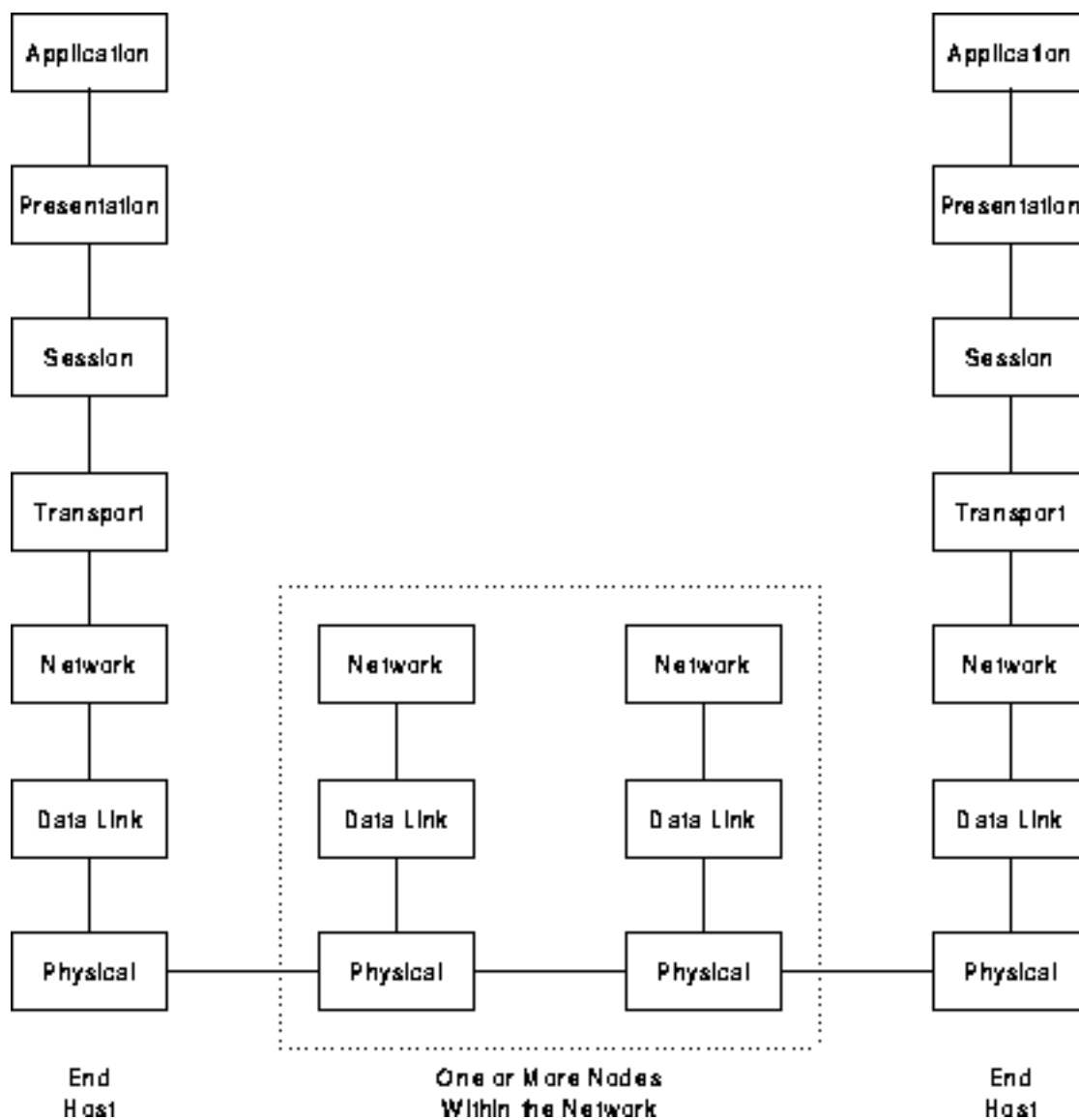




Figure: OSI network architecture.

Starting at the bottom and working up, the *physical* layer handles the transmission of raw bits over a communications link. The *data link* layer then collects a stream of bits into a larger aggregate called a *frame*. Network adaptors typically implement the data link level, meaning that frames, not raw bits, are actually delivered to hosts. The *network* layer handles routing among nodes within a packet-switched network. At this layer, the unit of data exchanged among nodes is typically called a *packet* rather than a frame, although they are fundamentally the same thing. The lower three layers are implemented on all network nodes, including switches within the network, and hosts connected along the exterior of the network. The *transport* layer then implements what we have up to this point been calling a process-to-process channel. Here, the unit of data exchanged is commonly called a *message* rather than a packet or a frame. The transport layer and above typically run only on the end hosts, and not on the intermediate switches or routers.

There is less agreement about the definitions of the top three layers. Working from the top down, the topmost layer is the *application*. Application layer protocols include things like the file transfer protocol, FTP, which defines a protocol by which file transfer applications can interoperate. Next, the

presentation layer is concerned with the format of data exchanged between peers; for example, whether an integer is 16, 32, or 64 bits long, and whether the most significant bit is transmitted first or last. Finally, the *session* layer provides a name space that is used to tie together the potentially different transport streams that are part of a single application. For example, it might manage an audio stream and a video stream that are being combined in a teleconferencing application.

Internet Architecture

The Internet architecture, which is also sometimes called the TCP/IP architecture after its two main protocols, is depicted in Figure . An alternative representation is given in Figure . The Internet architecture evolved out of the experiences with an earlier packet-switched network called the ARPANET. Both the Internet and the ARPANET were funded by the Advanced Research Projects Agency (ARPA), one of the R&D funding agencies of the U.S. Department of Defense. The Internet and ARPANET were around before the OSI architecture, and in fact, the experience gained from building them was a major influence on the OSI reference model.

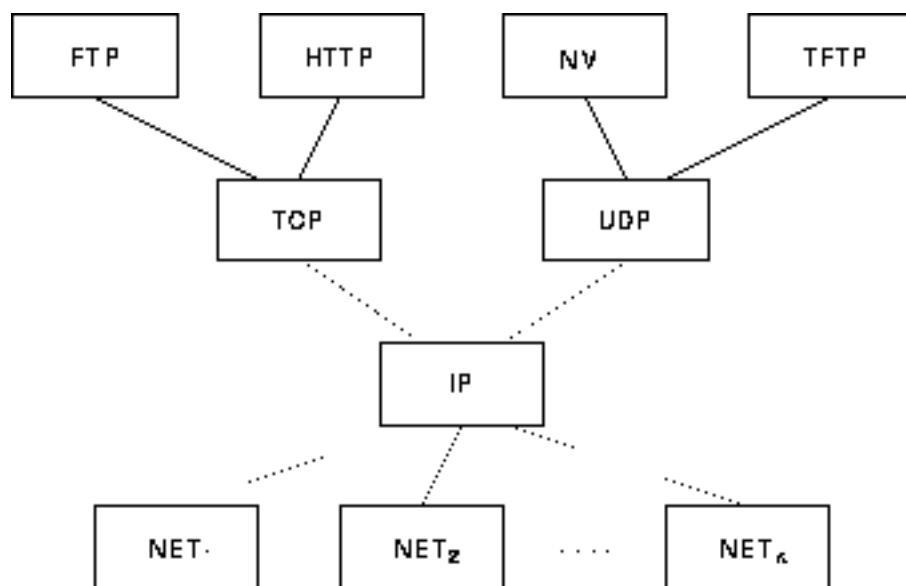


Figure: Internet Protocol Graph.

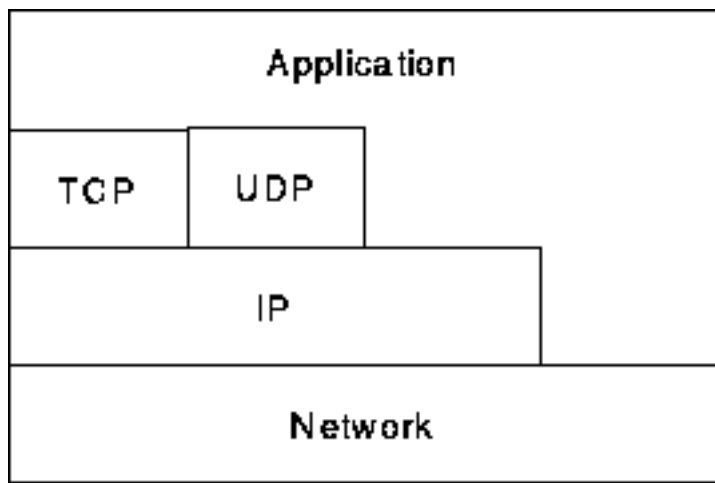




Figure: Alternative View of the Internet Architecture.

While the seven layer OSI model can, with some imagination, be applied to the Internet, a four layer model is often used instead. At the lowest level are a wide variety of network protocols, denoted NET_1, NET_2, and so on. In practice, these protocols are implemented by a combination of hardware (e.g., a network adaptor) and software (e.g., a network device driver). For example, one might find Ethernet or FDDI protocols at this layer. (These protocols in turn may actually involve several sub-layers, but the Internet architecture does not presume anything about them.) The second layer consists of a single protocol---IP, which stands for the *Internet Protocol*. This is the protocol that supports the interconnection of multiple networking technologies into a single, logical internetwork. The third layer contains two main protocols---the *Transmission Control Protocol* (TCP) and the *User Datagram Protocol* (UDP). TCP and UDP provide alternative logical channels to application programs---TCP provides a reliable byte stream channel and UDP provides an unreliable datagram delivery channel (*datagram* may be thought of as a synonym for message). In Internet-speak, TCP and UDP are sometimes called *end-to-end* protocols, although it is equally correct to refer to them as transport protocols.

Running above the transport layer are a range of application protocols that enable the interoperation of popular applications, protocols like FTP, Telnet (remote login), and SMTP (electronic mail). To understand the difference between an application layer protocol and an application, think of all the different World Wide Web browsers that are available (e.g., Mosaic, Netscape, Lynx, etc.). The reason that you can use any one of these application programs to access a particular site on the Web is because they all conform to the same application layer protocol: HTTP. Confusingly, the same word sometimes applies to both an application and the application layer protocol that it uses (e.g., FTP).

The Internet architecture has three features that are worth highlighting. First, as best illustrated by Figure , the Internet architecture does not imply strict layering. The application is free to bypass the defined transport layers and directly use IP or one of the underlying networks. In fact, programmers are free to define new channel abstractions or applications that run on top of any of the existing protocols.

Second, if you look closely at the protocol graph given in Figure , you will notice an *hourglass*

shape---wide at the top, narrow in the middle, and wide at the bottom. This shape actually reflects the central philosophy of the architecture. That is, IP serves as the focal point for the architecture---it defines a common method for exchanging packets among a wide collection of networks. Above IP can be arbitrarily many transport protocols, each offering a different channel abstraction to application programs. Thus, the issue of delivering messages from host-to-host are completely separated from the issue of providing a useful process-to-process communication service. Below IP, the architecture allows for arbitrarily many different network technologies, ranging from Ethernet, to FDDI, to ATM, to single point-to-point links.

A final attribute of the Internet architecture (or more accurately, of the IETF culture) is that in order for someone to propose a new protocol to be included in the architecture, they must produce both a protocol specification, and at least one (and preferably two) representative implementations of the specification. The existence of working implementations is required for standards to be adopted by the IETF. This cultural assumption of the design community helps to ensure that the architecture's protocols can be efficiently implemented. Even then, the implementations of the protocols in the Internet architecture have evolved quite significantly over time as researchers and practitioners have gained experience using them.

Sidebar: OSI versus Internet Architecture

While there have been countless spirited discussions over the past 15-20 years about the technical advantages of the ISO protocols versus the Internet protocols, such debates are no longer relevant. This is because the ISO protocols, with a handful of exceptions, are largely ignored, while the Internet thrives. (This statement does not imply that the OSI reference architecture is worthless---to the contrary, it continues to be somewhat influential by providing a model for understanding networks, even though the ISO protocols themselves have been a commercial failure.)

One explanation is that some of the technical arguments made against the OSI architecture during these debates were on the mark. There is a more likely explanation, however; one that is surprisingly pragmatic in nature. It is that an implementation of the TCP/IP protocol suite was bundled with the popular Unix operating system distributed by the University of California at Berkeley in the early 1980s. People are more likely to use software that is readily available than software that requires effort, and the wide-spread availability of Berkeley Unix allowed the TCP/IP architecture to achieve a critical mass.

To expand on this point a bit further, the ISO/ITU culture has always been to ``specify first and implement later". As the specification process is a long and tedious one, especially when you do not have any working code with which to experiment, the implementation of ISO protocols were always a long time in arriving. In contrast, the

Internet culture has been to ``implement as you go". In fact, to quote a T-shirt commonly worn at IETF meetings:

We reject kings, presidents, and voting. We believe in rough consensus and running code. (Dave Clark)

To understand just how important *working code* is, we observe that it is even more powerful than the U.S. government. In 1988 the National Institute for Standards and Technology (NIST), an agency within the U.S. government's Department of Commerce, approved a mandate that required government agencies to procure equipment that could run the ISO protocols. (The agencies did not have to use the ISO protocols, but the vendors had to support it, or at least demonstrate how their systems could support it.) Since the U.S. government is such a big consumer of computers, this mandate was expected to push the commercial sector towards adopting the ISO protocol suite. In reality, however, computers were shipped with ISO-compliant code, but people kept using TCP/IP. Well, to make a long story short, the mandate was officially rescinded in September 1994.

[Next](#) [Up](#) [Previous](#)

Next: [Plan for the](#) **Up:** [Foundation](#) **Previous:** [Requirements](#)

Plan for the Book

Just as a network architecture provides a blueprint for building a computer network, this section gives the blueprint for reading this book. This blueprint has two dimensions. In one dimension, the chapters give a layer-by-layer account of how to design a computer network. While it is accurate to characterize the sequence of chapters as describing a network from the bottom-up---i.e., starting at the low-levels of the network architecture and moving to the higher layers---we prefer to think of the book as first describing the simplest possible network, and then adding functionality and capabilities in later chapters. A summary of the chapters is given below. In the second dimension, there are a set of common themes that run through all the chapters. We think of these themes as collectively representing a *systems approach* to computer networks, a phrase you should recognize as important based on its being in the book's title. Section [1.1](#) identifies the themes that make up the systems approach to networking.

Chapter Organization

Each chapter begins with a statement of the next problem that must be solved---the individual problems have their genesis in Chapter 1---and then develops and evaluates solutions that can be applied to the problem. In this way, each chapter builds on the foundation developed in the previous chapters, with the end result being a recipe for building a computer network from the ground up.

Before getting to the lowest layers of the network, Chapter [2](#) first takes a brief detour by introducing the the *x*-kernel, a framework for implementing network protocols. While it is not essential that you understand the *x*-kernel in detail, this chapter plays a valuable role in this book. One reason is that we give fragments of *x*-kernel code in later chapters; reading Chapter [2](#) helps to make this code more accessible. Another reason is that is that Chapter [2](#) describes the underlying data structures used to implement various components of a protocol; seeing these data structures helps to ground your understanding of networks at the most basic level. The third, and perhaps most important reason is that this Chapter describes the mechanics of a protocol, and as such, it provides a scaffolding for the

concepts you learn throughout the rest of the book. Without Chapter [1](#), this book contains a comprehensive collection of algorithms and protocols that make up computer networks. With Chapter [2](#), however, you should be able to develop a concrete understanding of how all these algorithms and protocols fit together to provide a functioning computer network.

Chapters [3](#) and [4](#) then explore a variety of networking technologies. Chapter [3](#) begins by considering simple networks in which all the computers are directly connected to each other, either in a point-to-point configuration, or with a shared-access medium such as Ethernet or FDDI. It also discusses the point at which the network link connects to the computer---the network adaptor. Chapter [4](#) then considers switch-based networks in which intermediate nodes forward packets between machines that are not directly connected to each other. Here, routing and contention are the key topics.

Chapter [5](#) then covers the topic of internetworking. Starting with the problem of connecting a handful of Ethernets, then moving on to the simple interconnection of networks as exemplified by the early days of the Internet, and concluding with the problems associated with a global internet like the one that exists today, Chapter [5](#) addresses two main issues---scale and heterogeneity.

Assuming a collection of computers connected by some kind of network---either a particular networking technology or by the global Internet---Chapters [6](#) and [7](#) then look at the end-to-end issues. Chapter [6](#) considers three styles of end-to-end (transport) protocols: unreliable datagram protocols (UDP), byte-stream protocols (TCP), and request/reply protocols (RPC). Chapter [7](#) then focuses on application-specific transformations that are applied to the data being communicated. These transformations include encryption, compression, and presentation formatting.

Chapters [8](#) and [9](#) discuss advanced topics. Chapter [8](#) focuses on the issue of congestion control, which is cast as the problem of fair resource allocation. This chapter looks at both ``inside the network" and ``at the edge of the network" solutions. Chapter [9](#) then introduces the problems that must be addressed as networks become faster---moving into the gigabit-per-second range---and outlines how some of these problems are being solved.

Finally, so as not to leave you with the false impression that the networking community knows everything there is to know about networking, each chapter concludes with a discussion of some unresolved issue---an **Open Issue**. Typically, this is a controversial subject that is currently being debated in the networking community. Hopefully, this section will help to instill the realization that much more work remains to be done, that networking is an exciting and vibrant subject with plenty of challenges yet to be addressed by the next generation of network designers.

Systems Approach

The approach we take throughout all the chapters is to consider networking from a *systems* perspective. While the term "systems approach" is terribly over used, we use it to mean that the book first develops a collection of building blocks, and then shows how these components can be put together to construct a complete network that satisfies the numerous and sometimes conflicting goals of some user community. This systems-oriented perspective has several implications that are reflected in the content, organization, and emphasis of this book.

First, the systems approach implies doing experimental performance studies, and using the data you gather both to quantitatively analyze various design options, and to guide you in optimizing the implementation. This emphasis on empirical analysis pervades the book, as exemplified by the definition of networking's central performance metrics given earlier in this Chapter. Later chapters discuss the performance of the component being studied.

Second, system builders do not accept existing artifacts as gospel, but instead strive to understand the concepts that are fundamental to the system. In the context of this book, this means that starting from first principles, we answer the question of *why* networks are designed the way they are, rather than simply marching through the set of protocols that happen to be in use today. It is our experience that once someone understands the underlying concepts, any new protocol that one is confronted with will be relatively easy to digest.

Third, to help understand the concepts underlying the system in question, system builders often draw on a collection of design principles that have evolved from experience with computer systems in general. For example, a well-known system design rule that we have already seen is to manage complexity by introducing multiple layers of abstraction. As we will see throughout this book, layering is a "trick of the trade" applied quite frequently in the networking domain. Other rules of thumb guide the decomposition of a system into its component layers. We highlight these design principles as they are introduced throughout the book.

Fourth, the systems approach implies studying successful, working examples; systems cannot be studied in the abstract. For example, this book draws heavily---although not exclusively---from the Internet. As a functioning, global network, the Internet is rich with examples that illustrate how networks are designed. It is also the case that the Internet, because it encompasses every known Wide-Area Network (WAN) and Local-Area Network (LAN) technology, lends itself to the broadest possible coverage of networking topics. Note that while we plan to use the Internet as a framework for talking about networks, the book is not simply a tutorial on the Internet. The Internet is an evolving network, and in fact, is at a juncture where change is rapid. Five years from now, it is very unlikely that the Internet will look anything like it does today. Therefore, our goal is to focus on unifying principles, not just recount the specifics of the current Internet.

Fifth, a systems approach implies looking at "the big picture". That is, you need to evaluate design

decisions in terms of how they affect the entire system, not just with the goal of optimizing some small part of the system. In a networking context, this means understanding the entire network architecture before trying to optimize the design of one protocol.

Finally, system design often implies software, and computer networks are no exception. Although the primitive building blocks of a network are the commodity hardware that can be bought from computer vendors and communication services that can be leased from the phone company, it is software that turns this raw hardware into a richly functional, robust, and high-performance network. As mentioned above, one goal of this book is to provide the reader not only with an understanding of the components of a network, but also a feel for how all the pieces are put together. Code from the *x*-kernel---a working network subsystem---helps to make this all tangible.

Next	Up	Previous
------	----	----------

Next: [Summary](#) **Up:** [Foundation](#) **Previous:** [Network Architecture](#)

Summary

Computer networks like the Internet have experienced explosive growth over the past decade, and are now positioned to provide a wide range of services---remote file access, digital libraries, video conferencing---to tens of millions of users. Much of this growth can be attributed to the general-purpose nature of computer networks, and in particular, to the ability to add new functionality to the network by writing software that runs on affordable, high-performance computers. With this in mind, the overriding goal of this book is to describe computer networks in such a way that when you finish reading it, you should feel that if given an army of programmers at your disposal, you could actually build a fully functional computer network from the ground up. This chapter lays for foundation for realizing this goal.

The first step we have taken towards this goal is to carefully identify exactly what we expect from a network. For example, a network must first provide cost-effective connectivity among a set of computers. This is accomplished through a nested interconnection of nodes and links, and by sharing this hardware base through the use of statistical multiplexing. This results in a packet-switched network, on top of which we then define a collection of process-to-process communication services. Finally, the network as a whole must offer high performance, where the two performance metrics we are most interested in are latency and throughput.

The second step is to define a layered architecture that will serve as a blueprint for our design. The central objects of this architecture are network protocols. Protocols both provide a communication service to higher level protocols, and define the form and meaning of messages exchanged with their peers running on other machines. We have briefly surveyed two of the most widely used architectures: the OSI architecture and the Internet architecture. This book most closely follows the Internet architecture, both in its organization and as a source of examples.

Copyright 1996, Morgan Kaufmann Publishers

Open Issue: Ubiquitous Networking

There is little doubt that computer networks are on the brink of becoming an integral part of our everyday lives. What began over 20 years ago as experimental systems like the ARPANET---connecting mainframe computers over long-distance telephone lines---has turned into big business. And where there is big business, there are lots of players. In this case, there is the computing industry, which has become increasingly involved in supporting packet-switched networking products; the telephone carriers, which recognize the market for carrying all sorts of data, not just voice; and the TV cable industry, which currently owns the entertainment portion of the market.

Assuming the goal is ubiquitous networking---to bring the network into every household---the first problem that must be addressed is how to establish the necessary physical links. Many would argue that the ultimate answer is to bring an optical fiber into every home, but at an estimated \$1000 per house and 100 million homes in the US alone, this is a \$100 billion proposition. The most widely discussed alternative is to use the existing cable TV facilities. The problem is that today's cable facilities are asymmetric---you can deliver 150 channels into every home, but the outgoing bandwidth is severely limited. Such asymmetry implies that there are a small collection of *information providers*, but that most of us are simply *information consumers*. Many people would argue that in a democracy, we should all have an equal opportunity to provide information.

How the computer companies versus the telephone companies versus the cable industry struggle will play out in the market-place is anyone's guess. (If we knew the answer, we'd be charging a lot more for this book.) All that know is that there are many technical obstacles that stand between the current state-of-the-art, and the sort of global, ubiquitous, heterogeneous network that we believe is possible and desirable---issues of connectivity, levels of service, performance, reliability, and fairness. It is these challenges that are the focus of this book.

Copyright 1996, Morgan Kaufmann Publishers

Further Reading

Computer networks are not the first communication-oriented technology to find their way into the everyday fabric of our society. For example, the early part of this century saw the introduction of the telephone, and then during the 1950's, television became wide-spread. When considering the future of networking---how widely it will spread and how we will use it---it is instructive to study this history. Our first reference is a good starting point for doing this (the entire issue is devoted to the first 100 years of telecommunications).

The second and third papers are the seminal papers on the OSI and Internet architectures, respectively, where the Zimmerman paper introduces the OSI architecture and the Clark paper is a retrospective. The final two papers are not specific to networking, but ones that every systems person should read. The Saltzer *et. al.* paper motivates and describes one of the most widely applied rules of system design---the *end-to-end argument*. The paper by Mashey describes the thinking behind RISC architectures, but as we will soon discover, making good judgments about where to place functionality in a complex system is what system design is all about.

- J. Pierce. Telephony---A Personal View. *IEEE Communications Magazine*, Vol. 22, No. 5 (May 1984), 116--120.
- H. Zimmerman. OSI Reference Model---The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications*, COM-28(4): 425-432, April 1980.
- D. Clark. The Design Philosophy of the DARPA Internet Protocols. *Proceedings of the SIGCOMM '88 Symposium*, pages 106--114, August 1988.
- J. Saltzer, et. al. End-to-End Arguments in System Design. *ACM Transactions on Computer Systems*, 2(4): 277--288, November 1984.
- J. Mashey. RISC, MIPS, and the Motion of Complexity. In *UniForum 1986 Conference Proceedings* (1986), 116--124.

Several texts offer an introduction to computer networking. Stallings gives an encyclopedic treatment of

the subject, with an emphasis on the lower levels of the OSI hierarchy [[Sta91](#)]; Tanenbaum uses the OSI architecture as a organizational model [[Tan88](#)]; Comer gives an overview of the Internet architecture [[Com95](#)]; and Bertsekas and Gallager discuss networking from a performance modeling perspective [[BG92](#)].

For more on the social ramifications of computer networking, articles can be in most any newspaper or periodical. Two specific examples that we recommend are Barlow's commentary on the risks of asymmetric networks [[Bar95](#)], and an overview of the "Information Revolution" published in the National Geographic [[Swe95](#)]. (The latter has nice pictures.) Also, we recommend the "net surf" column in *Wired* for interesting WWW links.

To put computer networking into a larger context, two books---one dealing with the past and the other looking towards the future---are must reading. The first is Holzmann and Pehrson's *The Early History of Data Networks* [[HP95](#)]. Surprisingly, many of the ideas covered in the book you are now reading were invented during the 1700's. The second is *Realizing the Information Future: The Internet and Beyond*, a book prepared by the Computer Science and Telecommunications Board of the National Research Council [[Cou94](#)].

A good introduction to the Internet---its growth and some of the new applications that are making it more popular---can be found in the August 1994 issue of the *Communications of the ACM* [[IL94](#)]. To follow the history of the Internet from its beginning, the reader is encouraged to peruse the Internet's *Request for Comments* (RFC) series of documents. These documents, which include everything from the TCP specification to April Fools jokes, are retrievable on the Internet by doing anonymous `ftp` to hosts `ds.internic.net`, `nic.ddn.mil`, or `ftp.isi.edu`, and changing directory to `rftc`. For example, the protocol specifications for TCP, UDP, and IP are available in RFC's 793, 768, and 791, respectively.

To gain a better appreciation for the Internet philosophy and culture, two references are must reading; both are also quite entertaining. Padlipsky gives a good description of the early days, including a pointed comparison of the Internet and OSI architectures [[Pad85](#)]. For a more up-to-date account of what really happens behind the scenes at the IETF, we recommend [[Boo95](#)].

Finally, we conclude the "Further Reading" section of each Chapter with a set of *live* references, that is, URL's for locations on the World-Wide-Web where you can learn more about the topics discussed in that Chapter. Since these references are live, it is possible that they will not remain active for an indefinite period of time. For this reason, we limit the set of live references at the end of each chapter to sites that either export software, provide a service, or report on the activities of an ongoing working group or standardization body. In other words, we only give URL's for the kinds of material that cannot be easily referenced using standard citations. For this Chapter, we include four live references:

- <http://www.mkp.com>: information about this book, including supplements, addendums,

and so on;

- <http://www.acm.org/sigcomm/sos.html>: status of various networking standards, including those of the IETF, ISO, and IEEE;
- <http://www.ietf.cnri.reston.va.us/home.html>: information about the IETF and its working groups;
- <http://www.research.att.com/#netbib>: searchable bibliography of network-related research papers.

Next	Up	Previous
------	----	----------

Next: [Exercises](#) **Up:** [Foundation](#) **Previous:** [Open Issue: Ubiquitous](#)

Exercises

1. Use anonymous FTP to connect to `ds.internic.net`, and retrieve the RFC index. Also retrieve the protocol specifications for TCP, IP, and UDP.
2. Become familiar with a WWW browser such as Mosaic or Netscape. Use the browser to connect to

`http://www.cs.arizona.edu/xkernel/www`

Here you can read about current network research underway at the University of Arizona, and look at a picture of author Larry Peterson. See if you can follow links to find a picture of author Bruce Davie.

3. Learn about a WWW search tool, and then use this tool to locate information about the following topics: MBone, ATM, MPEG, and IPv6.
4. The Unix utility `whois` can be used to find the domain name corresponding to an organization, or vice versa. Read the man page for `whois` and experiment with it. Try `whois arizona.edu` and `whois tucson`, for starters.
5. Discover what you can about the network technologies used within your department or at your campus:
 1. Identify the kinds of hardware being used; e.g., link types, switches, routers.
 2. Identify the high-level protocols that are supported.
 3. Sketch the network topology.
 4. Estimate the number of hosts that are connected.
6. Compare and contrast a packet-switched network and a circuit-switched network. What are the relative advantages and disadvantages of each?
7. One useful property of an address is that it be *unique*; without uniqueness, it would be impossible to distinguish between nodes. List other useful properties that addresses might have.

8. Give an example of a situation where multicast addresses might be beneficial.
9. Explain why STDM is a cost-effective form of multiplexing for a voice telephone network and FDM is a cost-effective form of multiplexing for television and radio networks, and yet we reject both as not being cost-effective for a general-purpose computer network.
10. How "wide" is a bit on a 1Gbps link?
11. How long does it take to transmit x KB over a y Mbps link? Give answer as a ratio of x and y .
12. Suppose a 100Mbps point-to-point link is being set up between the Earth and a new lunar colony. The distance from the moon to the Earth is approximately 240,000 miles, and data travels over the link at the speed of light---186,000 miles per second.
 1. Calculate the minimum RTT for the link.
 2. Using the RTT as the delay, calculate the delay x bandwidth product for the link.
 3. What is the significance of the delay x bandwidth product computed in (b)?
 4. A camera on the lunar base takes pictures of the Earth and saves them in digital format to disk. Suppose Mission Control on Earth wishes to download the most current image, which is 25 MB. What is the minimum amount of time that can elapse between the time that the request for the data goes out and the transfer is finished?
13. The Unix utility `ping` can be used to find the RTT to various Internet hosts. Read the man page for `ping`, and use it to find the RTT to `cs.arizona.edu` in Tucson, and `thumper.bellcore.com` in New Jersey.
14. For each of the following operations on a remote file server, explain whether they are more likely to be delay-sensitive or bandwidth-sensitive.
 1. Open a file.
 2. Read the contents of a file.
 3. List the contents of a directory.
 4. Display the attributes of a file.
15. Use the idea of jitter to support the argument that the telephone network should be circuit-switched. Can a voice application be implemented on a packet-switched network? Explain your answer.
16. Suppose an application program running on a host connected to an Ethernet sends a message over

the TCP/IP Internet. Sketch the message at the time it leaves the source host, including any headers that have been attached to it.

17. Identify at least three different organizations that define network-related standards, and list some of the standards that they are responsible for.
18. Learn about the history of the ARPANET, and its successor, NSFNET. Identify the original ARPANET and NSFNET nodes.
19. Discuss the limitations of building a packet-switched network on top of the existing CATV network. What technical challenges must be addressed to make such a network a reality.
20. Suppose networks do achieve ubiquity in the next few years. Discuss the potential impact on our daily lives, and on society as a whole.



Next: [Protocol Implementation](#) **Up:** [Foundation](#) **Previous:** [Further Reading](#)

Protocol Implementation

- [Problem: Protocols Have to be Implemented](#)
 - [Object-Based Protocol Implementation](#)
 - [Protocols and Sessions](#)
 - [Messages](#)
 - [Events](#)
 - [Id Map](#)
 - [Example Protocol](#)
 - [Summary](#)
 - [Open Issue: Cost of Network Software](#)
 - [Further Reading](#)
 - [Exercises](#)
-

Problem: Protocols Have to be Implemented

Network architectures and protocol specifications are useful things, but the fact remains that a large part of building a computer network has to do with implementing protocols in software. There are two questions we have to ask about this software. One is what language/compiler are we using to implement the network software? The answer to this question is quite often C. It is the dominant language for implementing systems software in general, and network software is most certainly systems software. This is no accident. C gives the programmer the ability to manipulate data at the bit level (this is often called bit-twiddling), which is necessary when implementing network software.

The second question is what operating system (OS) does the network software use? The answer to this question is "whatever OS runs on the node in question." Perhaps more importantly, it is necessary to understand the role played by the OS. Simply stated, network software depends on many of the services provided by the OS, and in fact, network software almost always runs as part of the OS. For example, network protocols have to call the network device driver to send and receive messages, the system's timer facility to schedule events, and the system's memory subsystem to allocate buffers for messages. In addition, network software is generally a complex concurrent program that depends on the operating system for process support and synchronization. Thus, a network programmer needs to be not only an expert at network protocols, but also an operating system guru.

While each operating system provides its own set of routines to read the clock, allocate memory, synchronize processes, and so on, we use a specific operating system---the *x*-kernel---to make this book as tangible as possible. We have chosen to use the *x*-kernel because it was explicitly designed to help out network programmers who are *not* operating system experts. The hope is that by using the network-centric abstractions and interfaces provided by the *x*-kernel, protocol implementors can concentrate on the algorithms that make up network software. The good news is that because the *x*-kernel is efficient, protocols implemented in the *x*-kernel are also efficient.

By the end of this chapter, you probably won't be an OS expert (unless you were one already at the start) but you should know enough about the *x*-kernel to be able to understand the code segments that appear in the remainder of the book. This will establish the groundwork needed in later chapters to make the discussion of various algorithms and mechanisms more concrete.


Just as important as providing background material, this chapter also serves to crystallize the abstract concepts introduced in Chapter [□](#). This is because the *x*-kernel is simply a codification of the fundamental components that are common to all protocols. Thus, this chapter introduces a set of objects and operations available in the *x*-kernel which correspond to the fundamental concepts of networking introduced in the previous chapter. These concepts include protocols, channels, and messages, as well as operations like demultiplexing and encapsulation. It is this codification of fundamental protocol components that makes the *x*-kernel ideal for learning about network programming.

Next	Up	Previous
----------------------	--------------------	--------------------------

Next: [Object-Based Protocol Implementation](#) **Up:** [Protocol Implementation](#) **Previous:** [Protocol Implementation](#)

Object-Based Protocol Implementation

The *x*-kernel provides an object-based framework for implementing protocols. By using the term “object-based, we are not just being fashionable; this is actually an important attribute of the *x*-kernel. While we could probably devote several chapters to developing a precise definition of object-based, it is sufficient to say here that the key abstractions of the *x*-kernel are represented by objects, most of which we will describe in the following sections. An object may be conveniently thought of as a data structure with a collection of operations that are *exported*, that is, made available for invocation by other objects. Objects that have similar features are grouped into classes, with two obvious object classes for a protocol implementation environment being the *protocol* and the *message*.

To better understand the role played by *x*-kernel, recall from Chapter  that a protocol is itself an abstract object, one that exports both a service interface and a peer-to-peer interface. The former defines the operations by which other protocols on the same machine invoke the services of this protocol, while the latter defines the form and meaning of messages exchanged between peers (instances of the same protocol running on different machines) to implement this service. In a nutshell, the *x*-kernel provides a concrete representation for a protocol's service interface. While the protocol's specification defines what it means to send or receive a message using the protocol's service interface, the *x*-kernel defines the precise way in which these operations are invoked in a given system. For example, the *x*-kernel's operations for sending and receiving a message are `xPush` and `xPop`, respectively. An *x*-kernel protocol object would consist of an implementation of `xPush` and `xPop` that adheres to what the protocol specification says it means to send and receive messages using this protocol. In other words, the protocol specification defines the *semantics* of the service interface, while the *x*-kernel defines one possible *syntax* for that interface.

The fashionableness of object-based programming has been accompanied by a proliferation of object-oriented languages, of which C++ is probably the most well known example. The *x*-kernel, however, is written in C, which is not considered to be an object-oriented language. To deal with this, the *x*-kernel provides its own object infrastructure---the glue that makes it possible for one object to invoke an operation on another object. For example, when an object invokes the operation `xOpen` on some protocol `p`--- `xOpen` is the *x*-kernel operation for opening a communication channel---it includes `p` as one of the arguments to this operation. The *x*-kernel's object infrastructure takes care of invoking the

actual procedure that implements this operation on protocol p . In other words, the x -kernel uses `xOpen` (p, \dots) in place of the more conventional notation for invoking an operation on an object: p . `xOpen` (\dots).

Before getting into the details of the x -kernel, we have one word of caution---this chapter is not a substitute for an x -kernel programmer's manual. This is because we have taken certain liberties---swept a few uninteresting details under the rug and left out certain features---for the sake of improving the exposition. If you are trying to debug an x -kernel protocol you have written, then you should see the unabridged programmer's manual. Conversely, it is not necessary to understand every detail in this chapter to benefit from the rest of the book; you may want to skim it and return to it later if you find a code segment that seems impenetrable in a subsequent chapter.

...text deleted...

Next	Up	Previous
------	----	----------

Next: [Protocols and Sessions](#) **Up:** [Protocol Implementation](#) **Previous:** [Problem: Protocols Have](#)

Protocols and Sessions

Recall from Chapter [1](#) that a network architecture defines a protocol graph, with each protocol in the graph implementing an abstract channel (pipe) through which data can be sent. The *x*-kernel provides a framework for implementing this graph, including the functionality of each protocol in the graph. To help make the description of this framework more concrete, the following discussion focuses on the protocol graph illustrated in Figure [1.1](#); it is a portion of a protocol graph defined by the Internet architecture.

The two main classes of objects supported by the *x*-kernel are *protocols* and *sessions*. Protocol objects represent just what you would expect---protocols such as IP or TCP. Session objects represent the local end-point of a channel, and as such, typically implement the code that interprets messages and maintains any state associated with the channel. The protocol objects available in a particular network subsystem, along with the relationships among these protocols, is defined by a protocol graph at the time a kernel is configured. Session objects are dynamically created as channels are opened and closed. Loosely speaking, protocol objects export operations for opening channels---resulting in the creation of a session object---and session objects export operations for sending and receiving messages.

The set of operations exported by protocol and session objects is called the *uniform protocol interface*---it defines how protocols and sessions invoke operations on each other. At this stage, the important thing to know about the uniform protocol interface is that it specifies how high-level objects invoke operations on low-level objects to send outgoing messages, as well as how low-level objects invoke operations on high-level objects to handle incoming messages. For example, consider the specific pair of protocols TCP and IP in the Internet architecture, depicted in Figure [1.2](#). TCP sits directly above IP in this architecture, so the *x*-kernel's uniform protocol interface defines the operations that TCP invokes on IP, as well as the operations IP invokes on TCP, as illustrated in Figure [1.3](#).

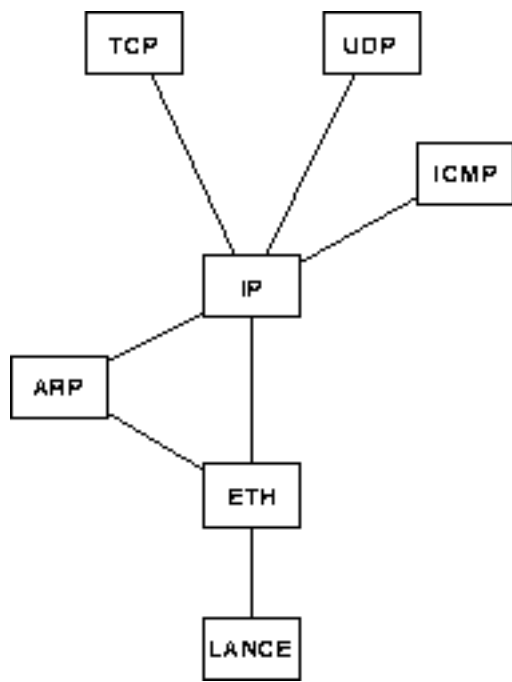


Figure: Example Protocol Graph.

Keep in mind that the following discussion defines a common *interface* between protocols, that is, the operations one protocol is allowed to invoke on the other. This is only half the story, however. The other half is that each protocol has to provide a routine that *implements* this interface. Thus, for an operation like `xOpen`, protocols like TCP and IP must include a routine that implements `xOpen`; by convention, we name this routine `tcpOpen` and `ipOpen`, respectively. Therefore, the following discussion not only defines the interface to each operation, but it also gives a rough outline of what every protocol's implementation of that operation is expected to do.

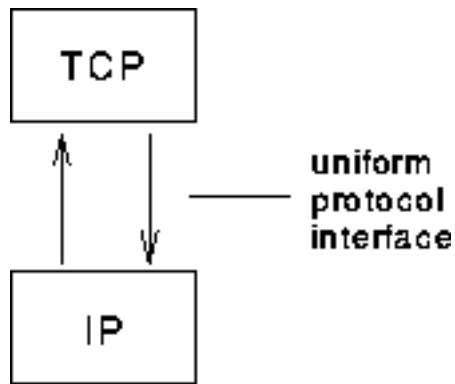


Figure: Uniform Protocol Interface.


Configuring a Protocol Graph

Before presenting the operations that protocol and session objects export, we first explain how a protocol programmer configures a protocol graph. As discussed in Chapter [4](#), standardization bodies

like the ISO and the IETF define a particular network architecture that includes a specific set of protocols. In the Internet architecture, for example, TCP depends on IP, by definition. This suggests that it is possible to ``hard-code" TCP's dependency on IP into the TCP implementation. While this could be done in the case of TCP, the *x*-kernel supports a more flexible mechanism for configuring a protocol graph. This makes it easy to plug protocols together in different ways. While this is a quite powerful thing to be able to do, one has to be careful that it makes sense to have any two protocols adjacent to each other in the protocol graph.

Quite simply, a user that wants to configure a protocol graph specifies the graph with a text file of the following form:

```
name=lance;  
name=eth protocols=lance;  
name=arp protocols=eth;  
name=ip protocols=eth,arp;  
name=icmp protocols=ip;  
name=udp protocols=ip;  
name=tcp protocols=ip;
```

This specification results in the protocol graph depicted in Figure . In this example graph, *lance* and *eth* combine to implement an Ethernet device driver: *lance* is the device-specific half and *eth* is the device-independent half. Also, *arp* is the Address Resolution Protocol (it is used to translate IP addresses into Ethernet addresses) and *icmp* is the Internet Control Message Protocol (it sends error messages on behalf of TCP and IP). The *name* field in each line specifies a protocol (by name) and the *protocols* field says which other protocols this protocol depends on. Not shown in this example is a *dir* field that identifies the directory where the named protocol implementation can be found; by default, it is the same as the name of the protocol. The *x*-kernel build program, called *compose*, parses this specification, and generates some C code that initializes the protocol graph when the system is booted.

Operations on Protocol Objects

The primary operation exported by a protocol object allows a higher level entity to *open* a channel to its peer. The return value from an open operation is a session object. Details about the session object are discussed in the following subsection. For now, think of a session as a convenient object for gaining access to the channel; the module that opened the protocol object can send and receive messages using the session object. An object that lets us gain access to something abstract is sometimes called a *handle*---you can think of it as the thing that makes it easy to grab something that is otherwise quite slippery. Thus, a session object provides a handle on a channel.

In the following discussion, we need a generic way to refer to the entity that opens a channel, since sometimes it is an application program, and sometimes its another protocol. We use the term *participant* for this purpose. That is, we think in terms of a pair of participants at level *i* communicating over a channel implemented by a protocol at level *i-1* (where the level number decreases as you move down the stack).

Opening a channel is an asymmetric activity. Generally, one participant initiates the channel (we call this the client). This local participant is therefore able to identify the remote participant, and is said to do an *active* open. In contrast, the other participant accepts the channel (we call it the server). This participant does not know what clients might try to talk to it until one of them actually makes contact. The server, therefore, does a *passive* open---it says to the lower level protocol that it is willing to accept a channel, but does not say with whom the channel will be.

Thus, the exact form of the open operation depends on whether the higher level entity is doing an active open or a passive open. In the case of an active open, the operation is:

```
Sessn xOpen(Protl hlp, Protl llp, Part *participants)
```

This operation says that high-level protocol *hlp* is opening low-level protocol *llp* so as to establish a channel with the specified *participants*. For a typical channel between a pair of participants, this last argument would contain both the local participant's address and the remote participant's address. The low-level protocol does whatever is necessary to establish the channel, which quite often implies opening a channel on a still lower-level protocol. Notice that *Protl* and *Sessn* are the C type definitions for protocol and session objects, respectively.

A high-level protocol passively opens a low-level protocol with a pair of operations:

```
XkReturn xOpenEnable(Protl hlp, Protl llp, Part *participant)
```

```
XkReturn xOpenDone(Protl hlp, Protl llp, Sessn session,  
                  Part *participants)
```

xOpenEnable is used by high-level protocol *hlp* to inform low-level protocol *llp* that it is willing to accept a connection. In this case, the high-level protocol usually specifies only a single participant---itself. The *xOpenEnable* operation returns immediately; it does not block waiting for a remote site to try to connect to it. The low-level protocol remembers this enabling; when some remote participant subsequently connects to the low-level protocol, *llp* calls the high-level protocol's *xOpenDone* operation to inform it of this event. The low-level protocol *llp* passes the newly created session as an argument to high-level protocol *hlp*, along with the complete set of *participants*, thereby informing the high-level protocol of the address for the remote entity that just connected to it. *XkReturn* is the return value of all the uniform protocol interface operations except for *xOpen*; it indicates whether the operation was successful (*XK_SUCCESS*) or not (*XK_FAILURE*).

In addition to these operations for opening a connection, *x*-kernel protocol objects also support an operation for demultiplexing incoming messages to the appropriate channel (session). In this case, a low-level session invokes this operation on the high-level protocol that at some earlier time had opened it. The operation is:

```
XkReturn xDemux(Prot1 hlp, Sessn lls, Msg *message)
```

It will be easier to understand how this operation is used after we look at session objects in more detail.


Operations on Session Objects

As already explained, a session can be thought of as a handle on a channel that is implemented by some protocol. One can also view it as an object that exports a pair of operations: one for sending messages, and one for receiving messages:

```
XkReturn xPush(Sessn lls, Msg *message)
```


```
XkReturn xPop(Sessn hls, Sessn lls, Msg *message, void *hdr)
```

The implementation of `xPush` and `xPop` is where the real work of a protocol is carried out---it's where headers are added to and stripped from messages, and then interpreted. In short, these two routines implement the algorithm that defines the protocol.

The operation of `xPush` is fairly straightforward. It is invoked by a high-level session to pass a message down to some low-level session (`lls`) that it had opened at some earlier time. `lls` then goes off and does what is needed with the message---perhaps using `xPush` to pass it down to a still lower level session. This is illustrated in Figure . In this figure we see three sessions, each of which implements one protocol in a stack, passing a message down the stack using `xPush`.

Passing messages back up the stack using `xPop` is more complicated. The main problem is that a session does not know what session is above it---all it knows is the protocol that is above it. So, a low-level session `lls` invokes the `xDemux` routine of the protocol above it. That protocol, since it did the work of opening the high level session `hls` to which this message needs to go, is able to pass the message to `hls` using its `xPop` routine. How does a protocol's `xDemux` routine know which of its potentially many sessions to pass the message up to? It uses the *demultiplexing key* found in its header.

In addition to the `hls` that is being called and the message being passed to it, `xPop` takes two other arguments. First, `lls` identifies the low-level session that handed up this message via `xDemux`. Second,

since xDemux has to inspect the message header to select the session on which to call xPop---i.e., it has already gone to the effort of extracting the protocol's header---it passes the header (`hdr`) as the final argument to xPop. This chain of events is illustrated in Figure .

To see how this works in practice, imagine we want to send a message using the TCP and IP protocols. An application program opens a channel by performing xOpen on TCP; TCP returns a session object to the application. TCP opens an IP channel by performing xOpen on IP; IP returns a session object to TCP. When the application wants to send a message, it invokes the xPush operation of the TCP session; this session in turn invokes the xPush operation of the IP session, which ultimately causes the message to be sent.

Now suppose an incoming message is delivered to the IP session. This session has no idea about the TCP session above it, so it does the only thing it knows how to do---it passes the message up to the TCP *protocol* using xDemux. The TCP protocol knows about all the TCP sessions, and so passes the message up to the appropriate TCP session using xPop. TCP's xDemux uses the demux key it found in the TCP header to select among all the TCP sessions.

The final operation that we need to be able to perform is one to close a session, which in effect closes the channel to the other machine.

```
XkReturn xClose(Sessn session)
```

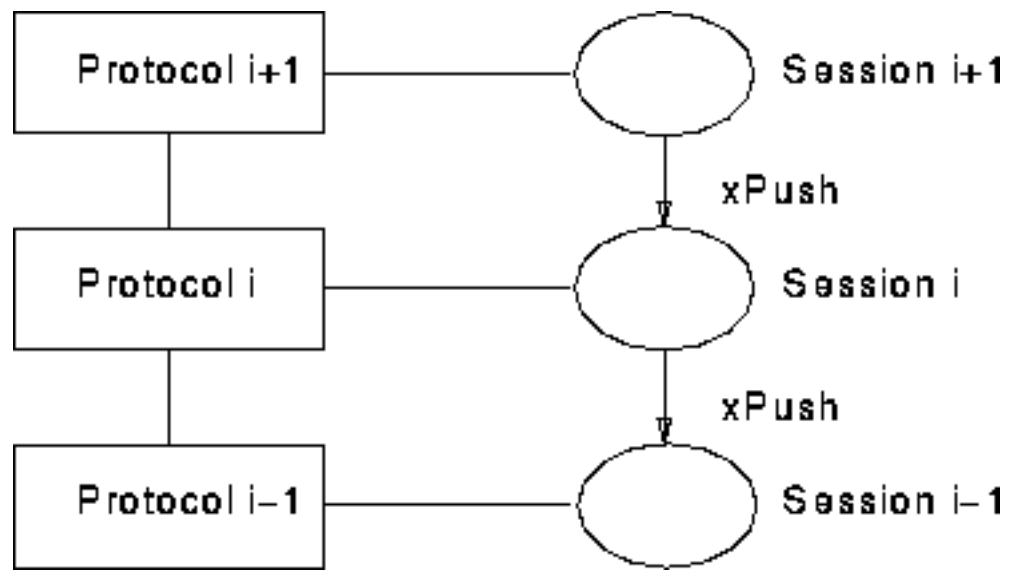


Figure: Using xPush to pass a message down a stack.

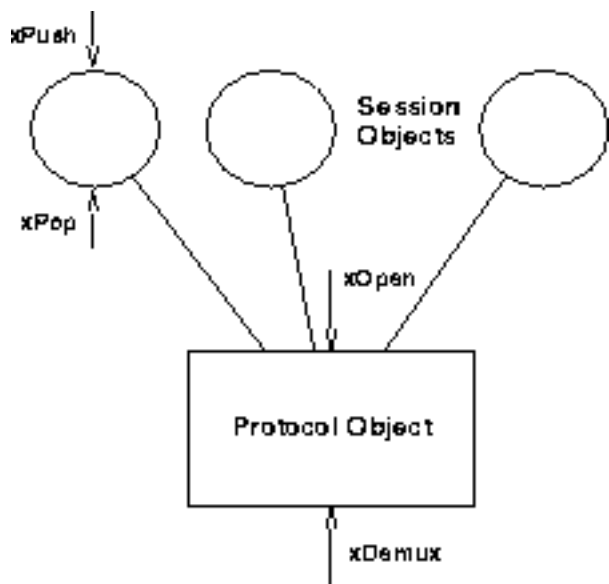


Figure: Using xDemux and xPop to pass a message up a stack.

In addition to sessions and protocols, this discussion has introduced two other *x*-kernel object classes: messages and participants. Both classes represent exactly what you think they do---the messages that protocols send to each other (corresponding to type definition `Msg`), and the addresses of participants that are communicating over some channel (corresponding to type definition `Part`). Message objects are discussed in more detail in Section [4.1](#). Participants are simple enough that we do not describe them further. (See Section [4.2](#) for an example of how participants are used.)

Process Models for Protocols

As we have said, protocol implementors typically have to be concerned about a lot of operating system issues. This subsection introduces one of the most important of these issues---the process model.

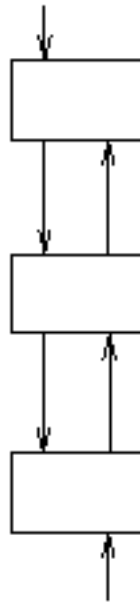
Most operating systems provide an abstraction called a *process*, or alternatively, a *thread*. Each process runs largely independently of other processes, and the OS is responsible for making sure that resources, such as address space and CPU cycles, are allocated to all the current processes. The process abstraction makes it fairly straightforward to have a lot of things executing concurrently on one machine; for example, each user application might execute in its own process, and various things inside the OS might execute as other processes. When the OS stops one process from executing on the CPU and starts up another one, we call this a *context switch*.

When designing a protocol implementation framework, one of the first questions to answer is: "Where are the processes?" There are essentially two choices, as illustrated in Figure [4.1](#). In the first, which we call the *process-per-protocol* model, each protocol is implemented by a separate process. This implies that as a message moves up or down the protocol stack, it is passed from one process/protocol to another---the process that implements protocol *i* processes the message, then passes it to protocol *i-1*, and so on. How one process/protocol passes a message to the next process/protocol depends on the

support the host OS provides for interprocess communication. Typically, there is a simple mechanism for enqueueing a message with a process. The important point, however, is that a context switch is required at each level of the protocol graph---typically a time-consuming operation.



(a) process-per-protocol



(b) process-per-message

Figure: Alternative Process Models.

The alternative, which we call the *process-per-message* model, treats each protocol as a static piece of code, and associates the processes with the messages. That is, when a message arrives from the network, the OS dispatches a process to be responsible for the message as it moves up the protocol graph. At each level, the procedure that implements that protocol is invoked, which eventually results in the procedure for the next protocol being invoked, and so on. For out-bound messages, the application's process invokes the necessary procedure calls until the message is delivered. In both directions, the protocol graph is traversed in a sequence of procedure calls.

Although the process-per-protocol model is sometimes easier to think about---I implement my protocol in my process and you implement your protocol in your process---the process-per-message model is generally more efficient. This is for a simple reason: a procedure call is an order of magnitude more efficient than a context switch on most computers. The former model requires the expense of a context switch at each level, while the latter model costs only a procedure call per level.

The *x*-kernel uses the process-per-message model. Tying this model back to the operations outlined above, this means that once a session (channel) is open at each level, a message can be sent down the protocol stack by a sequence of calls to `xPush`, and up the protocol stack by alternating calls to `xDemux` and `xPop`. This asymmetry--- `xPush` going down and `xDemux`/ `xPop` going up---is unappealing, but necessary. This is because when sending a message out, each layer knows which low-

level session to invoke `xPush` on because there is only one choice, while in the incoming case, the `xDemux` routine at each level has to first demultiplex the message to decide which session's `xPop` to call.

Notice that the high-level protocol does not reach down and *receive* a message from the low-level protocol. Instead, the low-level protocol does an *upcall*---a procedure call up the stack---to deliver the message to the high-level protocol. This is because a receive-style operation would imply that the high-level protocol is executing in a process that is waiting for new messages to arrive, which would then result in a costly context switch between the low-level and high-level protocols. By having the low-level protocol deliver the message to the high-level protocol, incoming messages can be processed by a sequence of procedure calls, just as outgoing messages are.

We conclude this discussion of processes by introducing three operations that the *x*-kernel provides for process synchronization:

```
void semInit(Semaphore *s, int count)
```

```
void semSignal(Semaphore *s)
```

```
void semWait(Semaphore *s)
```

These operations implement conventional counting semaphores. Specifically, every invocation of `semSignal` increments semaphore `s` by one, and every invocation of `semWait` decrements `s` by one, with the calling process blocked (suspended) should decrementing `s` cause its value to become less than zero. A process that is blocked during its call to `semWait` will be allowed to resume as soon as enough `semSignal` operations have been performed to raise the value of `s` above zero. Operation `semInit` initializes the value of `s` to `count`.

For those not familiar with how semaphores are used to synchronize processes, the OS-related references given at the end of this chapter give a good introduction. Also, examples of protocols that use semaphores are given in Chapter [10](#) and [11](#).

[Next](#) [Up](#) [Previous](#)

Next: [Messages](#) **Up:** [Protocol Implementation](#) **Previous:** [Object-Based Protocol Implementation](#)

Messages

We now turn our attention from how protocols invoke operations on each other, and consider what goes on inside a particular protocol. One of the most common things that a protocol does is manipulate messages. We have already seen in Chapter [1](#) how protocols add headers to, and strip headers from, messages. Another common way in which protocols manipulate messages is to break a single message into multiple fragments, and later to join these multiple fragments back into a single message. This is necessary because, as we mentioned in Section [1.1](#), most network links allow messages of only a certain size to be transmitted. Thus, a protocol that uses such a link to transmit a large message must first *fragment* the message on the source node, and then *reassemble* the fragments back into the original message on the destination node. We will see several examples of protocols that fragment and reassemble messages in later chapters.

Because manipulating messages is a basic part of all protocols, the **x**-kernel defines an abstract data type---called the *message* and given by the C type definition `MSG`---that includes an interface for performing these common operations. This section begins by presenting the **x**-kernel's message abstraction. It then outlines the underlying data structures and algorithms that implement the various operations defined by the interface.

Manipulating Messages

Implementation

Events

Another common activity for protocols is to schedule an event to happen some time in the future. To understand how a protocol might use such events, consider the situation where a network sometimes fails to deliver a message to the destination. A protocol that wants to offer a reliable channel across such a network might, after sending a message, schedule an event that is to occur after a certain period of time. If the protocol has not received confirmation that the message was delivered by the time this event happens, then the protocol retransmits the message. In this case, the event implements a *timeout*; we will see several examples of protocols that use timeouts in later chapters. Note that another use for events is to perform periodic maintenance functions, such as garbage collection.

This section introduces the interface to the *x*-kernel's event manager, which allows protocols to schedule a procedure that is to be called after a period of time. It also describes two common data structures used to implement events.

Scheduling Timeouts

Implementation

Id Map

The final *x*-kernel tool we describe is the id mapper. This tool provides a facility for maintaining a set of bindings between identifiers. The id mapper supports operations for adding new bindings to the set, removing bindings from the set, and mapping one identifier into another, relative to a set of bindings. Protocol implementations use these operations to translate identifiers extracted from message headers (e. g., addresses, demultiplexing keys) into capabilities for (pointers to) *x*-kernel objects such as sessions.

Demultiplexing Packets

Implementation

Example Protocol

This section presents an example protocol---A Simple Protocol (ASP)---that illustrates how the interfaces and support routines described in previous sections are used. ASP is a complete, working protocol that is typically configured on top of IP in an *x*-kernel protocol graph. Because of its simplicity, ASP does not implement any of the interesting algorithms found in the protocols described throughout this book; it only serves to illustrate the boilerplate code that is common to all protocols.

Protocol Semantics

Header Files

Initialization

Opening Connections

Demultiplexing Incoming Messages

Sending and Receiving Messages

Summary

This chapter introduces the *x*-kernel protocol implementation framework. The *x*-kernel provides two essential things. First, it defines an interface that protocols use to invoke each other's services. This interface specifies how to open and close connections, and to send and receive messages. The advantage of having such a protocol-to-protocol interface is that it makes it possible to plug-and-play with different protocol objects. Thus, by configuring a graph of protocol objects that adhere to this interface, one is able to build the desired network subsystem.

Second, the *x*-kernel provides a set of support routines that protocol implementations call to perform the tasks that they must perform; for example, manipulate messages, schedule events, and map identifiers. The key observation is that protocols look alike in many respects. The *x*-kernel simply codifies these common features and makes them available to protocol implementors in the form of a set of support routines. One advantage of such a framework is that it keeps protocol implementors from re-inventing the wheel: a single event manager can be used by all protocols. A second advantage is that a rising tide raises all ships: an improvement made to the algorithm or data structure used by any support routine benefits all protocols that use that routine.

Open Issue: Cost of Network Software

In the larger picture, implementing and maintaining network software is a very expensive business. It is estimated that large computer vendors like HP, Sun, SGI, IBM, and Digital spend on the order of \$80-100M per year to maintain their networking code. This is not an investment that necessarily makes these companies any money---it is something they have to do just to stay competitive. The cost is high because each company typically has to maintain several different protocol stacks across several different architecture/OS bases. There is reason to believe that a protocol implementation framework might help to improve this situation, since vendors would be able to maintain just one instance of each protocol---the one that runs in this framework---and then port this framework to all of their platforms. In fact, there is a move in industry to do just this. For example, the System V Streams mechanism is now used in Sun Microsystem's Solaris OS, and the *x*-kernel is included in a recent release of OSF/1.

A related issue is how innovations in protocol design and implementation find their way into commercial offerings. At one time, nearly every vendor had a version of Unix that was derived from the Berkeley version (BSD): HP had HPUNIX, Digital had Ultrix, and Sun had SunOS. This made technology transfer from the research community fairly easy, since most of the networking community was doing their research in BSD Unix. As a result, two years after someone demonstrated a good idea in the BSD implementation of TCP, it would show up in Ultrix or SunOS. Today, however, most of the Unix operating systems are quite far removed from BSD Unix---e.g., Solaris and Digital Unix are essentially based on System V Unix---not to mention the arrival of new operating systems like NT that have virtually no tie to Unix. This means that the technology transfer path is now broken; it will be much harder to move future innovations from the research world into the commercial world.

A second consequence of this shift away from BSD-based operating systems is that the days of all our implementations of TCP being derived from the same code base are over, meaning that maintaining interoperability is going to become more and more challenging. Again, a common protocol implementation platform might help to address this issue.

Copyright 1996, Morgan Kaufmann Publishers

Further Reading

There are two mature environment for implementing network protocols: the Stream mechanism from System V Unix, and the *x*-kernel. The following two papers give an overview of these two systems, and are strongly recommended:

- D. Ritchie. A Stream Input-Output System. *AT&T Bell Laboratories Technical Journal*, 63(8): 311--324, October 1984.
- N. Hutchinson and L. Peterson. The *x*-kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1): 64--76, January 1991.

Probably the most widely used implementation of the TCP/IP protocol stack is that found in Berkeley Unix. This implementation is well documented in [\[LMKQ89\]](#) and [\[SW95\]](#).

More generally, there is a large body of work addressing the issue of structuring and optimizing protocol implementations. Clark was one of the first to discuss the relationship between modular design and protocol performance [\[Cla82\]](#). Later papers then introduce the use of upcalls in structuring protocol code [\[Cla85\]](#), and study the processing overheads in TCP [\[CJRS89\]](#). Finally, [\[WM87\]](#) describes how to gain efficiency in through appropriate design and implementation choices.

Several papers have introduced specific techniques and mechanisms that can be used to improve protocol performance. For example, [\[HMPT89\]](#) describes some of the mechanisms used in the *x*-kernel, [\[MD93\]](#) discusses various implementation of demultiplexing tables, [\[VL87\]](#) introduces the timing wheel mechanism used to manage protocol events, and [\[DP93\]](#) describes an efficient buffer management strategy. Also, the performance of protocols running on parallel processors---locking is a key issue in such environments---is discussed in [\[BG93\]](#) and [\[NYKT94\]](#).

On a more general note, since many aspects of protocol implementation depend on an understanding of the basics of operating systems, we recommend [\[Fin88,BS88,Tan92\]](#) for an introduction to OS concepts.

A good introduction to formal specification languages, and their use in designing and validating

protocols, can be found in [Hol91]. This book also has a nice discussion of exactly what a protocol is. Also, [Gia94] describes an effort to implement a protocol stack in the functional language ML. In contrast to traditional formal specification work which focuses on a single protocol, this effort is concerned with the communication service that results when two or more protocols are composed together.

Finally, we recommend the following live reference:

- <http://www.cs.arizona.edu/xkernel/www>: information on the *x*-kernel, including programmer's manual and source code.

Next	Up	Previous
------	----	----------

Next: [Exercises](#) **Up:** [Protocol Implementation](#) **Previous:** [Open Issue: Cost](#)

Copyright 1996, Morgan Kaufmann Publishers

[Next](#) [Up](#) [Previous](#)

Next: [Direct Link Networks](#) **Up:** [Protocol Implementation](#) **Previous:** [Further Reading](#)

Exercises

Copyright 1996, Morgan Kaufmann Publishers

Direct Link Networks

- [Problem: Physically Connecting Hosts](#)
 - [Hardware Building Blocks](#)
 - [Encoding \(NRZ, NRZI, Manchester, 4B/5B\)](#)
 - [Framing](#)
 - [Error Detection](#)
 - [Reliable Transmission](#)
 - [CSMA/CD \(Ethernet\)](#)
 - [Token Rings \(FDDI\)](#)
 - [Network Adaptors](#)
 - [Summary](#)
 - [Open Issue: Does it belong in hardware?](#)
 - [Further Reading](#)
 - [Exercises](#)
-

Problem: Physically Connecting Hosts

The simplest network possible is one in which all the hosts are directly connected by some physical medium. This may be a wire or a fiber, and it may cover a small area (e.g. an office building) or a wide area (e.g. transcontinental). Connecting two or more nodes with a suitable medium is only the first step, however. There are five additional problems that must be addressed before the nodes can successfully exchange packets.

The first is *encoding* bits onto the wire or fiber so that they can be understood by a receiving host. Second is the matter of delineating the sequence of bits transmitted over the link into complete messages that can be delivered to the end node. This is called the *framing* problem, and the messages delivered to the end hosts are often called *frames*. Third, because frames are sometimes corrupted during transmission, it is necessary to detect these errors and take the appropriate action; this is the *error detection* problem. The fourth issue is making a link appear reliable in spite of the fact that it corrupts frames from time to time. Finally, in those cases where the link is shared by multiple hosts---as opposed to a simple point-to-point link---it is necessary to mediate access to this link. This is the *media access control* problem.

Although these five issues---encoding, framing, error detection, reliable delivery, and access mediation---can be discussed in the abstract, they are very real problems that are addressed in different ways by different networking technologies. This chapter considers these issues in the context of three specific network technologies: point-to-point links, Carrier Sense Multiple Access (CSMA) networks (of which Ethernet is the most famous example), and token rings (of which FDDI is the most famous example). The goal of this Chapter is simultaneously to survey the available network technology, and to explore these five fundamental issues.

Before tackling the specific issues of connecting hosts, this chapter begins by examining the building blocks that will be used: nodes and links. We then explore the first three issues---encoding, framing and error detection---in the context of a simple point-to-point link. The techniques introduced in these three sections are general, and therefore, apply equally well to multiple access networks. The problem of reliable delivery is considered next. Since link-level reliability is usually not implemented in shared access networks, this discussion focuses on point-to-point links only. Finally, we address the media access problem by examining the two main approaches to managing access to a shared link: CSMA and the token ring.

Note that these five functions are, in general, implemented in a network adaptor---a board that plugs into a host's I/O bus on one end, and into the physical medium on the other end. In other words, bits are exchanged between adaptors, but correct frames are exchanged between nodes. This adaptor is controlled by software running on the node---the device driver---which is, in turn, typically represented as the bottom-most protocol in a protocol graph. This chapter concludes with a concrete example of a network adaptor, and sketches the device driver for such an adaptor.

Next	Up	Previous
----------------------	--------------------	--------------------------

Next: [Hardware Building Blocks](#) **Up:** [Direct Link Networks](#) **Previous:** [Direct Link Networks](#)

Hardware Building Blocks

As we saw in Chapter [1](#), networks are constructed from two classes of hardware building blocks: *nodes* and *links*. This statement is just as true for the simplest possible network---one in which a single point-to-point link connects a pair of nodes---as it is for a world-wide internet. This section gives a brief overview of what we mean by nodes and links, and in doing so, defines the underlying technology that we will assume throughout the rest of this book.

Nodes

Links

Encoding (NRZ, NRZI, Manchester, 4B/5B)

The first step in turning nodes and links into usable building blocks is to understand how to connect them in such a way that bits can be transmitted from one node to the other. As mentioned in the preceding section, signals propagate over physical links. The task, therefore, is to encode the binary data that the source node wants to send into the signals that the links are able to carry, and then decode the signal back into the corresponding binary data at the receiving node. We consider this problem in the context of a digital link, in which case we are most likely working with two discrete signals. We generically refer to these as the high signal and the low signal, although in practice these signals would be two different voltages on a copper-based link and two different power levels on an optical link.

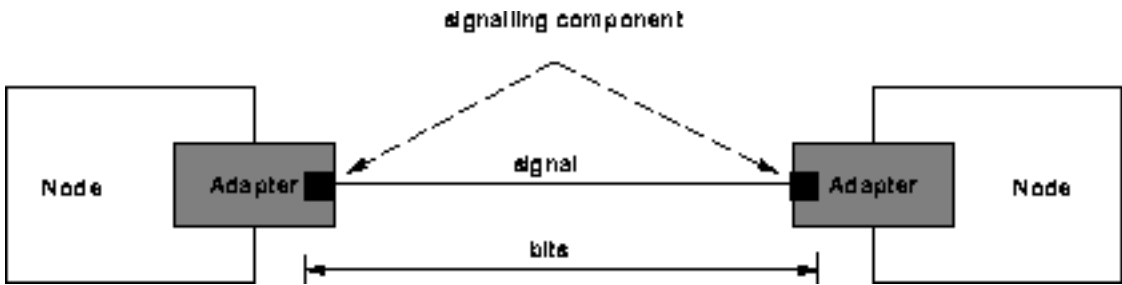





Figure: Signals between signalling components, bits between adaptors.

As we have said, most of the functions discussed in this chapter are performed by a network adaptor---a piece of hardware that connects a node to a link. The network adaptor contains a signalling component that actually encodes bits into signals at the sending node, and decodes signals into bits at the receiving node. Thus, as illustrated in Figure , signals travel over a link between two signalling components, and bits flow between network adaptors.

...text deleted...

Framing

Now that we have seen how to transmit a sequence of bits over a point-to-point link---from adaptor to adaptor---let's consider the scenario illustrated in Figure . Recall from Chapter  that we are focusing on packet-switched networks, which means that blocks of data (called frames at this level), not bit-streams, are exchanged between nodes. It is the network adaptor that enables the nodes to exchange frames. When node A wishes to transmit a frame to node B, it tells its adaptor to transmit a frame from its memory. This results in a sequence of bits being sent over the link. The adaptor on node B then collects together the sequence of bits arriving on the link, and deposits the corresponding frame in B's memory. Recognizing exactly what set of bits constitute a frame---that is, determining where the frame begins and ends---is the central challenge faced by the adaptor.

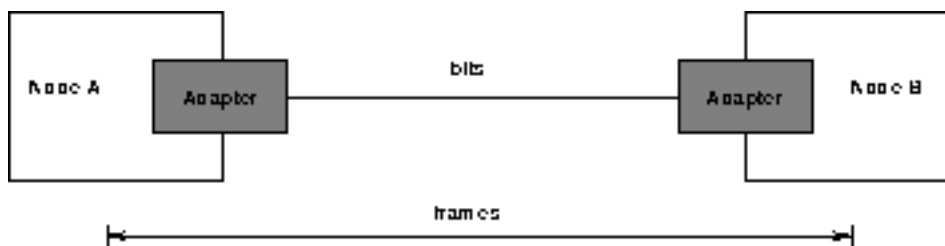


Figure: Bits between adaptors, frames between hosts.

There are several ways to address the framing problem. This section uses several different protocols to illustrate the various points in the design space. Note that while we discuss framing in the context of point-to-point links, the problem is a fundamental one that must also be addressed in multiple access networks like CSMA and token ring.

Byte-Oriented Protocols (BISYNC, IMP-IMP, DDCMP)

Bit-Oriented Protocols (HDLC)

Clock-Based Framing (SONET)

Copyright 1996, Morgan Kaufmann Publishers

Error Detection

As discussed in Chapter [□](#), bit errors are sometimes introduced into frames. This happens, for example, because of electrical interference or thermal noise. Although rare, especially on optical links, some mechanism is needed to detect these errors so that corrective action can be taken. Otherwise, the end user is left wondering why the C program that successfully compiled just a moment ago now suddenly has a syntax error in it, when all that happened in the interim is that it was copied across a network file system.

There is a long history of techniques for dealing with bit errors in computer systems, dating back to *Hamming* and *Reed/Solomon* codes that were developed for use when storing data on magnetic disks and in early core memories. In networking, the dominate method for detecting transmission errors is a technique known as *cyclic redundancy check* (CRC). It is used in nearly all the link-level protocols discussed in the previous section---e.g., IMP-IMP, HDLC, DDCMP---as well as in the CSMA and token ring protocols described later in this chapter. This section outlines the basic CRC algorithm. It also introduces two other approaches to error detection--- *two-dimensional parity* and *checksums*. The former is used by the BISYNC protocol when it is transmitting ASCII characters (CRC is used as the error code when BISYNC is used to transmit EBCDIC), and the latter is used by several Internet protocols.

Cyclic Redundancy Check

Two-Dimensional Parity

The Internet Checksum Algorithm

Copyright 1996, Morgan Kaufmann Publishers

Reliable Transmission

As we saw in the previous section, frames are sometimes corrupted while in transit, with an error code like CRC used to detect such errors. While some error codes are strong enough also to correct errors, in practice, the state-of-the-art in error correcting codes is not advanced enough to handle the range of bit and burst errors that can be introduced on a network link, and as a consequence, corrupt frames must be discarded. A link-level protocol that wants to deliver frames reliably must somehow recover from these discarded (lost) frames.

This is usually accomplished using a combination of two fundamental mechanisms--- *acknowledgments* and *timeouts*. An acknowledgment (ACK for short) is a small control frame that a protocol sends back to its peer saying that it has received an earlier frame. By control frame we mean a header without any data, although a protocol can *piggyback* an ACK on a data frame it just happens to be sending in the opposite direction. The receipt of an acknowledgment indicates to the sender of the original frame that its frame was successfully delivered. If the sender does not receive an acknowledgment after a reasonable amount of time, then it *retransmits* the original frame. This action of ``waiting a reasonable amount of time" is called a *timeout*.

The general strategy of using acknowledges and timeouts to implement reliable delivery is sometimes called Automatic Repeat reQuest, normally abbreviated ARQ. This section describes three different ARQ algorithms using generic language; i.e., we do not give detailed information about a particular protocol's header fields.

Stop-and-Wait

The simplest ARQ scheme is the *stop-and-wait* algorithm. The idea of stop-and-wait is straightforward: after transmitting one frame, the sender waits for an acknowledgment before transmitting the next frame. If the acknowledgment does not arrive after a certain period of time, the sender times out and retransmits the original frame.

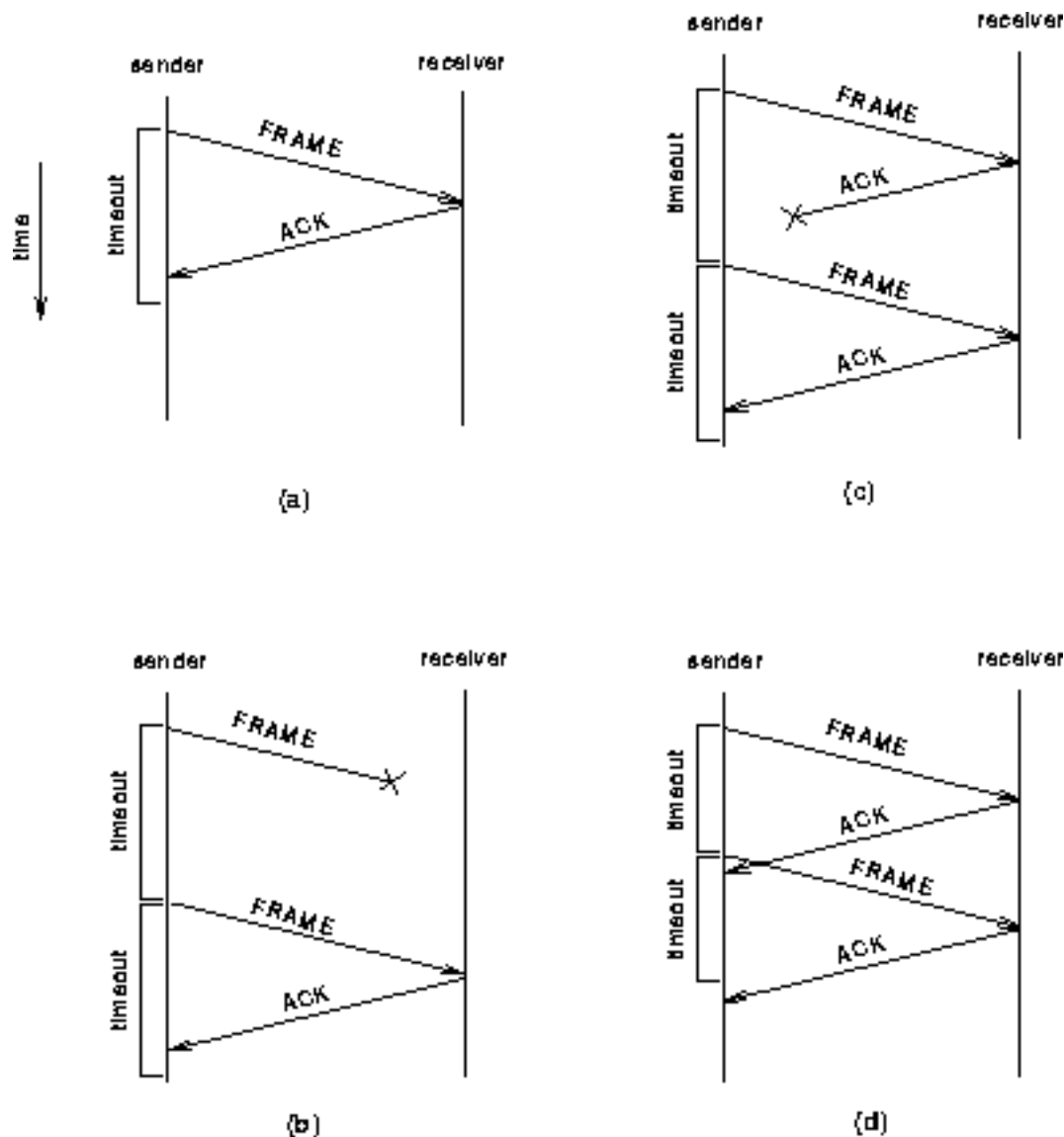


Figure: Time-line showing four different scenarios for stop-and-wait algorithm.

Figure illustrates four different scenarios that result from this basic algorithm. This figure is a time line, a common way to depict a protocol's behavior. The sending side is represented on the left, the receiving side is depicted on the right, and time flows from top to bottom. Figure (a) shows the situation where the ACK is received before the timer expires, (b) and (c) show the situation where the original frame and the ACK, respectively, are lost, and (d) shows the situation where the timeout fires too soon. Recall that by ``lost" we mean that the frame was corrupted while in transit, this corruption was detected by an error code on the receiver, and the frame was discarded.

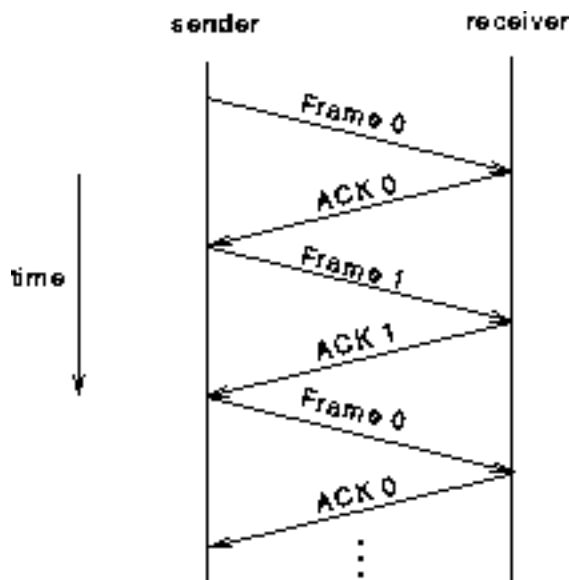





Figure: Time-line for stop-and-wait with 1-bit sequence number.

There is one important subtlety in the stop-and-wait algorithm. Suppose the sender sends a frame and the receiver acknowledges it, but the acknowledgment is either lost or delayed in arriving. This situation is illustrated in time-lines (c) and (d) of Figure . In both cases, the sender times out and retransmits the original frame, but the receiver will think that it is the next frame, since it correctly received and acknowledged the first frame. This has the potential to cause duplicate copies of a frame to be delivered. To address this problem, the header for a stop-and-wait protocol usually includes a 1-bit sequence number---i.e., the sequence number can take on the values 0 and 1---and the sequence numbers used for each frame alternate, as illustrated in Figure . Thus, when the sender retransmits frame 0, the receiver can determine that it is seeing a second copy of 0 rather than the first copy of frame 1, and therefore, ignore it (the receiver still acknowledges it).

The main shortcoming of the stop-and-wait algorithm is that it allows the sender to have only one outstanding frame on the link at a time, and this may be far below the link's capacity. Consider, for example, a 1.5Mbps link with a 45ms round-trip time (RTT). This link has a delay \times bandwidth product of 67.5Kb, or approximately 8KB. Since the sender can send only one frame per RTT, and assuming a frame size of 1KB, that implies a maximum sending rate of $1024 \times 8 / 0.045 = 182$ Kbps, or about one-eighth of the link's capacity. To use the link fully, then, we'd like the sender to be able to transmit up to 8 frames before having to wait for an acknowledgment.

The significance of the bandwidth \times delay product is that it represents the amount of data that could be in transit. We would like to be able to send this much data without waiting for the first acknowledgment. The principle at work here is often referred to as keeping the pipe full. The algorithms presented in the following two subsections do exactly this.

Sliding Window

Consider again to the scenario where the link has a delay \times bandwidth product of 8KB and frames are 1KB big. We would like for the sender to be ready to transmit the ninth frame at pretty much the same moment that the ACK for the first frame arrives. The algorithm that allows us to do this is called *sliding window*, and an illustrative time-line is given in Figure .

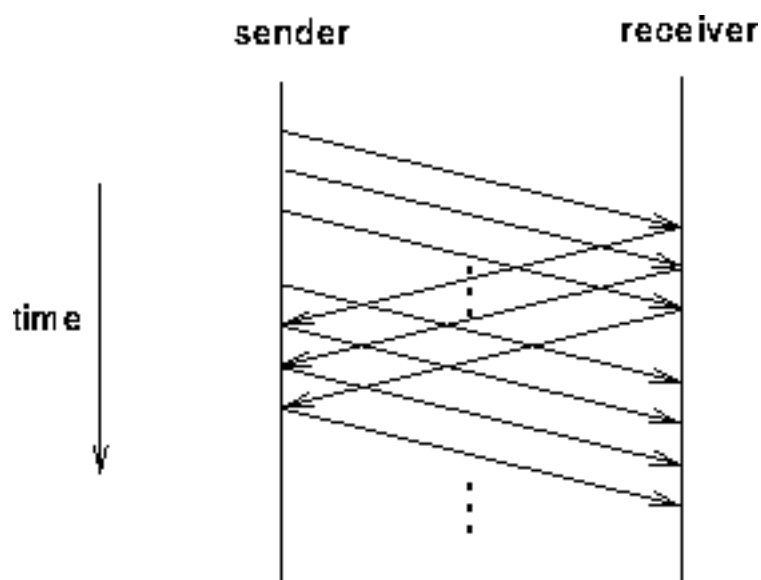


Figure: Time-line for Sliding Window algorithm.

Algorithm

The sliding window algorithm works as follows. First, the sender assigns a *sequence number*, denoted SeqNum, to each frame. For now, let's ignore the fact that SeqNum is implemented by a finite-sized header field, and instead assume that it can grow infinitely large. The sender maintains three variables: the *send window size*, denoted SWS, gives the upper bound on the number of outstanding (unacknowledged) frames that the sender can transmit; LAR denotes the sequence number of the *last acknowledgment received*; and LFS denotes the sequence number of the *last frame sent*. The sender also maintains the following invariant:

$$\text{LFS} - \text{LAR} + 1 \leq \text{SWS}.$$

This situation is illustrated in Figure .

When an acknowledgment arrives, the sender moves LAR to the right thereby allowing the sender to transmit another frame. Also, the sender associates a timer with each frame it transmits, and retransmits the frame should the timer expire before an ACK is received. Notice that the sender has to be willing to buffer up to SWS frames since it must be prepared to retransmit them until they are acknowledged.

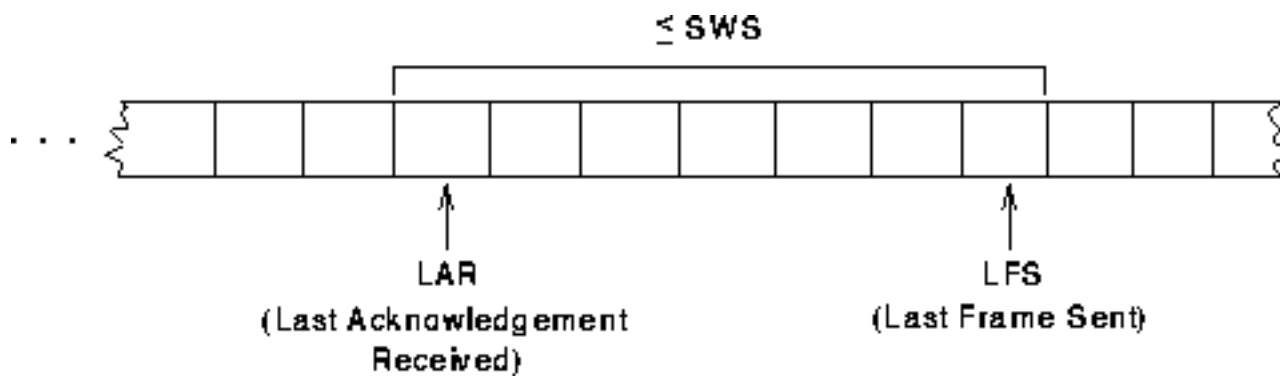


Figure: Sliding Window on sender.

The receiver maintains the following three variables: the *receive window size*, denoted RWS, gives the upper bound on the number of out-of-order frames that the receiver is willing to accept; LFA denotes the sequence number of the *last frame acceptable*; and NFE denotes the sequence number of the *next frame expected*. The receiver also maintains the following invariant:

$$LFA - NFE + 1 \leq RWS.$$

This situation is illustrated in Figure .

When a frame with sequence number SeqNum arrives, the receiver takes the following action. If $\text{SeqNum} < NFE$ or $\text{SeqNum} > LFA$, then the frame is outside the receiver's window, and it is discarded. If $NFE \leq \text{SeqNum} \leq LFA$, then the frame is within the receiver's window and it is accepted. Now the receiver needs to decide whether or not to send an ACK. Let SeqNumToAck denote the largest sequence number not yet acknowledged, such that all frames with sequence numbers less than SeqNumToAck have been received. The receiver acknowledges the receipt of SeqNumToAck, even if higher numbered packets have been received. This acknowledgment is said to be cumulative. It then sets $NFE = \text{SeqNumToAck} + 1$, and adjusts $LFA = \text{SeqNumToAck} + RWS$.

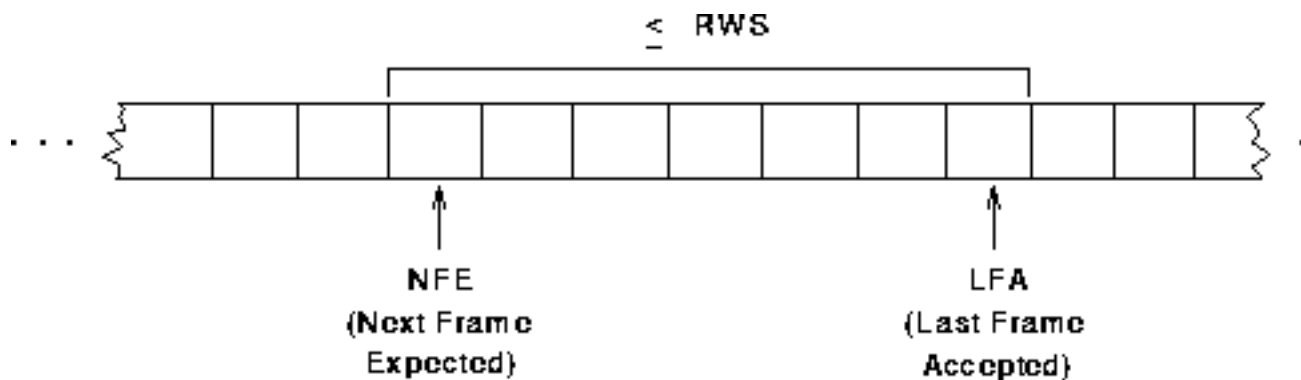


Figure: Sliding Window on receiver.


For example, suppose $NFE=5$ (i.e., the last ACK the receiver sent was for sequence number 4), and $RWS=4$. This implies that $LFA=9$. Should frames 6 and 7 arrive, they will be buffered because they are within the receiver's window. However, no ACK needs to be sent since frame 5 is yet to arrive. Frames 6

and 7 are said to have arrived out-of-order. (Technically, the receiver could resend an ACK for frame 4 when 6 and 7 arrive.) Should frame 5 then arrive---perhaps it is late because it was lost the first time and had to be retransmitted, or perhaps it was simply delayed---the receiver acknowledges frame 7, bumps NFE to 8, and sets LFA to 12. If frame 5 is in fact lost, then a timeout will occur at the sender and frame 5 will be resent. Because the sender has to backtrack by some number of frames (potentially as many as the window size), this scheme is known as *go-back-n*.

We observe that when a timeout occurs, the amount of data in transit decreases, since the sender is unable to advance its window until frame 5 is acknowledged. This means that when packet losses occur, this scheme is no longer keeping the pipe full. The longer it takes to notice that a packet loss has occurred, the more severe this problem becomes.

Notice that in this example, the receiver could have sent a *negative acknowledgment* (NAK) for frame 5 as soon as frame 6 arrived. However, this is unnecessary as the sender's timeout mechanism is sufficient to catch this situation, and sending NAKs adds additional complexity to the receiver. Also, as we mentioned, it would have been legitimate to send additional acknowledgments of frame 4 when frames 6 and 7 arrived; in some cases, a sender can use duplicate ACKs as a clue that a frame was lost. Both approaches help to improve performance by allowing early detection of packet losses.

Yet another variation on this scheme would be to use *selective acknowledgments*. That is, the receiver could acknowledge exactly those frames which it has received, rather than just the highest numbered frame received in order. So, in the above example, the receiver could acknowledge the receipt of frames 6 and 7. Giving more information to the sender makes it potentially easier for the sender to keep the pipe full, but adds complexity to the implementation.

The sending window size is selected according to how many frames we want to have outstanding on the link at a give time; SWS is easy to compute for a given delay \times bandwidth product.  On the other hand, the receiver can set RWS to whatever it wants. Two common settings are RWS=1, which implies that the receiver will not buffer any frames that arrive out-of-order, and RWS=SWS, which implies that the receiver can buffer any of the frames the sender transmits.

Finite Sequence Numbers

We now have to return to the one simplification we introduced into the algorithm---our assumption that sequence numbers can grow infinitely large. In practice, of course, a frame's sequence number is specified in a header field of some finite size. For example, a 3-bit field means that there are 8 possible sequence numbers, 0... 7. This makes it necessary to reuse sequence numbers, or stated another way, sequence numbers wrap around. This introduces the problem of being able to distinguish between different incarnations of the same sequence numbers, which implies that the number of possible sequence numbers must be larger than the number of outstanding frames allowed. For example, stop-and-wait allowed one outstanding frame at a time, and had two distinct sequence numbers.

Suppose we have one more number in our space of sequence numbers than we have potentially outstanding frames; i.e., $SWS \leq \text{MaxSeqNum} - 1$, where MaxSeqNum is the number of available sequence numbers. Is this sufficient? The answer is no. To see this, consider the situation where we have the eight sequence numbers 0 through 7, and $SWS = RWS = 7$. Suppose the sender transmits frames 0..6, they are successfully received, but the ACKs are lost. The receiver is now expecting frames 7,0..5, but the sender times out and sends frames 0..6. Unfortunately, the receiver is expecting the second incarnation of frames 0..5, but gets the first incarnation of these frames. This is exactly the situation we wanted to avoid.

It turns out that the sending window size can be no more than half as big as the number of available sequence numbers, or stated more precisely,

$$SWS < (\text{MaxSeqNum} + 1) / 2.$$

Intuitively, what this is saying is that the sliding window protocol alternates between the two halves of the sequence number space, just as stop-and-wait alternates between sequence numbers 0 and 1. The only difference is that it continually slides between the two halves rather than discretely alternating between them.

Implementation

The following shows the *x*-kernel routines that implement the sending and receiving sides of the sliding window algorithm. The routines are taken from a working *x*-kernel protocol named, appropriately enough, Sliding Window Protocol (SWP).

We start by defining a pair of data structures. First, the frame header is very simple: it contains a sequence number (`SeqNum`) and an acknowledgment number (`AckNum`). It also contains a `Flags` field that indicates whether the frame is an ACK or carries data.

```
typedef u_char   SwpSeqno;

typedef struct {
    SwpSeqno      SeqNum;           /* sequence number of this frame */
    SwpSeqno      AckNum;           /* ack of received frame */
    u_char        Flags;           /* up to 8 bits worth of flags */
} SwpHdr;
```

Next, the state of the sliding window algorithm is defined by the following structure. For the sending side of the protocol, this state includes variables `LAR` and `LFS`, as described earlier in this section, as well as a queue that holds frames that have been transmitted but not yet acknowledged (`txq`). The sending state also includes a *semaphore* called `sendWindowNotFull`. We will see how this is used

below.

For the receiving side of the protocol, the state includes the variable NFE, as described earlier in this section, plus a queue that holds frames that have been received out of order (rxq). Finally, although not shown, the sender and receiver sliding window sizes are defined by constants SWS and RWS, respectively.

```
typedef struct {
    /* sender side state: */
    SwpSeqno    LAR;                /* seqno of last ACK received */
    SwpSeqno    LFS;                /* last frame sent */
    Semaphore    sendWindowNotFull;
    SwpHdr      hdr;                /* pre-initialized header */
    struct txq_slot {
        Event    timeout;          /* event associated with send-timeout */
    } txq[SWS];

    /* receiver side state: */
    SwpSeqno    NFE;                /* seqno of next frame expected */
    struct rxq_slot {
        int      received;         /* is msg valid? */
        Msg      msg;
    } rxq[RWS];
} SState;
```

The sending side of SWP is implemented by procedure swpPush. This routine is rather simple. First, semWait causes this process to block until it is OK to send another frame. Once allowed to proceed, swpPush sets the sequence number in the frame's header, saves a copy of the frame in the transmit queue, sends the frame to the next lower level protocol, and schedules a timeout event to handle the case where the frame is not acknowledged.

The one piece of complexity in this routine is the use of semWait and the sendWindowNotFull semaphore. sendWindowNotFull is a *counting* semaphore that is initialized to the size of the sender's sliding window (SWS). Each time the sender transmits a frame, the semWait operation decrements this count, and blocks the sender should the count go to zero. Each time an ACK is received, the semSignal operation invoked in swpPop (see below) increments this count, thus unblocking any waiting sender.

```
static XkReturn
```

```

swpPush(Sessn s, Msg *msg)
{
    SwpState *state = (SwpState *)s->state;
    struct txq_slot *slot;

    /* wait for send window to open before sending anything */
    semWait(&state->sendWindowNotFull);
    state->hdr.SeqNum = ++state->LFS;
    slot = &state->txq[state->hdr.SeqNum % SWS];
    bcopy(&state->hdr, msgPush(msg, sizeof(SwpHdr)), sizeof(SwpHdr));
    msgConstructCopy(&slot->msg, msg);
    slot->lls = xGetDown(s, 0);
    slot->timeout = evSchedule(swpTimeout, slot, SWP_SEND_TIMEOUT);
    return xPush(slot->lls, msg);
}

```

The receiving side of SWP is implemented in the *x*-kernel routine `swpPop`. This routine actually handles two different kinds of incoming messages: ACKs for frames sent earlier from this node, and data frames arriving at this node. In a sense, the ACK half of this routine is the counterpart to the sender side of the algorithm given in `swpPush`. A decision as to whether the incoming message is an ACK or a data frame is made by checking the `Flags` field in the header. Note that this particular implementation does not support piggybacking ACKs on data frames.

When the incoming frame is an ACK, `swpPop` simply finds the slot in the transmit queue (`txq`) that corresponds to the ACK, and both frees the frame saved there and cancels the timeout event. This work is actually done in a loop since the ACK may be cumulative. The only other thing to notice about this case is the call to subroutine `swpInWindow`. This subroutine, which is given below, ensures that the sequence number for the frame being acknowledged is within the range of ACKs that the sender currently expects to receive.

When the incoming frame contains data, `swpPop` first calls `swpInWindow` to make sure the sequence number of the frame is within the range of sequence numbers that it expects. If it is, it loops over the set of consecutive frames it has received, and passes them up to the higher level protocol by invoking the `xDemux` routine. It also sends a cumulative ACK back to the sender, but does so by looping over the receive queue (it does not use the `SeqNumToAck` variable used in the prose description given earlier in this section).

```

static XkReturn
swpPop(Sessn s, Sessn ds, Msg *dg, void *inHdr)
{
    SwpState *state = (SwpState *)s->state;
    SwpHdr    *hdr = (SwpHdr *)inHdr;

```



```

if (hdr->Flags & FLAG_ACK_VALID)
{
    /* received an acknowledgment---do SENDER-side */
    if (swpInWindow(hdr->AckNum, state->LAR + 1, state->LFS))
    {
        do
        {
            struct txq_slot *slot;

            slot = &state->txq[++state->LAR % SWS];
            evCancel(slot->timeout);
            msgDestroy(&slot->msg);
            semSignal(&state->sendWindowNotFull);
        } while (state->LAR != hdr->AckNum);
    }
}
if (hdr->Flags & FLAG_HAS_DATA)
{
    struct rxq_slot *slot;

    /* received data packet---do RECEIVER side */
    slot = &state->rxq[hdr->SeqNum % RWS];
    if (!swpInWindow(hdr->SeqNum, state->NFE, state->NFE + RWS -
1))
    {
        /* drop the message */
        return XK_SUCCESS;
    }
    msgConstructCopy(&slot->msg, dg);
    slot->received = TRUE;
    if (hdr->SeqNum == state->NFE)
    {
        SwpHdr ackhdr;
        Msg m;

        while (slot->received)
        {
            xDemux(xGetUp(s), s, &slot->msg);
            msgDestroy(&slot->msg);
            slot->received = FALSE;
            slot = &state->rxq[++state->NFE % RWS];
        }
        /* send ACK: */
    }
}

```

```

        ackhdr.AckNum = state->NFE - 1;
        ackhdr.Flags = FLAG_ACK_VALID;
        msgConstructBuffer(&m, (char *)&ackhdr, sizeof(ackhdr));
        xPush(ds, &m);
        msgDestroy(&m);
    }
}
return XK_SUCCESS;
}

```

Finally, `swpInWindow` is a simple subroutine that checks to see if a given sequence number falls between some minimum and maximum sequence number.

```

static bool
swpInWindow(SwpSeqno seqno, SwpSeqno min, SwpSeqno max)
{
    SwpSeqno pos, maxpos;

    pos      = seqno - min;          /* pos *should* be in range [0..MAX)
*/
    maxpos = max - min + 1;          /* maxpos is in range [0..MAX] */
    return pos < maxpos;
}

```

Frame Order and Flow Control

The sliding window protocol is perhaps the best known algorithm in computer networking. What is easily confusing about the algorithm, however, is that it can be used to serve three different roles. The first role is the one we have been concentrating on in this section---to reliably deliver frames across an unreliable link. (In general, the algorithm can be used to reliably deliver messages across an unreliable network.) This is the core function of the algorithm.

The second role that the sliding window algorithm can serve is to preserve the order in which frames are transmitted. This is easy to do at the receiver---since each frame has a sequence number, the receiver just makes sure that it does not pass a frame up to the next higher level protocol until it has already passed up all frames with a smaller sequence number. That is, the receiver buffers (i.e., does not pass along) out-of-order frames. The version of the sliding window algorithm described in this section does preserve frame order, although one could imagine a variation where the receiver passes frames to the next protocol without waiting for all earlier frames to be delivered. A question we should ask ourselves is if we really need for the sliding window protocol to keep the frames in order, or if instead, this is unnecessary

functionality at the link level. Unfortunately, we have not seen yet enough of the network architecture to answer this question---we need to first understand how a sequence of point-to-point links are connected by switches to form an end-to-end path.

The third role that the sliding window algorithm sometimes plays is to support *flow control*---a feedback mechanism by which the receiver is able to throttle the sender. Such a mechanism is used to keep the sender from overrunning the receiver, that is, transmitting more data than the receiver is able to process. This is usually accomplished by augmenting the sliding window protocol so that the receiver not only acknowledges frames it has received, but it also informs the sender of how many frames it has room to receive. The number of frames that the receiver is capable of receiving corresponds to how much free buffer space it has. As in the case of ordered delivery, we need to make sure that flow control is necessary at the link level before incorporating it into the sliding window protocol.

One important thing to take away from this discussion is the system design principle we call separation of concerns. That is, one must be careful to distinguish between different functions that are sometimes rolled together in one mechanism, and to make sure that each function is necessary and being supported in the most effective way. In this particular case, reliable delivery, ordered delivery, and flow control are sometimes combined in a single sliding window protocol, and we should ask ourselves if this is the right thing to do at the link level. With this question in mind, we revisit the sliding window algorithm in Chapter [10](#) (we show how X.25 networks use it to implement hop-by-hop flow control), and in Chapter [11](#) (we describe how TCP uses it to implement a reliable byte-stream channel).

Concurrent Logical Channels

The IMP-IMP protocol used in the ARPANET provides an interesting alternative to the sliding window protocol, in that it is able to keep the pipe full while still using the simple stop-and-wait algorithm. One important consequence of this approach is that it does not keep the frames sent over a given link in any particular order. Also, it implies nothing about flow control.

The idea underlying the IMP-IMP protocol, which we refer to as *concurrent logical channels*, is to multiplex several logical channels onto a single point-to-point link, and to run the stop-and-wait algorithm on each of these logical channels. There is no relationship maintained among the frames sent on any of the logical channels, yet because a different frame can be outstanding on each of the several logical channels, the sender can keep the link full.

More precisely, the sender keeps three bits of state for each channel: a boolean saying whether the channel is currently busy, the 1-bit sequence number to use the next time a frame is sent on this logical channel, and the next sequence number to expect on a frame that arrives on this channel. When the node has a frame to send, it uses the lowest idle channel, and otherwise behaves just like stop-and-wait.

In practice, the ARPANET supported eight logical channels over each ground link, and 16 over each satellite link. In the ground-link case, the header for each frame included a 3-bit channel number and a 1-bit sequence number, for a total of 4 bits. This is exactly the number of bits the sliding window protocol requires to support up to eight outstanding frames on the link.

Next	Up	Previous
----------------------	--------------------	--------------------------

Next: [CSMA/CD \(Ethernet\)](#) **Up:** [Direct Link Networks](#) **Previous:** [Error Detection](#)

Copyright 1996, Morgan Kaufmann Publishers

[Next](#)[Up](#)[Previous](#)

Next: [Token Rings \(FDDI\)](#) **Up:** [Direct Link Networks](#) **Previous:** [Reliable Transmission](#)

CSMA/CD (Ethernet)

Easily the most successful local area networking technology of the last 20 years is the Ethernet. Developed in the mid-1970's by researchers at the Xerox Palo Alto Research Center (PARC), the Ethernet is a working example of the more general *Carrier Sense, Multiple Access with Collision Detect* (CSMA/CD) local-area network technology. The Ethernet supports a transmission rate of 10Mbps. This section focuses on the most common or 'classical' form of Ethernet. While the basic algorithms have not changed, Ethernet has adapted over the years to run at a variety of speeds and over a range of physical media.

As indicated by the CSMA name, the Ethernet is a "multiple access" network; a set of nodes send and receive frames over a shared link. One can, therefore, think of an Ethernet as being like a bus that has multiple stations plugged into it. The "carrier sense" in CSMA/CD means that all the nodes can distinguish between an idle and a busy link, and "collision detect" means that a node listens as it transmits, and can therefore detect when a frame it is transmitting has interfered (collided) with a frame transmitted by another node.

The Ethernet has its roots in an early packet-radio network, called Aloha, developed at the University of Hawaii to support computer communication across the Hawaiian islands. Like the Aloha network, the fundamental problem faced by the Ethernet is how to mediate access to a shared medium fairly and efficiently (in Aloha the medium was the atmosphere, while in Ethernet the medium is a coax cable). That is, the core idea in both Aloha and the Ethernet is an algorithm that controls when each node can transmit.

Physical Properties

Protocol

Experience


[Next](#) [Up](#) [Previous](#)

Next: [Token Rings \(FDDI\)](#) **Up:** [Direct Link Networks](#) **Previous:** [Reliable Transmission](#)

Copyright 1996, Morgan Kaufmann Publishers

Token Rings (FDDI)

Alongside the Ethernet, token rings are the other significant class of shared media network. There are more different types of token rings than there are types of Ethernet; this section will discuss only one type in detail---FDDI (Fiber Distributed Data Interface). Earlier token ring networks include the 10Mbps and 80 Mbps PRONET ring, IBM's 4Mbps token ring, and the 16Mbps IEEE 802.5/token ring. In many respects, the ideas embodied in FDDI are a superset of the ideas found in any other token ring network. This makes FDDI a good example for studying token rings in general.

As the name suggests, a token ring network like FDDI consists of a set of nodes connected in a ring. (See Figure ) Data always flows in a particular direction on the ring, with each node receiving frames from its up-stream neighbor, and then forwarding them to its down-stream neighbor. This ring-based topology is in contrast to the Ethernet's bus topology. Like the Ethernet, however, the ring is viewed as a single shared medium; it does not behave as a collection of independent point-to-point links that just happen to be configured in a loop. Thus, a token ring shares two key features with an Ethernet. First, it involves a distributed algorithm that controls when each node is allowed to transmit. Second, all nodes see all frames, with the node identified in the frame header as the destination saving a copy of the frame as it flows past.

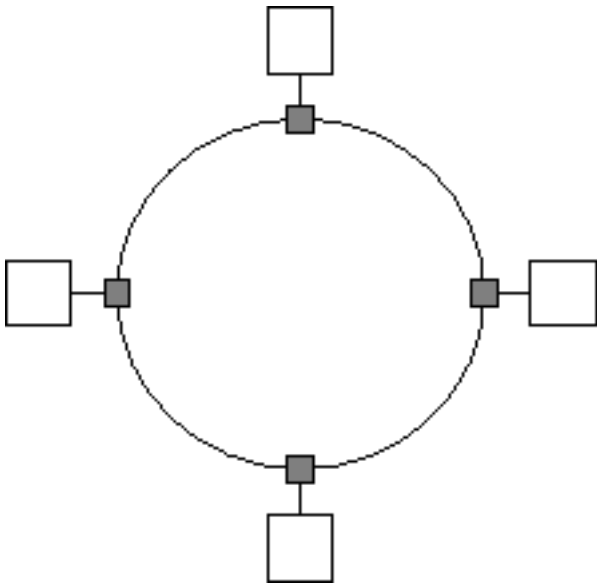


Figure: Ring Network

The term ``token" in token ring comes from the way access to the shared ring is managed. The idea is that a token, which is really just a special sequence of bits (e.g., 01111110) circulates around the ring; each node receives and then forwards the token. When a node that has a frame to transmit sees the token, it takes the token off the ring (i.e., does not forward the special bit pattern) and instead inserts its frame into the ring. Each node along the way simply forwards the frame, with the destination node saving a copy. (The destination also forwards the message onto the next node on the ring.) When the frame makes its all the way back around to the sender, this node strips its frame off the ring (rather than continuing to forward it) and re-inserts the token. In this way, some node down-stream will have the opportunity to transmit a frame. The media access algorithm is fair in the sense that as the token circulates around the ring, each node gets a chance to transmit. Nodes are serviced in a round-robin fashion.

Physical Properties

Timed Token Algorithm

Token Maintenance

Frame Format

[Next](#) [Up](#) [Previous](#)

Next: [Network Adaptors](#) **Up:** [Direct Link Networks](#) **Previous:** [CSMA/CD \(Ethernet\)](#)

Copyright 1996, Morgan Kaufmann Publishers

Network Adaptors

Nearly all the networking functionality described in this chapter is implemented in the network adaptor: framing, error detection, and the media access protocol. The only exception is the point-to-point ARQ schemes described in Section [3.1](#), which are typically implemented in the lowest-level protocol running on the host. We conclude this chapter by sketching the design of a generic network adaptor, and the device driver software that controls it.

When reading this section, keep in mind that no two network adaptors are exactly alike; they vary in countless small details. Our focus, therefore, is on their general characteristics, although we do include some examples from an actual adaptor to make the discussion more tangible.

Components

View From the Host

Example Device Driver

Summary

This chapter introduces the hardware building blocks of a computer network---nodes and links---and discusses the five key problems that must be solved so that two or more nodes that are directly connected by a physical link can exchange messages with each other.

First, physical links carry signals. It is therefore necessary to encode the bits that make up a binary message into the signal at the source node, and then recover the bits from the signal at the receiving node. This is the encoding problem, and it is made challenging by the need to keep the sender and receiver's clocks synchronized. We discussed four different encoding techniques---NRZ, NRZI, Manchester, and 4B/5B---which differ largely in how they encode clock information along with the data being transmitted. One of the key attributes of an encoding scheme is its efficiency, that is, the ratio of signal pulses to encoded bits.

Once it is possible to transmit bits between nodes, the next step is to figure out how to package these bits into frames. This is the framing problem, and it boils down to being able to recognize the beginning and end of each frame. Again, we looked at several different techniques, including byte-oriented protocols, bit-oriented protocols, and clock-based protocols.

Assuming each node is able to recognize the collection of bits that make up a frame, the third issue is to determine if those bits are in fact correct, or if they have possibly been corrupted in transit. This is the error detection problem, and we looked at three different approaches: cyclic redundancy codes, two-dimensional parity, and checksums. Of these, the CRC approach gives the strongest guarantees and is the most widely used at the link level.

Given that some frames will arrive at the destination node containing errors, and thus have to be discarded, the next problem is how to recover from such losses. The goal is to make the link appear reliable. The general approach to this problem is called ARQ, and it involves using a combination of acknowledgments and timeouts. We looked at three specific ARQ algorithms: stop-and-wait, sliding window, and concurrent channels. What makes these algorithms interesting is how effectively they use the link, with the goal keeping the pipe full.

The final problem is not relevant to point-to-point links, but it is the central issue in multiple access links. The problem is how to mediate access to a shared link so that all nodes eventually have a chance

to transmit their data. In this case, we looked at two different media access protocols---CSMA/CD and token ring---which have been put to practical use in the Ethernet and FDDI local area networks, respectively. What both of these technologies have in common is that control over the network is distributed over all the nodes connected to the network; there is no dependency on a central arbitrator.

We concluded the chapter by observing that in practice, most of the algorithms that address these five problems are implemented on the adaptor that connects the host to the link. It turns out that the design of this adaptor is of critical importance in how well the network, as a whole, performs.

Next	Up	Previous
----------------------	--------------------	--------------------------

Next: [Open Issue: Does](#) **Up:** [Direct Link Networks](#) **Previous:** [Network Adaptors](#)

Open Issue: Does it belong in hardware?

One of the most important questions in the design of any computer system is: "what belongs in hardware and what belongs in software". In the case of networking, the network adaptor finds itself at the heart of this question. For example, why is the Ethernet algorithm presented in Section [10.1](#) typically implemented on the network adaptor, while the higher level protocols discussed later in this book are not?

It is certainly possible to put a general-purpose microprocessor on the network adaptor, which gives you the opportunity to move high-level protocols like TCP/IP there. The reason this is typically not done is complicated, but it comes down to the economics of computer design: the host processor is usually the fastest processor on a computer, and it would be a shame if this fast host processor had to wait for a slower adaptor processor to run TCP/IP when it could have done the job faster itself. On the flip side, some protocol processing does belong on the adaptor. The general rule of thumb is any processing for which a fixed processor can keep pace with the link speed---i.e., a faster processor would not improve the situation---is a good candidate for being moved to the adaptor. In other words, any function that is already limited by the link speed, as opposed to the processor at the end of the link, might be effectively implemented on the adaptor.

In general, making the call as to what functionality belongs on the network adaptor and what belongs on the host computer is a difficult one, and one that is re-examined each time someone designs a new network adaptor.

Independent of exactly what protocols are implemented on the adaptor, it is generally the case that the data will eventually find its way onto the main computer, and when it does, the efficiency with which the data is moved between the adaptor and the computer's memory is very important. Recall from Section [10.1](#) that memory bandwidth---the rate at which data can be moved from one memory location to another---is typically the limiting factor in how a workstation-class machine performs. An inefficient host/adaptor data transfer mechanism can, therefore, limit the throughput rate seen by application programs running on the host. First, there is the issue of whether DMA or programmed I/O is used; each has advantages in different situations. Second, there is the issue of how well the adaptor is integrated with the operating system's buffer mechanism; a carefully integrated system is usually able to avoid copying data at higher level of the protocol graph, thereby improving application-to-application

throughput.

[Next](#) [Up](#) [Previous](#)

Next: [Further Reading](#) **Up:** [Direct Link Networks](#) **Previous:** [Summary](#)

Copyright 1996, Morgan Kaufmann Publishers

Further Reading

One of the most important contributions in computer networking over the last 20 years is the original paper by Metcalf and Boggs introducing the Ethernet. Many years later, Boggs (along with Mogul and Kent) reported their practical experiences with Ethernet, debunking many of the myths that had found their way into the literature over the years. Both papers are must reading, and so make this Chapter's recommended reading list. The third paper on our reading list describes FDDI, with a particular emphasis on the standardization process.

- R. Metcalf and D. Boggs. Ethernet: Distributed Packet Switching for Local Computer Networks. *Communications of the ACM*, 19(7): 395--403, July 1976.
- D. Boggs, J. Mogul, and C. Kent. Measured Capacity of an Ethernet. *Proceedings of the SIGCOMM '88 Symposium*, pages 222--234, August 1988.
- F. Ross and J. Hamstra. Forging FDDI. *IEEE Journal of Selected Areas in Communication*, Vol. 11, 181--190, February 1993.

There are countless text books with a heavy emphasis on the lower levels of the network hierarchy, with a particular focus on *telecommunications*---networking from the phone company's perspective. Books by Spragins [[SHP91](#)] and Minoli [[Min93](#)] are two good examples. Several other books concentrate on various local area network technologies. Of these, Stallings' book is the most comprehensive [[Sta90](#)], while Jain gives a thorough description of FDDI [[Jai94](#)]. Jain's book also gives a good introduction to the low-level details of optical communication. Also, a comprehensive overview of FDDI can be found in [[Ros86](#)].

For an introduction to information theory, Blahut's book is a good place to start [[Bla87](#)], with Shannon's seminal paper on link capacity found in [[Sha48](#)].

For a general introduction to the mathematics behind error codes, [[RF89](#)] is recommended. For a detailed discussion of the mathematics, and a description of how the hardware that is generally used for CRC calculation in particular, see [[PB61](#)].

On the topic of network adaptor design, much work has been done recently as researchers try to connect hosts to networks running at higher and higher rates. For example, see [[DPD94](#),[Dav91](#),[TS93](#),[Ram93](#),[EWL94](#),[Met93](#),[KC88](#),[CFFD93](#),[Ste94a](#)].

For general information on computer architecture, Hennessy and Patterson's book [[PH90](#)] is an excellent reference, while [[Sit92](#)] explains the DEC Alpha architecture in detail. (Many of the performance experiments discussed in this book were run on the DEC Alpha.)

Finally, we recommend the following live reference:

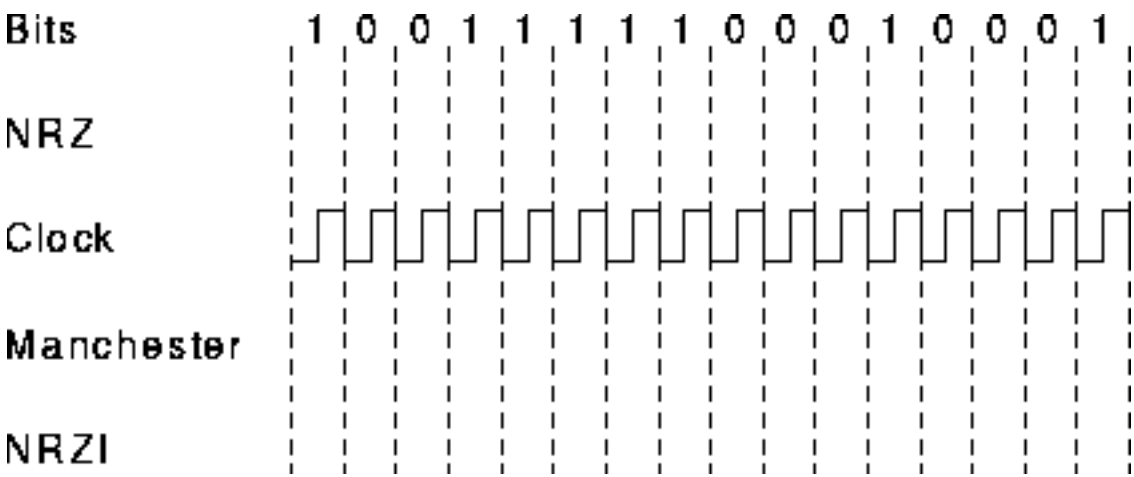
- <http://stdsbbs.ieee.org/>: status of various IEEE network-related standards.

Next	Up	Previous
----------------------	--------------------	--------------------------

Next: [Exercises](#) **Up:** [Direct Link Networks](#) **Previous:** [Open Issue: Does](#)

Exercises

- Many different mediums can be used to transmit information between network nodes, in addition to those mentioned in this Chapter. For example, communication may be done via laser beam, microwave beam, radio waves, sound waves, and infra-red. Select two or three of these media and investigate the advantages, capabilities, and drawbacks of each. Summarize your studies by indicating the situations and environments where each of the media you have studied would be most appropriately used.
- Show the NRZ, Manchester, and NRZI encodings for the bit pattern shown below. Assume that the NRZI signal starts out low.



- Show the 4B/5B encoding, and the resulting NRZI signal, for the bit sequence: 1110 0101 0000 0011.
- Assuming a framing protocol that uses bit stuffing, show the bit sequence transmitted over the link when the frame contains the following bit sequence:


110101111101011111101011111110.

Mark the stuffed bits.

5. Suppose the following sequence of bits arrived over a link:

1101011111010111110010111110110.

Show the the resulting frame after any stuffed bits have been removed. Indicate any errors that might have been introduced into the frame.

6. Suppose you want to send some data using the BISYC framing protocol, and the last two bytes of your data are DLE and ETX. What sequence of bytes would be transmitted immediately prior to the CRC? What would your answer be if the IMP-IMP protocol were used?
7. Suppose we want to transmit the message 11001001 and protect it from errors using the CRC polynomial $x^3 + 1$.
1. Use polynomial long division to determine the message that should be transmitted.
 2. Suppose the leftmost bit of the message is inverted due to noise on the transmission link. What is the result of the receiver's CRC calculation? How does the receiver know that an error occurred?
8. What are the advantages of a CRC over the IP checksum algorithm? What are the disadvantages?
9. Give an example of a 4-bit error that would not be detected by two-dimensional parity, using the data in Figure . What is the general set of circumstances under which 4-bit errors will be undetected?
10. Typically the checksum field is included in the data being checksummed, with this field zeroed-out by the sender when computing the checksum. On the receiver, when the result of computing the checksum---again this sum includes the checksum field---results in an answer of zero, the receiver accepts the message as correct. Explain why this is. Give a small example that illustrates your answer.
11. Consider an ARQ protocol that uses only negative acknowledges (NAK), but no positive acknowledgments (ACK). Describe what timeouts would need to be scheduled. Explain why an ACK-based protocol is usually preferred to a NAK-based protocol.
12. Consider an ARQ algorithm running over a 20Km point-to-point fiber link.
1. Compute the propagation delay for this link, assuming the speed of light is 2×10^8 mps in the fiber.
 2. Suggest a suitable timeout value for the ARQ algorithm to use.
 3. Why might it still be possible for the ARQ algorithm to timeout and retransmit a frame given this timeout value.

13. In the sliding window protocol, why doesn't having RWS greater than SWS make any sense?
14. Suppose you are designing a sliding window protocol for a 1Mbps point-to-point link to the moon, which has a one-way latency of 1.25s. Assuming each frame carries 1KB of data, how many bits do you need for the sequence number.
15. This Chapter suggests that the sliding window protocol can be used to implement flow control. One could imagine doing this by having the receiver delay ACKs, that is, not send the ACK until there is free buffer space to hold the next frame. In doing this, each ACK would simultaneously acknowledge the receipt of the last frame and tell the source that there is now free buffer space available to hold the next frame. Explain why implementing flow control in this way is not a good idea.
16. Implement the concurrent logical channel protocol in the *x*-kernel.
17. Extend the *x*-kernel implementation of SWP to piggyback ACKs on data frames.
18. Extend the *x*-kernel implementation of SWP to support selective ACKs.
19. Experiment with the sender's window size (*SWS*) in the *x*-kernel implementation of SWP. Plot the effective throughput for $SWS=1, \dots, 32$ on whatever network you have available. Running SWP over an Ethernet is fine; you can also vary the link latency by configuring *x*-kernel protocol *VDELAY* into your protocol graph. (See Appendix of the *x-kernel Programmer's Manual* for a description of *VDELAY*.)
20. What company manufactures the Ethernet adaptor for the machine you most commonly use? Determine what address prefix has been assigned to this manufacturer.
21. Why is it important for protocols configured on top of the Ethernet to have a length field in their header, indicating how long the message is?
22. Considering that the maximum length of an Ethernet is 1500m, compute the worst case propagation delay of an Ethernet. Explain why the Ethernet algorithm uses 51.2 μ s instead of the number you just computed.
23. Some network applications are a better match for an Ethernet, and some are a better match for an FDDI network. Which network would be the better match for a remote terminal application (e.g., Telnet) and which would be a better match for a file transfer application (e.g., FTP)? Would your answer be the same if you assume a 100Mbps Ethernet, rather than 10Mbps? Give a general explanation for what it is about each of these applications that suggests one network is a better match than the other.

24. For a 100Mbps ring network that uses delayed release, has a token rotation time of 200 μ s, and allows each station to transmit one 1KB packet each time it possesses the token, determine the network's maximum effective throughput rate.
25. Consider a token ring network like FDDI where a station is allowed to hold the token for some period of time, known as the *token holding time* and denoted THT. Let RingLatency denote the time it takes the token to make one complete rotation around the network in the case where none of the stations have any data to send.
1. Express, in terms of THT and RingLatency, the efficiency of this network when only a single station is active.
 2. What setting of THT would be optimal for a network that had only one station active (with data to send) at a time?
 3. In the case where N stations are active, give an upper bound on the *token rotation time*, denoted TRT, for the network.
26. Consider a token ring with a ring latency of 200 μ s. Assuming the delayed token release strategy is used, what is the effective throughput rate that can be achieved if the ring has a bandwidth of 4Mbps. What is the effective throughput rate that can be achieved if the ring has a bandwidth of 100Mbps.
27. Repeat the previous problem for a ring that uses immediate token release.
28. Give the code fragment that determines if the Lance Ethernet adaptor just issued a receive interrupt.
29. Sketch the implementation of the device-independent half of an *x*-kernel Ethernet driver. Compare your implementation to protocol ETH distributed with the *x*-kernel.

Next	Up	Previous
----------------------	--------------------	--------------------------

Next: [Packet Switching](#) **Up:** [Direct Link Networks](#) **Previous:** [Further Reading](#)

Packet Switching

- [Problem: Not All Machines Are Directly Connected](#)
 - [Switching and Forwarding](#)
 - [Routing](#)
 - [Cell Switching \(ATM\)](#)
 - [Switching Hardware](#)
 - [Summary](#)
 - [Open Issue: What Makes a Route Good?](#)
 - [Further Reading](#)
 - [Exercises](#)
-

Problem: Not All Machines Are Directly Connected

The directly connected networks described in the previous chapter suffer from two limitations. First, there is a limit to how many hosts can be attached. For example, only two hosts can be attached to a point-to-point link, and an Ethernet can connect up to only 1024 hosts. Second, there is a limit to how large a geographic area a single network can serve. For example, an Ethernet can span only 1500 meters, and even though point-to-point links can be quite long, they do not really serve the area between the two ends. Since our goal is to build networks that can be global in scale, the next problem is to enable communication between hosts that are not directly connected.

This problem is not unlike the one addressed in the telephone network: your phone is not directly connected to every person you might want to call, but instead is connected to an exchange which contains a *switch*. It is the switches that create the impression that you have a connection to the person at the other end of the call. Similarly, computer networks use *packet switches* (as distinct from the *circuit switches* used for telephony) to enable packets to travel from one host to another, even when no direct connection exists between those hosts. This chapter introduces the major concepts of packet switching, which lies at the heart of computer networking.


A packet switch is a device with several inputs and outputs leading to and from the hosts that the switch interconnects. There are three main problems that a packet switch must address. First, the core job of a switch is to take packets that arrive on an input and *forward* (or *switch*) them to the right output so that they will reach their appropriate destination. Knowing which output is the right one requires the switch to know something about the possible routes to the destination. The process of accumulating and sharing this knowledge, the second problem for a packet switch, is called *routing*. Third, a switch must deal with the fact that its outputs have a certain bandwidth. If the number of packets arriving at a switch that need to be sent out on a certain output exceeds the capacity of that output, then we have a problem of *contention*. The switch queues (buffers) packets until the contention subsides, but if it lasts too long, the switch will run out of buffer space and be forced to discard packets. When packets are discarded too frequently, the switch is said to be *congested*. The ability of a switch to handle contention is a key aspect of its performance, with many high performance switches using exotic hardware to reduce the effects of contention.

This chapter introduces these three issues of packet switching: forwarding, routing, and contention. For the most part, these discussions apply to a wide range of packet switched technologies. There is one particular technology, however that has attracted so much attention of late that it warrants some more specific attention. That technology is *asynchronous transfer mode* (ATM). The third section of this chapter introduces ATM, providing a handy backdrop for a discussion of the hardware that can be used to implement a packet switch, which is the subject of the final section. This discussion of switches focuses on contention; we postpone the related problem of congestion until Chapter [10](#).

[Next](#) [Up](#) [Previous](#)

Next: [Switching and Forwarding](#) **Up:** [Packet Switching](#) **Previous:** [Packet Switching](#)

Switching and Forwarding

In the simplest terms, a switch is a mechanism that allows us to interconnect links to form a larger network. It does this by adding a star topology (see Figure ) to the point-to-point link, bus (Ethernet) and ring (FDDI) topologies established in the last chapter. There are several nice features to a star topology:

- We can connect hosts to the switch using point-to-point links, which typically means that we can build networks of large geographic scope;
- Even though a switch has a fixed number of inputs and outputs, which limits the number of hosts that can be connected to a single switch, large networks can be built by interconnecting a number of switches.
- Adding a new host to the network by connecting it to the switch does not necessarily mean that the hosts already connected will get worse performance.

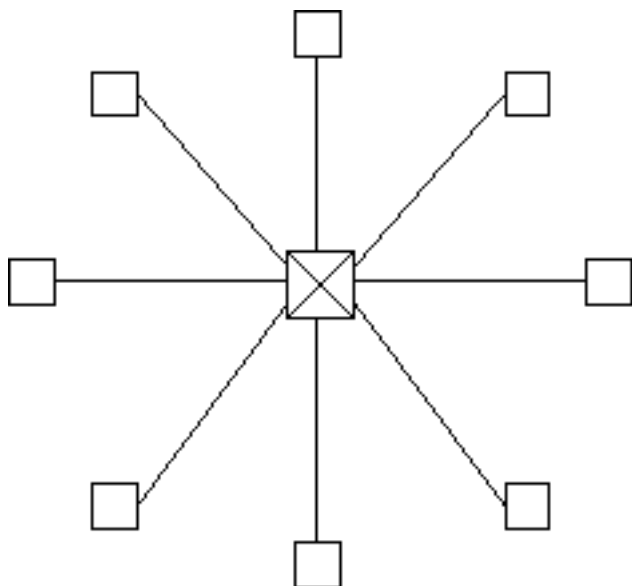


Figure: A switch provides a star topology.

This last claim cannot be made for the shared media networks discussed in the last chapter. For example, it is impossible for two hosts on the same Ethernet to transmit continuously at 10Mbps because they share the same transmission medium. Every host on a switched network has its own link to the switch, so it may be entirely possible for many hosts to transmit at the full link speed (bandwidth), provided the switch is designed with enough *aggregate* capacity. Providing high aggregate throughput is one of the design goals for a switch; we return to this topic in Section [10.1](#). In general, switched networks are considered more *scalable* (i.e., more capable of growing to large numbers of nodes) than shared media networks because of this ability to support many hosts at full speed.

...text deleted...

Source Routing

Virtual Circuit Switching

Datagrams

Implementation and Performance

Next	Up	Previous
----------------------	--------------------	--------------------------

Next: [Routing](#) **Up:** [Packet Switching](#) **Previous:** [Problem: Not All](#)

Routing

The preceding discussion assumes that either the hosts (in the case of source routing) or the switches have enough knowledge of the network topology so they can choose the right port onto which each packet should be output. In the case of virtual circuits, routing is an issue only for the connection request packet; all subsequent packets follow the same path as the request. In datagram networks, routing is an issue for every packet. In either case, a switch needs to be able to look at the packet's destination address and determine which of the output ports is the best choice to get the packet to that address. As we saw in Section [1.4](#), the switch makes this decision by consulting a table that we called the *forwarding table*. (This table is often called a *routing table*.) The fundamental problem of routing is, how do switches acquire the information in their forwarding tables.

We need to make an important distinction, which is often neglected, between forwarding and routing. Forwarding consists of taking a packet, looking at its destination address, consulting a table, and sending the packet in a direction determined by that table. Routing is the process by which forwarding tables are built, and is a topic to which people devote entire careers.

Network as a Graph

Distance Vector

Link State

Metrics

Routing, Addressing and Hierarchy

Copyright 1996, Morgan Kaufmann Publishers

Cell Switching (ATM)

Now that we have discussed some of the general issues associated with switched networks, it is time to focus on one particular switching technology: *asynchronous transfer mode* (ATM). ATM has become a tremendously important technology in recent years for a variety of reasons, not least of which is that it has been embraced by the telephone industry, which has historically been less than active in data communications except as a supplier of links on top of which other people have built networks. ATM also happened to be in the right place at the right time, as a high speed switching technology that appeared on the scene just when shared media like Ethernet and FDDI were starting to look a bit too slow for many users of computer networks.

ATM is a connection-oriented packet-switched network, which is to say, it uses virtual circuits very much in the manner described in Section [1.4](#). In ATM terminology, the connection setup phase is called *signalling*. At the time of this writing, the ATM Forum (the standards body that governs ATM) is still hammering out the details of an ATM signalling protocol known as Q.2931. In addition to discovering a suitable route across an ATM network, Q.2931 is also responsible for allocating resources at the switches along the circuit. This is done in an effort to ensure the circuit a particular quality of service. We return to this topic in Chapter [4](#), where we discuss it in the context of similar efforts to implement QoS.

The thing that makes ATM really unusual is that the packets that are switched in an ATM network are of fixed length. That length happens to be 53 bytes---5 bytes of header followed by 48 bytes of payload---a rather interesting choice discussed in more detail below. To distinguish these fixed length packets from the more common variable length packets normally used in computer networks, they are given a special name: *cells*. ATM may be thought of as the canonical example of cell switching.

Cells

All the packet switching technologies we have looked at so far have used variable length packets. Variable length packets are normally constrained to fall within some bounds. The lower bound is set by the minimum amount of information that needs to be contained in the packet, which is typically a header with no optional extensions. The upper bound may be set by a variety of factors; the maximum FDDI packet size, for example, determines how long each station is allowed to transmit without passing on the token, and thus, determines how long a station might have to wait for the token to reach it. Cells, in contrast, are both fixed length and small in size. While this seems like a simple enough design choice, there are actually a lot of factors involved, as explained in the following paragraphs.

Cell Size

Variable length packets have some nice characteristics. If you only have one byte to send (e.g. to acknowledge the receipt of a packet), you put it in a minimum-sized packet. If you have a large file to send, however, you break it up into as many maximum-sized packets as you need. You do not need to send any extraneous padding in the first case, and in the second, you drive down the ratio of header to data bytes, thus increasing bandwidth efficiency. You also minimize the total number of packets sent, thereby minimizing the total processing incurred by per-packet operations. This can be particularly important in obtaining high throughput, since many network devices are limited not by how many *bits* per second they can process, but rather by the number of *packets* per second.

So, why fixed length cells? One of the main reasons was to facilitate the implementation of hardware switches. When ATM was being created in the mid and late 80's, 10 Mbps Ethernet was the cutting edge technology in terms of speed. To go much faster, most people thought in terms of hardware. Also, in the telephone world, people think big when they think of switches---telephone switches often serve tens of thousands of customers. Fixed length packets turn out to be a very helpful thing if you want to build fast, highly scalable switches. There are two main reasons for this:

- it is easier to build hardware to do simple jobs, and the job of processing packets is simpler when you already know how long each one will be;
- if all packets are the same length, then you can have lots of switching elements all doing much the same thing in parallel, each of them taking the same time to do its job.

This second reason, enabling parallelism, greatly improves the scalability of switch designs. We will examine a highly scalable, parallel switch in Section [4.1](#). It would be overstating the case to say that fast, parallel hardware switches can only be built using fixed-length cells. However, it is certainly true that cells ease the task of building such hardware, and that there was a lot of knowledge about how to build cell switches in hardware at the time ATM standards were being defined.

Another nice property of cells relates to the behavior of queues. Queues build up in a switch when traffic from several inputs may be heading for a single output. In general, once you extract a packet from a queue and start transmitting it, you need to continue until the whole packet is transmitted; it is not

practical to preempt the transmission of a packet. (Recall that this idea is at the heart of statistical multiplexing.) The longest time that a queue output can be tied up is equal to the time it takes to transmit a maximum sized packet. Fixed length cells mean that a queue output is never tied up for more than the time it takes to transmit one cell, which is almost certainly shorter than the maximally sized packet on a variable length packet network. Thus, if tight control over the latency experienced by cells when they pass through a queue is important, cells provide some advantage. Of course, long queues can still build up and there is no getting around the fact that some cells will have to wait their turn. What you get from cells is not much shorter queues, but potentially finer control over the behavior of queues.


An example will help to clarify this idea. Imagine a network with variable length packets, where the maximum packet length is 4KB and the link speed is 100 Mbps. The time to transmit a maximum sized packet is $4096 \times 8 / 100 = 327.68 \mu\text{s}$. Thus, a high priority packet that arrives just after the switch starts to transmit a 4KB packet will have to sit in the queue $327.68 \mu\text{s}$ waiting for access to the link. In contrast, the longest wait if the switch were forwarding 53-byte cells would be $53 \times 8 / 100 = 4.24 \mu\text{s}$. This may not seem like a big deal, but the ability to control delay, and especially its variation with time (jitter), can be important for some applications.

Queues of cells also tend to be a little shorter than queues of packets for the following reason. When a packet begins to arrive in an empty queue, it is typical for the switch to wait for the whole packet to arrive before it can start transmitting the packet on an outgoing link. This means that the link sits idle while the packet arrives. However, if you imagine a large packet being replaced by a 'train' of small cells, then as soon as the first cell in the train has entered the queue, the switch can transmit it. Imagine in the example above what would happen if two 4KB packets arrived in a queue at about the same time. The link sits idles for $327.68 \mu\text{s}$ while these two packets arrive, and at the end of that period we have 8KB in the queue. Only then can the queue start to empty. If those same two packets were sent as trains of cells, then transmission of the cells could start $4.24 \mu\text{s}$ after the first train starts to arrive. At the end of $327.68 \mu\text{s}$, the link has been active for a little over $323 \mu\text{s}$ and there are just over 4KB of data left in the queue, not 8KB as before. Shorter queues mean less delay for all the traffic.

Having decided to use small fixed-length packets, the next question is, what is the right length to fix them at? If you make them too short, then the amount of header information that needs to be carried around relative to the amount of data that fits in one cell gets larger, so the percentage of link bandwidth that is actually used to carry data goes down. Even more seriously, if you build a device that processes cells at some maximum number of cells per second, then as cells get shorter, the total data rate drops in direct proportion to cell size. An example of such a device might be a network adapter that reassembles cells into larger units before handing them up to the host. The performance of such a device directly depends on cell size. On the other hand, if you make cells too big, then there is a problem of wasted bandwidth caused by the need to pad transmitted data to fill a complete cell. If the cell payload size is 48 bytes and you want to send 1 byte, you'll need to send 47 bytes of padding. If this happens a lot, then the utilization of the link will be very low.

Efficient link utilization is not the only factor that influences cell size. For example, cell size has a

particular effect on voice traffic, and since ATM grew out of the telephony community, one of the major concerns was that it be able to carry voice effectively. The standard digital encoding of voice is done at 64Kbps (8-bit samples taken at 8 KHz). To maximize efficiency, you want to collect a full cell's worth of voice samples before transmitting a cell. A sampling rate of 8 KHz means that one byte is sampled every 125 μ s, so the time it takes to fill an n -byte cell with samples is $n \times 125\mu$ s. If cells are, say, 1000 bytes long, it would take 125ms just to collect a full cell of samples before you even start to transmit it to the receiver. That sort of latency starts to be quite noticeable to a human listener. Even considerably smaller latencies create problems for voice, particularly in the form of echoes. Echoes can be eliminated by a piece of technology called echo cancellors, but these add cost to a telephone network that many network operators would rather avoid.


All of the above factors caused a great deal of debate in the international standards bodies when ATM was being standardized, and the fact that no length was perfect in all cases was used by those opposed to ATM to argue that fixed length cells were a bad idea in the first place. As is so often the case with standards, the end result was a compromise that pleased almost no-one: 48 bytes was chosen as the length for the ATM cell payload. Probably the greatest tragedy of this choice was that it is not a power of two, which means that it is quite a mismatch to most things that computers handle, like pages and cache lines. Rather less controversially, the header was fixed at 5 bytes. The format of an ATM cell is shown in Figure .

Sidebar: A Compromise of 48 Bytes

The explanation for why the payload of an ATM cell is 48 bytes is an interesting one, and makes an excellent case for studying the process of standardization. As the ATM standard was evolving, the US telephone companies were pushing a 64-byte cell size, while the European companies were advocating 32-byte cells. The reason the Europeans wanted the smaller size is that since the countries they served were of a small enough size, they would not have to install echo cancellors if they were able to keep the latency induced by generating a complete cell small enough. Thirty-two byte cells were adequate for this purpose. In contrast, the US is a large enough country that the phone companies had to install echo cancellors anyway, and so the larger cell size reflected a desire to improve the header-to-payload ratio.

It turns out that averaging is a classic form of compromise---48 bytes is simply the average of 64 bytes and 32 bytes. So as not to leave the false impression that this use of compromise-by-averaging is an isolated incident, we note that the 7-layer OSI model was actually a compromise between 6 and 8 layers.

Cell Format

The ATM cell actually comes in two different formats, depending on where you look in the network. The one shown in Figure  is called the UNI (user-network interface) format; the alternative is the NNI (network-network interface). The UNI format is used when transmitting cells between a host and a switch, while the NNI format is used when transmitting cells between switches. The only difference is that the NNI format replaces the GFC field with four extra bits of VPI. Clearly, understanding all the three letter acronyms (TLAs) is a key part of understanding ATM.

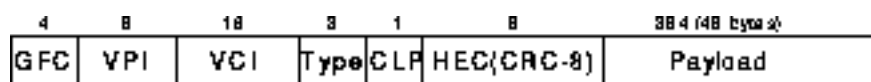





Figure: ATM cell format at the UNI

Starting from the leftmost byte of the cell (which is the first one transmitted), the UNI cell has 4 bits for 'Generic Flow Control' (GFC). The use of these bits is not well defined at the time of writing, but they are intended to have local significance at a site and may be overwritten in the network. The basic idea behind the GFC bits is to provide a means to arbitrate access to the link if the local site uses some shared medium to connect to ATM.

The next 24 bits contain a 16-bit Virtual Circuit Identifier (VCI) and a 8-bit Virtual Path Identifier (VPI). The difference between the two is explained below, but for now it is adequate to think of them as a single 24-bit identifier that is used to identify a virtual connection, just as in Section . Following the VCI/VPI is 3-bit Type field that has eight possible values. Four of them, when the first bit in the field is set, relate to management functions. When that bit is clear, this indicates the cell contains user data. In this case, the second bit is the 'explicit forward congestion indication' (EFCI) bit and the third is the 'user signalling' bit. The former can be set by a congested switch to tell an end node it is congested; it has its roots in the DECbit described in Section  and, although the standards are not yet firm on this point, is intended to be used similarly. The latter is used primarily in conjunction with ATM Adaptation Layer 5 to delineate frames, as discussed below.

Next is a bit to indicate 'Cell Loss Priority' (CLP); a user or network element may set this bit to indicate cells that should be dropped preferentially in the event of overload. For example, a video coding application could set this bit for cells that, if dropped, would not dramatically degrade the quality of the video. A network element might set this bit for cells that have been transmitted by a user in excess of the amount that was negotiated.

The last byte of the header is an 8-bit CRC, known as the 'Header Error Check' (HEC). It uses the CRC-8 polynomial given in Section , and provides error detection and single-bit error correction capability on the cell header only. Protecting the cell header is particularly important because an error in the VCI will cause the cell to be mis-delivered.

Segmentation and Reassembly

Up to this point, we have assumed that a low level protocol could just accept that packet handed down to it by a high level protocol, attach its own header, and pass the packet on down. This is not possible with ATM, however, since the packets handed down from above are often larger than 48 bytes, and thus, will not fit in the payload of an ATM cell. The solution to this problem is to *fragment* the high-level message into low-level packets on the source, transmit the individual low-level packets over the network, and then *reassemble* the fragments back together at the destination. This general technique is usually called *fragmentation and reassembly*. In the case of ATM, however, it is often called *segmentation and reassembly*.

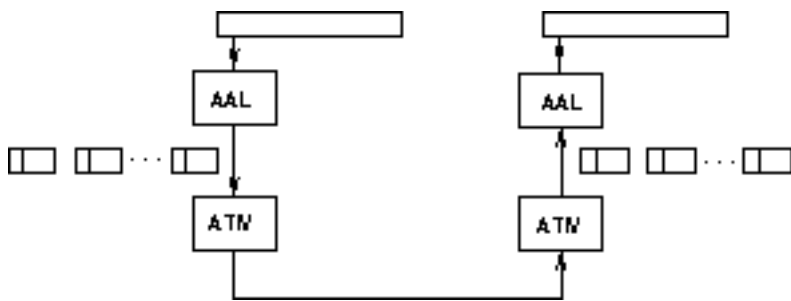



Figure: Segmentation and reassembly in ATM.

Segmentation is not unique to ATM, but it is much more of a problem than in a network with a maximum packet size of, say, 1500 bytes. To address the issue, a protocol layer was added that sits between ATM and the variable length packet protocols that might use ATM, such as IP. This layer is called the ATM Adaptation Layer (AAL), and to a first approximation, the AAL header simply contains the information needed by the destination to reassemble the individual cells back into the original message. The relationship between the AAL and ATM is illustrated in Figure .

Because ATM was designed to support all sorts of services, including voice, video, and data, it was felt that different services would have different AAL needs. Thus, four adaptation layers were originally defined: 1 and 2 were designed to support applications, like voice, that require guaranteed bit rates, while 3 and 4 were intended to provide support for packet data running over ATM. The idea was that AAL3 would be used by connection-oriented packet services (such as X.25) and AAL4 would be used by connectionless services (such as IP). Eventually, the reasons for having different AALs for these two types of service were found to be insufficient, and the AALs merged into one that is inconveniently known as AAL3/4. Meanwhile, some perceived shortcomings in AAL3/4 caused a fifth AAL to be proposed, called AAL5. Thus there are now four AALs: 1, 2, 3/4 and 5.

The two that support computer communications are described below.

ATM Adaptation Layer 3/4

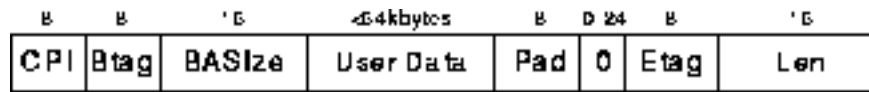


Figure: ATM Adaptation Layer 3/4 packet format

The main function of AAL3/4 is to provide enough information to allow variable length packets to be transported across the ATM network as a series of fixed length cells. That is, the AAL supports the segmentation and reassembly process. Since we are now working at a new layer of the network hierarchy, convention requires us to introduce a new name for a packet---in this case, they are called *protocol data units* (PDUs). The task of segmentation/reassembly involves two different packet formats. The first of these is the 'convergence sublayer protocol data unit' (CS-PDU), as depicted in Figure [link]. The CS-PDU defines a way of encapsulating variable length PDUs prior to segmenting them into cells. The PDU passed down to the AAL layer is encapsulated by adding a header and a trailer, and the resultant CS-PDU is segmented into ATM cells.

The CS-PDU format begins with an 8-bit 'common part indicator' (CPI), which is like a version field, indicating which version of the CS-PDU format is in use. Only the value zero is currently defined. The next 8 bits contain the 'beginning tag' (Btag), and it is supposed to match the 'end tag' (Etag) for a given PDU. This protects against the situation where the loss of the last cell of one PDU and the first cell of another causes two PDUs to be inadvertently joined into a single PDU and passed up to the next layer in the protocol stack. The BAsize field (Buffer Allocation size) is not necessarily the length of the PDU (which appears in the trailer); it is supposed to be a hint to the reassembly process as to how much buffer space to allocate for the reassembly. The reason for not including the actual length here is that the sending host might not have known how long the CS-PDU was when it transmitted the header.

The CS-PDU trailer contains the Etag, the real length of the PDU, and a padding byte of zeroes.

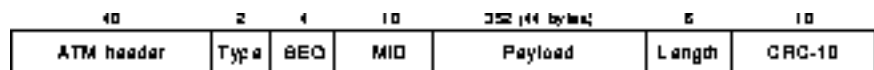


Figure: ATM cell format for AAL3/4

The second part of AAL3/4 is the header and trailer that is carried in each cell, as depicted in Figure [link]. Thus, the CS-PDU is actually segmented into 44-byte chunks; an AAL3/4 header and trailer is attached to each one, bringing it up to 48 bytes, which is then carried as the payload of an ATM cell.

The first two bits of the AAL3/4 header contain the Type field, which indicates if this is the first cell of a CS-PDU, the last cell of a CS-PDU, a cell in the middle of a CS-PDU, or a single cell PDU (in which case it is both first and last). The official names for these four conditions are shown in Table [link], along with the bit encodings.

Value	Name	Meaning
10	BOM	Beginning of Message
00	COM	Continuation of Message
01	EOM	End of Message
11	SSM	Single Segment Message


Table: AAL3/4 type field.

Next is a 4-bit sequence number (SEQ), which is intended simply to detect cell loss or misordering so that reassembly can be aborted. Clearly, a sequence number this small can miss cell losses if the number of lost cells is large enough. This is followed by a multiplexing identifier (MID), which can be used to multiplex several PDUs onto a single connection. The 6-bit Length field contains the number of bytes of PDU that are contained in the cell; it must equal 44 for BOM and COM cells. Finally, a 10-bit CRC is used to detect errors anywhere in the 48-byte cell payload.

One thing to note about AAL 3/4 is that it exacerbates the fixed per-cell overhead that we discussed above. With 44 bytes of data to 9 bytes of header, the best possible bandwidth utilization falls to 83%.

ATM Adaptation Layer 5

One thing you may have noticed in the discussion of AAL3/4 is that it seems to take a lot of fields, and thus a lot of overhead to perform the conceptually simple function of segmentation and reassembly. This observation was, in fact, made by several people in the early days of ATM, and numerous competing proposals arose for an AAL to support computer communications over ATM. There was a movement known informally as 'Back the Bit' which argued that, if we could just have one bit in the ATM header (as opposed to the AAL header) to delineate the end of a frame, then segmentation and reassembly could be accomplished without using any of the 48-byte ATM payload for segmentation/reassembly information. This movement eventually led to the definition of the user signalling bit described above, and to the standardization of AAL5.

What AAL5 does is replace the 2-bit Type field of AAL3/4 with 1 bit of framing information in the ATM cell header. By setting that one bit, we can identify the last cell of a PDU; the next cell is assumed to be the first cell of the next PDU, and subsequent cells are assumed to be COM cells until another cell is received with the user signalling bit set. All the pieces of AAL3/4 that provide protection against lost, corrupt, or misordered cells, including the loss of an EOM cell, are provided by the AAL5 CS-PDU packet format, depicted in Figure .

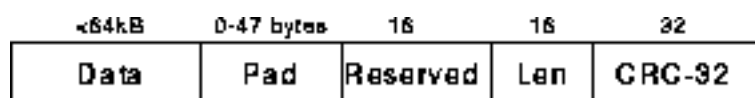


Figure: ATM Adaptation Layer 5 packet format

The AAL5 CS-PDU consists simply of the data portion (the PDU handed down by the higher layer protocol) and an 8-byte trailer. To make sure that the trailer always falls at the tail end of an ATM cell, there may be up to 47 bytes of padding between the data and the trailer. The first two bytes of the trailer are currently reserved and must be zero. The length field is the number of bytes carried in the PDU, not including the trailer or any padding before the trailer. Finally, there is a 32-bit CRC.

Somewhat surprisingly, AAL5 provides almost the same functionality as AAL3/4 without using 4 bytes out of every cell. For example, the CRC-32 detects lost or misordered cells, as well as bit errors in the data. In fact, having a checksum over the entire PDU rather than doing it on a per-cell basis as in AAL3/4 provides stronger protection. For example, it protects against the loss of 16 consecutive cells, an event that would not be picked up by the sequence number checking of AAL3/4. Also, a 32-bit CRC protects against longer burst errors than a 10-bit CRC.

The main feature missing from AAL5 is the ability to provide an additional layer of multiplexing onto one virtual circuit using the MID. It is not clear whether this is a significant loss. For example, if one is being charged for every virtual circuit that one sets up across a network, then multiplexing traffic from lots of different applications onto one connection might be a plus. However, this approach has the drawback that all applications will have to live with whatever quality of service (e.g., delay and bandwidth guarantees) has been chosen for that one connection, which may mean that some applications are not receiving appropriate service. Certainly the large (24-bit) space available for VCI/VPI combinations suggests that it should be possible for a host to open many virtual connections and avoid multiplexing at this level, in which case the MID is of little value.

In general, AAL5 has been wholeheartedly embraced by the computer communications community (at least, by that part of the community that has embraced ATM at all). For example, it is the preferred AAL in the IETF for transmitting IP datagrams over ATM. Its more efficient use of bandwidth and simple design are the main things that make it more appealing than AAL3/4.

Virtual Paths

As mentioned above, ATM uses a 24-bit identifier for virtual circuits, and these circuits operate almost exactly like the ones described in Section [10.1.1](#). The one twist is that the 24-bit identifier is split into two parts: an 8-bit Virtual Path Identifier (VPI) and a 16-bit Virtual Circuit Identifier (VCI). What this does is provide some hierarchy in the identifier, just as there may be hierarchy in addresses as we discussed in Section [10.1.1](#). To understand how a hierarchical virtual circuit identifier might be used, consider the following example. (We ignore the fact that in some places there might be a Network-network interface (NNI) with a different sized VPI; just assume that 8-bit VPIs are used everywhere.)

Suppose that a corporation has two sites that connect to a public ATM network, and that at each site the

corporation has a network of ATM switches. We could imagine establishing a virtual path between two sites using only the VPI field. Thus, the switches in the public network would use the VPI as the only field on which to make forwarding decisions. From their point of view, this is a virtual circuit network with 8-bit circuit identifiers. The 16-bit VCI is of no interest to these public switches, and they neither use it for switching nor remap it. Within the corporate sites, however, the full 24-bit space is used for switching. Any traffic that needs to flow between the two sites is routed to a switch that has a connection to the public network, and its top 8 bits (the VPI), is mapped onto the appropriate value to get the data to the other site. This idea is illustrated in Figure [10.10](#). Note that the virtual path acts like a fat pipe that contains a bundle of virtual circuits, all of which have the same 8 bits in their most significant byte.

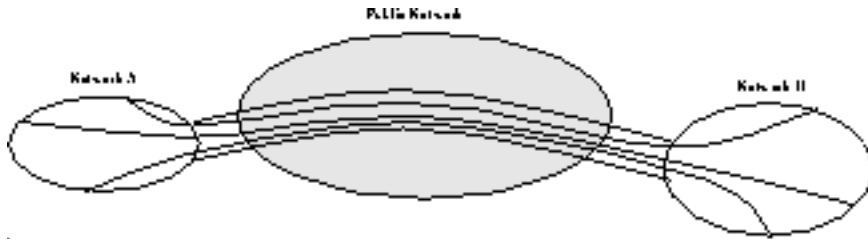



Figure: Example of Virtual Path

The advantage of this approach is clear: although there may be thousands or millions of virtual connections across the public network, the switches in the public network behave as if there is only one connection. This means that there can be much less connection state stored in the switches, avoiding the need for big, expensive tables of per-VCI information.

Physical Layers for ATM

While the layered approach to protocol design might lead you to think that we do not need to worry about what type of point-to-point link ATM runs on top of, this turns out not to be the case. From a simple pragmatic point of view, when you buy an ATM adaptor for a workstation or an ATM switch, it comes with some physical medium over which ATM cells will be sent. Of course, this is true for other networking protocols such as FDDI and Ethernet. Like these protocols, ATM can also run over several physical media and physical layer protocols.


From early in the process of standardizing ATM, it has been assumed that ATM will run on top of a SONET physical layer (see Section [10.11](#)). Some people even get ATM and SONET confused because they have been so tightly coupled for so long. While it is true that standard ways of carrying ATM cells inside a SONET frame have been defined, and that you can now buy ATM-over-SONET products, the two are entirely separable. For example, one could lease a SONET link from a phone company and send whatever one wants over it, including variable length packets. Also, one can send ATM cells over many other physical layers instead of SONET, and standards have been (or are being) defined for these encapsulations.

When you send ATM cells over some physical medium, the main issue is how to find the boundaries of the ATM cells; this is exactly the framing problem described in Chapter . With SONET, there are two easy ways to do this. One of the overhead bytes in the SONET frame can be used as a pointer into the SONET payload to the start of an ATM cell. Having found the start of one cell, the next cell starts 53 bytes further on in the SONET payload, and so on. In theory, you only need to read this pointer once, but in practice, it makes sense to read it every time the SONET overhead goes by so that you can detect errors or resynchronize if needed.

The other way to find the edges of ATM cells takes advantage of the fact that every cell has a CRC in the 5th byte of the cell. Thus, if you run a CRC calculation over the last 5 bytes received and the answer comes out to indicate no errors, then it is probably true that you just read an ATM header. If this happens several times in a row at 53 byte intervals, you can be pretty sure you have found the cell boundary.

Sidebar: ATM in the LAN

As we mentioned above, ATM grew out of the telephony community, who envisioned it being used as a way to build large public networks that could transport voice, video and data traffic. However, it was subsequently embraced by the computer and data communications industries as a technology to be used in LANS---a replacement for Ethernet and FDDI. Its popularity in this realm can be attributed to many factors, most notably the fact that it offered significantly higher bandwidth than Ethernet and, unlike FDDI, the bandwidth is switched rather than shared, meaning that, in theory, every host can send or receive at the full link speed.

The problem with running ATM in a LAN, however, is that it doesn't look like a 'traditional' LAN. Because most LANs (i.e. Ethernets and token rings) are shared media networks, i.e., every node on the LAN is connected to the same link, it is easy to implement things like broadcast (sending to everybody) and multicast (sending to a group). Thus, lots of the protocols that people depend on in their LANs---e.g., the Address Resolution Protocol (ARP) described in Section ---depend in turn on the ability of all LANs to support multicast and broadcast. However, because of its connection-oriented and switched nature, ATM behaves rather differently shared media LAN. For example, how can you broadcast to all nodes on an ATM LAN if you don't know all their addresses? There are two possible solutions to this problem, and both of them have been explored. One is to redesign the protocols that assume things about LANs which are not in fact true of ATM. Thus, for example, there is a new protocol called ATMARP which, unlike traditional ARP, does not depend on broadcast. The alternative is to make ATM behave more like a shared media LAN, in the sense of supporting multicast and

broadcast---without losing the performance advantages of a switched network. This approach has been developed by the ATM Forum under the title `LAN Emulation' (which might be more correctly titled `Shared Media Emulation'). This approach aims to add functionality to ATM LANs so that anything that runs over a shared media LAN can operate over an ATM LAN.

[Next](#) [Up](#) [Previous](#)

Next: [Switching Hardware](#) **Up:** [Packet Switching](#) **Previous:** [Routing](#)

Switching Hardware

Whether a switch has to handle virtual circuits or datagrams, variable length packets or fixed length cells, the basic issues that have to be addressed remain the same. A switch is a multi-input, multi-output device, and its job is to get as many packets as possible from the inputs to the appropriate outputs. Most of what we have talked about in the previous sections has revolved around deciding which output to send a packet to. In this section, we look at how the process of getting packets from the inputs to the outputs as fast as possible.

Recall from Section [1.1](#) that the performance of a switch implemented in software on a general-purpose processor is limited by the bandwidth of machine's I/O bus (amongst other factors). When you consider that each packet must traverse the bus twice---once from the adaptor to memory and once from memory to the adaptor---it is easy to see that a processor with a 1Gbps I/O bus would be able to handle at most ten T3 (45Mbps) links, up to three STS-3 (155Mbps) links, and not even one STS-12 (622Mbps) link. Considering that the whole purpose of a switch is to connect reasonably large numbers of links, this is not an ideal situation. Fortunately, switches can be implemented by special-purpose hardware, as we now discuss.

Design Goals

Ports and Fabrics

Crossbar Switches

Self-Routing Fabrics

Shared Media Switches

Copyright 1996, Morgan Kaufmann Publishers

Summary

This chapter has started to look at some of the issues involved in building large, scalable networks by using switches, rather than just links, to interconnect hosts. There are several different ways to decide how to switch packets, the two main ones being the datagram (connectionless) model and the virtual circuit (connection-oriented) model.

One of the hardest parts of building large switched networks is figuring out what direction to send a packet from any given node in the network so that it will reach its intended destination. This is the routing problem, and what makes it hard is that no-one node has global knowledge of the network. The solution is that every node must be involved in a routing protocol to exchange information with its neighbors in an effort to build up a reasonable global picture. Such protocols have to be robust in the face of node and link failures, and must adapt to the addition of new nodes and links. Ideally, they should also be sensitive to the different characteristics of the links such as latency, throughput, and the current load. In this context, we studied two general routing algorithms: distance-vector and link-state.

Independent of the switching model and how routing is done, switches need to forward packets from inputs to outputs at a high rate, and in some circumstances, switches need to grow to a large size to accommodate hundreds or thousands of ports. Building switches that both scale and offer high performance is complicated by the problem of contention, and as a consequence, switches often employ special-purpose hardware rather than being built from general-purpose workstations. Some switch designers have targeted the 'central office switch', which would allow the telephone companies to provide ATM to everyone's home instead of a simple voice telephone line. Building such large, fast switches is a tough problem which has yielded a wealth of clever switch architectures.

In addition to the issues for forwarding, routing, and contention, we observe that the related problem of congestion has come up at several points throughout this chapter. We postpone a discussion of congestion until Chapter [□](#), after we have seen more of the network architecture. This is because it is impossible to fully appreciate congestion (both the problem and how to address it) without understanding both what happens inside the network (the topic of this and the next chapter) and what happens at the edges of the network (the topic of Chapter [□](#)).

Open Issue: What Makes a Route Good?

It is tempting to think that all the hard problems of routing have been solved by the algorithms described in this chapter. This is certainly not the case, as the large number of people who continue to work on routing problems will attest. One challenge for routing is the support of mobile hosts. If you have a wireless link to your laptop and roam around the country, or even if you unplug it from the wall in New Jersey and take it to Arizona and re-plug it, how can packets be efficiently delivered? Moreover, will these mechanisms be scalable enough to support an environment where much of the world's population has a mobile computing device? This is one of the hard problems currently being tackled.

Also, routing promises to become even more difficult as application requirements become more demanding, and networks try to provide a range of qualities of service that meet different application needs. For example, a high resolution video broadcast application would probably prefer to be routed over a high speed satellite link than over a lower speed, lower latency terrestrial link, whereas an interactive video application with a lesser bandwidth requirement might prefer the lower speed, lower latency link. In other words, link cost becomes a function of application needs, not something that can be calculated in isolation.

Coupled with all these problems is the unavoidable fact that networks continue to grow beyond the expectations of the original designers. Thus, the scaling problems for routing continues to get harder, something we will look at more closely in the next chapter.

Further Reading

Among the more accessible and interesting papers on routing are the two papers that describe the 'new' and the 'revised' ARPANET routing mechanisms. Both make this chapter's reading list. The third paper gives a broader perspective on routing. The current level of interest in ATM has yielded plenty of survey papers. We recommend the Lyles-Swinehart paper as a good early example, partially because it discusses ATM from the perspective of the desktop rather than the perspective of the telephone company. Finally, the last paper describes the Sunshine switch, and is especially interesting because it provides insights into the important role of traffic analysis in switch design. In particular, the Sunshine designers were among the first to realize that cells were unlikely to arrive at a switch in a totally uncorrelated way and to factor this into their design.

- J. McQuillan et al. The new routing algorithm for the ARPANET. *IEEE Transactions on Communications*, COM-28(5), 711--719, May 1980.
- A. Khanna and J. Zinky. The revised ARPANET routing metric. *Proceedings of the SIGCOMM '88 Symposium*, pages 45--56, August 1988.
- M. Schwartz and T. Stern. Routing techniques used in computer communication networks. *IEEE Transactions on Communications*, COM-28(4): 539--552, April 1980.
- J. Lyles and D. Swinehart. The emerging gigabit environment and the role of local ATM. *IEEE Communications*, 30(4): 52--58, April 1992.
- J. N. Giacopelli *et al.* Sunshine: a high performance self-routing broadband packet switch architecture. *IEEE Journal on Selected Areas in Communications*, 9(8): 1289--1298, October 1991.

A good general overview of routing protocols and issues can be found in [\[Per92\]](#) and [\[Per93\]](#). The former includes a description of the circumstances that led to the ARPANET crash in 1981.

For more information on ATM, one of the early overview books is [\[Pry91\]](#). As one of the key ATM standards-setting bodies, the ATM Forum produces new specifications for ATM, the User Network

Interface (UNI) specification version 3.1 being the most recent at the time of this writing. (See live reference below.)

There have been literally thousands of papers published on switch architectures. One early paper that explains Batcher networks well is, not surprisingly, one by Batcher himself [Bat68]. Sorting networks are explained in [DY75], and the Knockout switch is described in [YHA87]. A reasonable survey of ATM switch architectures appears in [Par94], and a good overview of the performance of different switching fabrics can be found in [T. 93]. An example of a modern non-ATM switch can be found in [GG94].

An excellent text to read if you want to learn about the mathematical analysis of network performance is [Kle75], by one of the pioneers of the ARPANET. Many papers have been published on the applications of queueing theory to packet switching. We recommend [PF94] as a recent contribution focused on the Internet, and [LTWW94], a significant paper that introduces the important concept of 'long-range dependence' and shows the inadequacy of many traditional approaches to traffic modeling.

Finally, we recommend the following live reference:

- <http://www.atmforum.com>: current activities of the ATM Forum.

Next	Up	Previous
------	----	----------

Next: [Exercises](#) **Up:** [Packet Switching](#) **Previous:** [Open Issue: What](#)

[Next](#) [Up](#) [Previous](#)

Next: [Internetworking](#) **Up:** [Packet Switching](#) **Previous:** [Further Reading](#)

Exercises

Copyright 1996, Morgan Kaufmann Publishers

Internetworking

- [Problem: There Is More than One Network](#)
 - [Bridges and Extended LANs](#)
 - [Simple Internetworking \(IP\)](#)
 - [Global Internet](#)
 - [Next Generation IP](#)
 - [Multicast](#)
 - [Host Names \(DNS\)](#)
 - [Summary](#)
 - [Open Issue: IP versus ATM](#)
 - [Further Reading](#)
 - [Exercises](#)
-

Problem: There Is More than One Network

We have now seen how to build a single network using point-to-point links, shared media, and switches. The problem is that lots of people have built networks with these various technologies and they all want to be able to communicate with each other, not just with the other users of a single network. This chapter is about the problems of interconnecting different networks.

There are two important problems that must be addressed when connecting networks: *heterogeneity* and *scale*. The problem of heterogeneity is essentially this: users on one type of network want to be able to communicate with users on different types of networks. To further complicate matters, establishing connectivity between hosts on two different networks may require traversing several networks in between, each of which may be of yet another type. These different networks may be Ethernets, token rings, point-to-point links, or switched networks of various flavors, and each of them is likely to have its own addressing scheme, media access protocols, service model, and so on. The challenge of heterogeneity is to provide a useful and fairly predictable host-to-host service over this hodge-podge of different networks. To understand the problems of scaling, it is worth considering the observed growth of the Internet, which has roughly doubled in size each year for 20 years. This sort of growth forces us to think about how to do routing efficiently in a network that might have billions of nodes, and this issue is closely related to how we do addressing.

This chapter looks at a series of approaches to interconnecting networks, starting with a simple approach that supports only very limited amounts of heterogeneity and scale, and ending with an introduction of a new standard for building a global internetwork. In between these two extremes, we trace the evolution of of the TCP/IP Internet in an effort to understand the problems of heterogeneity and scale in detail, along with the general techniques that can be applied to them.

The chapter begins with a description of bridging as a way of interconnecting networks. Bridges basically allow us to take a number of LANs and interconnect them to form an extended LAN---something that behaves like a LAN but does not suffer from the same limitations of physical size or number of attached hosts. As a way of interconnecting networks, however, bridging accommodates very little heterogeneity, and it does not scale to large sizes.

The next section introduces the Internet Protocol (IP), and shows how it can be used to build a scalable, heterogeneous internetwork. This section includes a discussion of the Internet's service model, which is

the key to its ability to handle heterogeneity. It also describes how the Internet's hierarchical addressing and routing scheme has allowed the Internet to scale to a modestly large size.

The following section then discusses several of the problems (growing pains) that the Internet has experienced over the past several years, and introduces the techniques that have been employed to address these problems. In particular, this section discusses the Internet's routing protocols, which are based on the principles outlined in Chapter [1](#).

The experience gained from using these techniques has led to the design of a new version of IP, which is known as Next Generation IP, or IP version 6 (IPv6). This new version of IP has been developed to allow further scaling of the Internet, and it is described in the next section. Throughout all these discussions, we see the importance of hierarchy in building scalable networks.

The chapter concludes by considering two issues related to internetworking. First, we show how multicast delivery---the ability to deliver packets efficiently to a set of receivers---can be incorporated into an internet. Second, we describe a *naming* scheme that can be used to identify objects, such as hosts, in a large internet.

Next	Up	Previous
----------------------	--------------------	--------------------------

Next: [Bridges and Extended](#) **Up:** [Internetworking](#) **Previous:** [Internetworking](#)

Bridges and Extended LANs

Suppose you have a pair of Ethernets that you want to interconnect. One thing you might do is put a repeater between them, as described in Chapter [1](#). This would not be a workable solution, however, if doing so exceeds the physical limitations of the Ethernet. (Recall that no more than two repeaters between any pair of hosts and no more than a total of 1500m in length is allowed.) An alternative would be to put a node between the two Ethernets, and have it forward frames from one Ethernet to another. This node would be in promiscuous mode, accepting all frames transmitted on either of the Ethernets, so it can forward it to the other.

The node we have just described is typically called a *bridge*, and a collection of LANs connected by one or more bridges is usually said to form an *extended LAN*. While bridges look a lot like the switches described in the last chapter, they are different in that they connect two or more multi-access networks rather than a set of point-to-point links. Bridges simply accept and forward LAN frames, without the need for a higher-level packet header that is used to make a routing decision. It is for this reason that we say a bridge is a link-level node that forwards frames, rather than a network-level node that switches packets.

While the simple 'accept and forward all frames' algorithm just described works, and corresponds to the functionality provided by early bridges, this strategy has been refined to make bridges an even more effective mechanism for interconnecting a set of LANs. The rest of this section fills in the more interesting details.

Learning Bridges

Spanning Tree Algorithm

Broadcast and Multicast

Limitations of Bridges

[Next](#) [Up](#) [Previous](#)

Next: [Simple Internetworking \(IP\)](#) **Up:** [Internetworking](#) **Previous:** [Problem: There Is](#)

Simple Internetworking (IP)

The remainder of this chapter explores some ways to go beyond the limitations of bridged networks, enabling us to build large, highly heterogeneous networks with reasonably efficient routing. We refer to such networks as internetworks. In the next three sections, we make a steady progression towards larger and larger internetworks. We start with the basic functionality of the currently deployed version of the Internet Protocol (IP), and then examine various techniques that have been developed to extend the scalability of the Internet in Section [\[\]](#). This discussion culminates in Section [\[\]](#), where we describe the ``next generation'' IP, which is also known as IP version 6. Before delving into the details of an internetworking protocol, however, let's consider more carefully what the word `internetwork' means.

What is an Internetwork?

Global Addresses

Datagram Forwarding in IP

Address Translation (ARP)

Error Reporting (ICMP)

Copyright 1996, Morgan Kaufmann Publishers

Global Internet

At this point, we have seen how to connect a heterogeneous collection of networks to create an internetwork, and how to use the simple hierarchy of the IP address to make routing in an internet somewhat scalable. We say 'somewhat' scalable because even though each router does not need to know about all the hosts connected to the internet, it does need to know about all the networks connected to the internet. Today's Internet has tens of thousands of networks connected to it. Route propagation protocols, such as those discussed in Chapter [4](#), do not scale to those kinds of numbers. This section looks at three techniques that greatly improve scalability, and which have enabled the Internet to grow as far as it has.

...text deleted...

Subnetting

Route Propagation (RIP,OSPF,BGP)

Classless Routing (CIDR)

Next Generation IP

In many respects, the motivation for Next Generation IP (IPng) is the same as the motivation for the techniques described in the last section: to deal with scaling problems caused by the Internet's massive growth. Subnetting and CIDR (and BGP-4, which help CIDR to be deployed) have helped to contain the rate at which the Internet address space is being consumed (the address depletion problem) and have also helped to control the growth of routing table information needed in the Internet's routers (the routing information problem). However, there will come a point at which these techniques are no longer adequate. In particular, it is virtually impossible to achieve 100% address utilization efficiency, so the address space will probably be exhausted well before the 4 billionth host is connected to the Internet. Even if we were able to use all 4 billion addresses, it's not too hard to imagine ways in which that number could be exhausted, such as the assignment of IP addresses to set-top boxes for cable TV or to electricity meters. All these things argue that a bigger address space than that provided by 32 bits will eventually be needed.

Historical Perspective

The IETF began looking at the problem of evolving the IP address space in 1991, and several alternatives were proposed. Since the IP address is carried in the header of every IP packet, increasing the size of the address dictates a change in the packet header. This means a new version of the Internet protocol, and as a consequence, a need for new software for every host and router in the Internet. This is clearly not a trivial matter---it is a major change that needs to be thought about very carefully.

The significance of the change to a new version of IP caused a sort of snowball effect. The general feeling was that if you are going to make a change of this magnitude, you might as well fix as many other things in IP as possible at the same time. Consequently, the IETF solicited white papers from anyone who cared to write one, asking for input on the features that might be desired for a new version of IP. In addition to the need to accommodate scalable routing and addressing, some of the things that arose as 'wish list' items for IPng were:

- support for real-time services;
- security support;
- autoconfiguration (i.e. the ability of hosts to automatically configure themselves with such information as their own IP address and domain name);
- enhanced routing functionality, including support for mobile hosts.

In addition to the 'wish list' features, one absolutely non-negotiable feature for IPng was that there must be a transition plan to move from the current version of IP (version 4) to the new version. With the Internet being so large and having no centralized control, it would be completely impossible to have a 'flag day' on which everyone shut down their hosts and routers and installed a new version of IP. Thus, there will probably be a long transition period in which some hosts and routers will run IPv4 only, some will run IPv4 and IPng, and some will run IPng only.

The IETF appointed a committee called the IPng Directorate to collect all the inputs on IPng requirements and to evaluate proposals for a protocol to become IPng. Over the life of this committee there were a number of proposals, some of which merged with other proposals, and eventually one was chosen by the Directorate to be the basis for IPng. That proposal was called SIPP (Simple Internet Protocol Plus). SIPP originally called for a doubling of the IP address size to 64 bits. When the directorate selected SIPP, they stipulated several changes, one of which was another doubling of the address to 128 bits (16 bytes). Once it was decided to assign IP version number 6 to the new protocol, it became known as IPv6. The rest of this section describes the features of IPv6. At the time of writing, most of the specifications for IPv6 are moving to 'Proposed Standard' status in the IETF.

Addresses and Routing

First and foremost, IPv6 provides a 128-bit address space, as opposed to the 32 bits of version 4. Thus, while version 4 could potentially address 4 billion nodes if address assignment efficiency reaches 100%, IPv6 could address 3.4×10^{38} , again assuming 100% efficiency. As we have seen, though, 100% efficiency in address assignment is not likely. Some analysis of other addressing schemes, such as those of the French and U.S. telephone networks, as well as that of IPv4, turned up some empirical numbers for address assignment efficiency. Based on the most pessimistic estimates of efficiency drawn from this study, the IPv6 address space is predicted to provide over 1,500 addresses per square foot of the earth's surface, which certainly seems like it should serve us well even when toasters on Venus have IP addresses.

Address Space Allocation

IPv6 addresses do not have classes, but the address space is subdivided based on prefixes just as in IPv4. Rather than specifying different address classes, the prefixes specify different uses of the IPv6 address. The current assignment of prefixes is listed in Table [1](#).

Prefix	Use
0000 0000	Reserved
0000 0001	Unassigned
0000 001	Reserved for NSAP Allocation
0000 010	Reserved for IPX Allocation
0000 011	Unassigned
0000 1	Unassigned
0001	Unassigned
001	Unassigned
010	Provider-Based Unicast Address
011	Unassigned
100	Reserved for Geographic-Based Unicast Addresses
101	Unassigned
110	Unassigned
1110	Unassigned
1111 0	Unassigned
1111 10	Unassigned
1111 110	Unassigned
1111 1110 0	Unassigned
1111 1110 10	Link Local Use Addresses
1111 1110 11	Site Local Use Addresses
1111 1111	Multicast Addresses

Table: IPv6 address assignments

This allocation of the address space turns out to be easier to explain than it looks. First, the entire functionality of IPv4's three main address classes (A, B and C) is contained inside the 010 prefix. Provider-based unicast addresses, as we will see shortly, are a lot like classless IPv4 addresses, only much longer. These are the main ones to worry about now, with one eighth of the address space allocated to this important form of address. Obviously, large chunks of address space have been left unassigned to allow for future growth and new features. Two portions of the address space (0000 001 and 0000 010) have been reserved for encoding of other (non-IP) address schemes. NSAP addresses are used by the ISO protocols, and IPX addresses are used by Novell's network layer protocol. Exactly how

these encodings will work is not yet defined.

Geographic unicast addresses are not yet defined, but the idea is that you should be able to have an address that relates to where you are physically rather than one that depends on who your network provider is. This is the flip side of the coin to provider-based addressing. The challenge is to design a geographic scheme that scales well.

The idea behind 'link local use' addresses is to enable a host to construct an address that will work on the network to which it is connected without being concerned about global uniqueness of the address. This may be useful for autoconfiguration, as we will see in the next subsection. Similarly, the 'site local use' addresses are intended to allow valid addresses to be constructed on a site that is not connected to the larger Internet; again, global uniqueness need not be an issue.

Finally, the 'multicast' address space is for multicast, thereby serving the same role as class D addresses in IPv4. Note that multicast addresses are easy to distinguish---they start with a byte of all ones.

Within the 'reserved' address space (addresses beginning with one byte of zeroes) are some important special types of address. A node may be assigned an 'IPv4-compatible IPv6 address' by zero-extending a 32 bit IPv4 address to 128 bits. A node that is only capable of understanding IPv4 can be assigned an 'IPv4-mapped IPv6 address' by prefixing the 32-bit IPv4 address with two bytes of all ones and then zero extending the result to 128 bits. These two special address types have uses in the IPv4 to IPv6 transition.

Address Notation

Just as with IPv4, there is some special notation for writing down IPv6 addresses. The standard representation is `x:x:x:x:x:x:x:x` where each 'x' is a hexadecimal representation of a 16-bit piece of the address. An example would be

```
47CD:1234:4422:AC02:0022:1234:A456:0124
```

Any IPv6 address can be written using this notation. Since there are a few special types of IPv6 address, there are some special notations that may be helpful in certain circumstances. For example, an address with a large number of contiguous zeroes can be written more compactly by omitting all the zero fields. For example

```
47CD:0000:0000:0000:0000:0000:A456:0124
```

could be written

```
47CD::A456:0124
```

Clearly, this form of shorthand can only be done for one set of contiguous zeroes in an address to avoid ambiguity.

Since there are the two types of IPv6 address which contain embedded IPv4 address, these have their own special notation which make extraction of the IPv4 address easier. For example, the 'IPv4-mapped IPv6 address' of a host whose IPv4 address was 128.96.33.81 could be written as

```
:::00FF:128.96.33.81
```


That is, the last 32 bits are written in IPv4 notation, rather than as a pair of hexadecimal numbers separated by a colon. Note that the double colon at the front indicates the leading zeroes.

Provider-Based Unicast Addresses

By far the most important thing that IPv6 must provide when it is deployed is plain old unicast addressing. It must do this in a way that supports the rapid rate of addition of new hosts to the Internet and that allows routing to be done in a scalable way as the number of physical networks in the Internet grows. Thus, at the heart of IPv6 is the unicast address allocation plan that determines how provider-based addresses---those with the 010 prefix---will be assigned to service providers, autonomous systems, networks, hosts, and routers.

In fact, the address allocation plan that is proposed for IPv6 unicast addresses is extremely similar to that being deployed with CIDR in IPv4. To understand how it works and how it provides scalability, it is helpful to define some new terms. We may think of a non-transit AS (i.e. a stub or multihomed AS) as a *subscriber* and we may think of a transit AS as a *provider*. Furthermore, we may subdivide providers into *direct* and *indirect*. The former are directly connected to subscribers, and are often referred to as regional networks. The latter primarily connect other providers, are not connected directly to subscribers, and are often known as *backbone* networks.


With this set of definitions, we can see that the Internet is not just an arbitrarily interconnected set of AS's, but has some intrinsic hierarchy. The difficulty is in making use of this hierarchy without inventing mechanisms that fail when the hierarchy is not strictly observed, as happened with EGP. For example, the distinction between direct and indirect providers becomes blurred when a subscriber connects to a backbone or a direct provider starts connecting to many other providers.

As with CIDR, the goal of the IPv6 address allocation plan is to provide aggregation of routing information to reduce the burden on intra-domain routers. Again, the key idea is to use an address prefix---a set of contiguous bits at the most significant end of the address---to aggregate reachability information to a large number of networks and even to a large number of AS's. The main way to achieve this is to assign an address prefix to a direct provider and for that direct provider to assign longer prefixes which begin with that prefix to the subscribers. This is exactly what we observed in Figure .

Thus, a provider can advertise a single prefix for all its subscribers. It should now be clear why this scheme is referred to as 'provider based addressing'. The IPv6 addresses assigned to a site under this scheme will all contain a prefix which identifies the provider for that site.

Of course, the drawback is that if a site decides to change providers, it will need to obtain a new address prefix and renumber all the nodes in the site. This could be a colossal undertaking, enough to dissuade most people from ever changing providers. It is for this reason that geographic addressing schemes are being investigated. For now, however, provider-based is necessary to make routing work efficiently.

One question is whether it makes sense for hierarchical aggregation to take place at other levels in the hierarchy. For example, should all providers obtain their address prefixes from within a prefix allocated to the backbone to which they connect? Given that most providers connect to multiple backbones, this probably doesn't make sense. Also, since the number of providers is much smaller than the number of sites, the benefits of aggregating at this level are much less.

One place where aggregation may make sense is at the national or continental level. Continental boundaries form natural divisions in the Internet topology, and if all addresses in Europe, for example, had a common prefix, then a great deal of aggregation could be done, so that most routers in other continents would only need one routing table entry for all networks with the Europe prefix. Providers in Europe would all select their prefixes such that they began with the European prefix. With this scheme, an IPv6 address might look like Figure . The RegistryID might be an identifier assigned to a European address registry, with different IDs assigned to other continents or countries. Note that prefixes would be of different lengths under this scenario. For example, a provider with few customers could have a longer prefix (and thus less total address space available) than one with many customers.

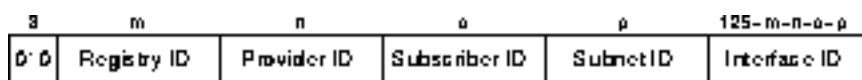


Figure: An IPv6 provider-based unicast address

One tricky situation that arises is where a subscriber is connected to more than one provider. Which prefix should the subscriber use for his site? There is no perfect solution to the problem. For example, suppose a subscriber is connected to two providers X and Y. If the subscriber takes its prefix from X, then Y has to advertise a prefix that has no relationship to its other subscribers, and as a consequence, cannot be aggregated. If the subscriber numbers part of its AS with the prefix of X and part with the prefix of Y, it runs the risk of having half his site becoming unreachable if the connection to one provider goes down. One solution that works fairly well if X and Y have a lot of subscribers in common is for them to have 3 prefixes between them: one for subscribers of X only, one for subscribers of Y only, and one for the sites which are subscribers of both X and Y.

IPv6 Features

As mentioned at the beginning of this section, the primary motivation behind the development of IPv6 is to support the continued growth of the Internet. Once the IP header had to be changed for the sake of the addresses, however, the door was open for more far reaching changes, including changes to the Internet's underlying best effort service model.

Although many of these extensions are not yet well defined, enough hooks have been put into the IPv6 packet format to support them once they are more mature. We therefore limit our discussion to an overview of the IPv6 header, and an introduction to two of features that are easy to explain given the material covered so far in this book---autoconfiguration and source-directed routing. Many of the other features that may one day end up in IPv6 are covered elsewhere in this book---multicast is discussed in Section 10.1, network security is the topic of Section 10.2, and a new service model proposed for the Internet is described in Section 10.3.

Packet Format

Despite the fact that IPv6 extends IPv4 in several ways, its header format is actually simpler. This is due to a concerted effort to remove unnecessary functionality from the protocol. Figure 10-1 shows the result.

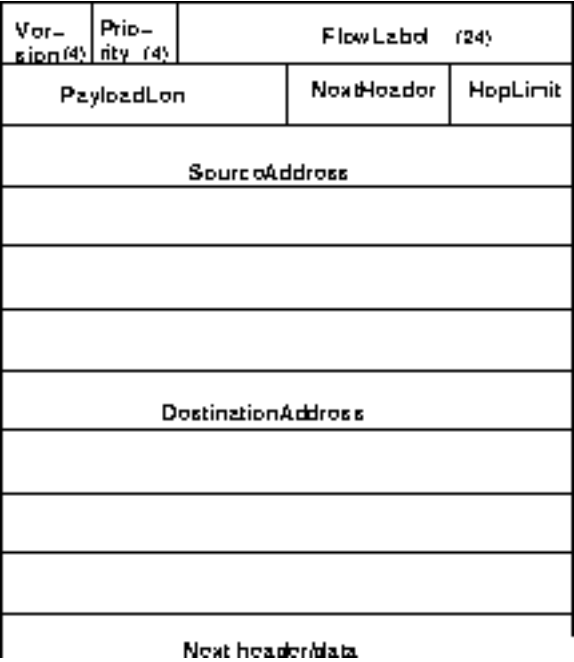


Figure: IPv6 packet header

As with many headers, this one starts with a `Version` field, which is set to 6 for IPv6. The `Version` field is in the same place relative to the start of the header as IPv4's version field so that header processing software can immediately decide which header format to look for. The `Priority` and `FlowLabel` fields both relate to quality of service issues, as discussed in Section 10.1.


The `PayloadLen` gives the length of the packet, excluding the IPv6 header, measured in bytes. The `NextHeader` field cleverly replaces both the IP options and the `Protocol` field of IPv4. If options are required, then they are carried in one or more special headers following the IP header, and this is indicated by the value of the `NextHeader` field. If there are no special headers, the `NextHeader` field is the header for the higher level protocol running over IP (e.g. TCP or UDP), that is, it serves the same purpose as the IPv4 `Protocol` field. Also, fragmentation is now handled as an optional header, which means the fragmentation-related fields of IPv4 are not included in the IPv6 header. The `HopLimit` field is simply the TTL of IPv4, renamed to reflect the way it is actually used.

Finally, the bulk of the header is taken up with the source and destination addresses, each of which is 16 bytes (128 bits) long. Thus, the IPv6 header is always 40 bytes long. Considering that IPv6 addresses are four times longer than those of IPv4, this compares quite well with the IPv4 header, which is 20 bytes long in the absence of options.

The way that IPv6 handles options is quite an improvement over IPv4. In IPv4, if any options were present, every router had to parse the entire options field to see if any of the options were relevant. This is because the options were all buried at the end of the IP header, as an unordered collection of `(type, length, value)` tuples. In contrast, IPv6 treats options as *extension headers* that must, if present, appear in a specific order. This means that each router can quickly determine if any of the options are relevant to it, which in most cases, they will not be. Usually this can be determined by just looking at the `NextHeader` field. The end result is that option processing is much more efficient in IPv6, an important factor in router performance. In addition, the new formatting of options as extension headers means that they can be of arbitrary length, whereas in IPv4 they were limited to 44 bytes at most. We will see how some of the options are used below.



Figure: IPv6 fragmentation extension header

To see how extension headers work, consider the fragmentation header shown in Figure . This header provides functionality similar to the fragmentation fields in the IPv4 header, but is only present if fragmentation is necessary. If it was the first or only extension header, then the `NextHeader` field of the IPv6 header would contain the value 44, which says 'the next header is a fragmentation header'. In this case, the `NextHeader` field of the fragmentation header contains a value describing the header that follows it. For example, if there are no more extension headers, then the next header might be the TCP header, which results in `NextHeader` containing the value 6. (Note that when the next header is a higher layer protocol, not an extension header, the value of the `NextHeader` field is the same as the value of the `ProtocolNum` field in an IPv4 header.) If the fragmentation header were followed by, say, an authentication header, then the fragmentation header's `NextHeader` field would contain the value 51.


Autoconfiguration

While the Internet's growth has been impressive, one factor that has inhibited faster acceptance of the technology is the fact that getting connected to the Internet has typically required a fair amount of system administration expertise. In particular, every host that is connected to the Internet needs to be configured with a certain minimum amount of information, such as a valid IP address, a subnet mask for the link to which it attaches, and the address of a name server. Thus, it has not been possible to unpack a new computer and connect it to the Internet without some pre-configuration. One goal of IPv6, therefore, is to provide support for autoconfiguration, sometimes referred to as 'plug-and-play' operation.

There are various aspects to autoconfiguration, but the most crucial step is *address autoconfiguration*, since a host cannot communicate with anything else until it gets an IP address. There are two proposed approaches to address autoconfiguration: a *stateful* approach, in which hosts talk to a configuration server, and a *stateless* approach, in which hosts construct their IP address essentially on their own. The latter is the one that is being standardized at the time of writing.

Recall that IPv6 unicast addresses are hierarchical, and that the least significant portion is the 'Interface ID'. Thus, we can subdivide the autoconfiguration problem into two parts:

- obtain an interface ID that is unique on the link to which the host is attached;
- obtain the correct address prefix for this subnet.

The first part turns out to be rather easy, since every host on a link must have a unique link-level address. For example, all hosts on an Ethernet have a unique 48 bit Ethernet address. This can be turned into a valid 'link local use address' by adding the appropriate prefix from Table  (1111 1110 10) followed by enough zeroes to make up 128 bits. For some devices---for example, printers or hosts on a small routerless network that do not connect to any other networks---this address may be perfectly adequate. Those devices that need a globally valid address depend on a router on the same link to periodically advertise the appropriate prefix for the link. Clearly, this requires that the router is configured with the correct address prefix, and that this prefix is chosen in such a way that there is enough space at the end to attach an appropriate link-level address (e.g. 48 bits.)

The ability to embed link level addresses as long as 48 bits into IPv6 addresses was one of the reasons for choosing such a large address size. Not only does 128 bits allow the embedding, but it leaves plenty of space for the multilevel hierarchy of addressing that we discussed above.

Advanced Routing Capabilities

Another of IPv6's extension headers is the routing header. In the absence of this header, routing for IPv6

differs very little from that of IPv4 under CIDR. The routing header contains a list of IPv6 addresses that represent nodes or topological areas that the packet should visit en-route to its destination. A topological area may be, for example, a backbone provider's network. Specifying that packets must visit this network would be a way of implementing provider selection on a packet-by-packet basis. Thus, a host could say that it wants some packets to go via a provider that is cheap, others go via a provider that provides high reliability, and still others via a provider that the host trusts to provide security.

To provide the ability to specify topological entities rather than individual nodes, IPv6 defines an *anycast* address. An anycast address is assigned to a set of interfaces, and packets sent to that address will go to the 'nearest' of those interfaces, with nearest being determined by the routing protocols. For example, all the routers of a backbone provider could be assigned a single anycast address, which would be used in the routing header.

The anycast address and the routing header are also expected to be used to provide enhanced routing support to mobile hosts. The detailed mechanisms for providing this support are still being defined.

Sidebar: Transition from IPv4 to IPv6

The most important idea behind the transition from IPv4 to IPv6 is that the Internet is far too big and decentralized to have a 'flag day'---a day on which every host and router is upgraded from IPv4 to IPv6. Thus, IPv6 needs to be deployed incrementally in such a way that hosts and routers that only understand IPv4 can continue to function for as long as possible. Ideally, IPv4 nodes should be able to talk to other IPv4 nodes and some set of other IPv6-capable nodes indefinitely. Also, IPv6 hosts should be capable of talking to other IPv6 nodes even when some of the infrastructure between them may only support IPv4. Two major mechanisms have been defined to help this transition: *dual-stack operation*, and *tunneling*.

The idea of dual stacks is fairly straightforward: IPv6 nodes run both IPv6 and IPv4 and use the version field to decide which stack should process an arriving packet. In this case, the IPv6 address could be unrelated to the IPv4 address, or it could be the 'IPv4-mapped IPv6 address' described earlier in this section.

Tunneling is a technique used in a variety of situations in which an IP packet is sent as the *payload* of an IP packet, that is, a new IP header is attached in front of the header of an IP packet. For IPv6 transition, tunneling is used to send an IPv6 packet over a piece of the network which only understands IPv4. This means that the IPv6 packet is encapsulated within an IPv4 header which has the address of the tunnel endpoint in its header, transmitted across the IPv4-only piece of network, and then decapsulated at the endpoint. The endpoint could be either a router or a host; in either case, it must be IPv6 capable to

be able to process the IPv6 packet after decapsulation. If the endpoint is a host with an IPv4-mapped IPv6 address, then tunneling can be done automatically, by extracting the IPv4 address from the IPv6 address and using it to form the IPv4 header. Otherwise, the tunnel must be configured manually. In this case, the encapsulating node needs to know the IPv4 address of the other end of the tunnel, since it cannot be extracted from the IPv6 header. From the perspective of IPv6, the other end of the tunnel looks like a regular IPv6 node that is just one hop away, even though there may be many hops of IPv4 infrastructure between the tunnel endpoints.

Tunneling is a very powerful technique with applications beyond this one. One of the main challenges of tunneling is dealing with all the special cases, such as fragmentation within the tunnel and error handling.

[Next](#) [Up](#) [Previous](#)

Next: [Multicast](#) **Up:** [Internetworking](#) **Previous:** [Global Internet](#)

Multicast

As we saw in Chapter [□](#), multi-access networks like Ethernet and FDDI implement multicast in hardware. This section describes how to extend multicast, in software, across an internetwork of such networks. The approach described in this section is based on an implementation of multicast used in the current Internet (IPv4). Multicast is also to be supported in the next generation of IP (IPv6), but first the scalability of the solution must be improved. Several proposals on how to do this are currently being considered; see the references at the end of this chapter for more information.

The motivation for multicast is that there are applications that want to send a packet to more than one destination host. Instead of forcing the source host to send a separate packet to each of the destination hosts, we want the source to send a single packet to a *multicast address*, and for the network---or internet, in this case---to deliver a copy of that packet to each of a *group* of hosts. Hosts can then choose to join or leave this group at will, without synchronizing or negotiating with other members of the group. Also, a host may belong to more than one group at a time.

Internet multicast can be implemented on top of a collection of networks that support hardware multicast (or broadcast) by extending the forwarding function implemented by the routers that connect these networks. This section describes two such extensions: one that extends the distance-vector protocol used by some routers (e.g., RIP), and one that extends the link-state protocol used by other routers (e.g., OSPF). Notice that both extensions involve adding a specially designated multicast address type to the internet address. For example, multicast addresses have been added to IPv4 by defining a new address class; IPv6 was defined from the start to support multicast addresses.

Link-State Multicast

Distance-Vector Multicast

[Next](#) [Up](#) [Previous](#)

Next: [Host Names \(DNS\)](#) **Up:** [Internetworking](#) **Previous:** [Next Generation IP](#)

Copyright 1996, Morgan Kaufmann Publishers

Host Names (DNS)

The addresses we have been using to identify hosts, while perfectly suited for processing by routers, are not exactly user-friendly. It is for this reason that a unique *name* is also typically assigned to each host in a network. Host names differ from host addresses in two important ways. First, they are usually variable length and mnemonic, thereby making them easier for humans to remember. In contrast, fixed-length numeric addresses are easier for routers to process. Second, names typically contain no information that helps the network locate (route packets towards) the host. Addresses, in contrast, sometimes have routing information embedded in them; flat addresses are the exception.

Before getting into the details of how hosts are named in a network, we first introduce some basic terminology. First a *name space* defines the set of possible names. A name space can be either *flat* (names are not divisible into components), or it can be *hierarchical* (Unix file names are the obvious example). Second, the naming system maintains a collection of *bindings* of names to values. Values can be anything we want the naming system to return when presented with a name; in many cases it is an address. Finally, a *resolution mechanism* is the procedure, that when invoked with a name, returns the corresponding value. A *name server* is a specific implementation of a resolution mechanism that is available on a network and can queried by sending it a message.

Because of its large size, the Internet has a particularly well developed naming system in place---the *domain name system* (DNS). We therefore use DNS as a framework for discussing the problem of naming hosts. Note that the Internet did not always use DNS. Early in its history, when there were only a few hundred hosts on the Internet, a central authority called the *network information center* (NIC) maintained a flat table of name-to-address bindings; this table was called `hosts.txt`. Whenever a site wanted to add a new host to the Internet, the site administrator sent email to the NIC giving the new host's name/address pair. This information was manually entered into the table, the modified table was mailed out to the various sites every few days, and the system administrator at each site installed the table on every host at the site. Name resolution was then simply implemented by a procedure that looks up a host's name in the local copy of the table, and returns the corresponding address.

It should come as no surprise that the `hosts.txt` approach to naming did not work well as the number of hosts in the Internet started to grow. Therefore, in the mid 1980's, the domain naming system was put

into place. DNS employs a hierarchical name space rather than a flat name space, and the `table' of bindings that implement this name space is partitioned into disjoint pieces and distributed throughout the Internet. These sub-tables are made available in name servers that can be queried over the network.

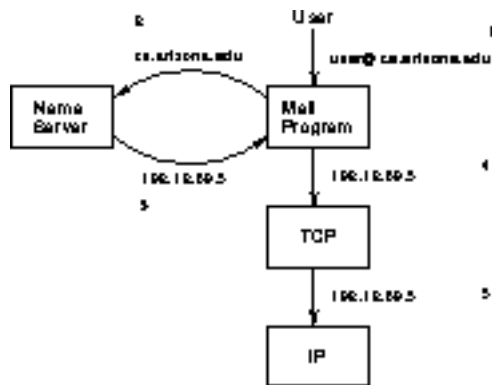


Figure: Names translated into addresses.

Note that we are jumping the gun a little bit by introducing host names at this point. What happens in the Internet is that a user presents a host name to an application program (possibly embedded in a compound name such as an email address or URL), and this program engages the naming system to translate this name into a host address. The application then opens a connection to this host by presenting some transport protocol (e.g., TCP) with the host's IP address. This situation is illustrated in the case of sending email in Figure [1](#). We discuss name resolution in this chapter, rather than some later chapter, because the main problem that the naming system must address is exactly the same problem that an internetwork itself must solve---the problem of scale.

Domain Hierarchy

Name Servers

Name Resolution

Copyright 1996, Morgan Kaufmann Publishers

Summary

The main theme of this chapter is how to build big networks by interconnecting smaller networks. We looked first at bridging, a technique that is mostly used to interconnect a small to moderate number of similar networks. What bridging does not do well is tackle the two closely-related problems of building very large networks: heterogeneity and scale. The Internet Protocol (IP) is the key tool for dealing with these problems, and provided most of the examples for this chapter.

IP tackles heterogeneity by defining a simple, common service model for an internetwork, which is based on the best-effort delivery of IP datagrams. An important part of the service model is the global addressing scheme, which enables any two nodes in an internetwork to uniquely identify each other for the purposes of exchanging data. The IP service model is simple enough to be supported by any known networking technology, and the ARP mechanism is used to translate global IP addresses into local link-layer addresses.

We then saw a succession of scaling problems and the ways that IP deals with them. The major scaling issues are the efficient use of address space and the growth of routing tables as the Internet grows. The hierarchical IP address format, with network and host parts, gives us one level of hierarchy to manage scale. Subnetting lets us make more efficient use of network numbers and helps consolidate routing information. In effect, it adds one more level of hierarchy to the address. Autonomous systems allow us to partition the routing problems into two parts, inter-domain and intra-domain routing, each of which is much smaller than the total routing problem would be. Classless routing, provided by CIDR and BGP-4, let us introduce more levels of hierarchy and to achieve further routing aggregation. These mechanisms have enabled today's Internet to sustain remarkable growth.

Eventually, all these mechanisms will be unable to keep up with the Internet's growth and a new address format will be needed. This will mean a new IP datagram format, and a new version of the protocol. Originally known as Next Generation IP (IPng), this new protocol is now known as IPv6 and will provide a 128-bit address with CIDR-like addressing and routing. At the same time, it will add considerable functionality to IP, including autoconfiguration, support for multicast, and advanced routing capabilities.

Open Issue: IP versus ATM

While we have presented IP as the only protocol for global internetworking, there are other contenders, most notably ATM. One of the design goals of ATM has, from day one, been that it could provide all sorts of services, including voice, video and data applications. Many proponents of ATM are so enamored of this planned capability that they argue that ATM is the only rational choice for future networks. They tend to refer to any network that uses variable length packets as a 'legacy' network and assume that the future of global networking lies in ATM LANs connected to ATM WANs, with all serious users having direct ATM connections into their workstations. Of course, many in the IP community have a slightly different view. They see ATM as just another technology over which one can run IP. After all, IP has run over every other technology that has come along, so why should ATM be any different? They tend to refer to ATM as 'just another subnet technology.' (One IP proponent responded to the use of the term 'legacy network' by asserting that ATM will never be successful enough even to become a legacy.) As you might expect, this does not go over well with the more strident proponents of ATM.

There are challenges for both IP and ATM if either is to succeed on a global scale. For ATM, the main obstacle is likely to be the huge installed base of non-ATM technology. It is hard to believe that all those users will want to discard their existing network adaptors, hubs, routers, wiring, and so on. For IP, one of the biggest challenges will be to provide quality of service guarantees that are suitable for high quality voice and video, something that is likely to be available in ATM networks from the outset. In a sense, ATM does not deal well with heterogeneous technologies: it works best when everyone uses ATM. IP, at present, does not deal well with a wide range of applications: it is best suited to those without real-time constraints. As we will see in Chapter [4](#), the IP community is working to address that shortcoming of IP.

In the end, the real challenge is likely to be in the integration of IP and ATM. The IETF is already working on some IP over ATM issues, having generated proposed standards for the encapsulation of IP datagrams inside ATM CS-PDUs, and for an address resolution mechanism that maps between the ATM and IP address spaces. At the time of writing, the efficient support of IP multicast over ATM networks is high on the IETF agenda. A likely challenge for the future integration of ATM and IP is the provision of end-to-end quality of service guarantees in an internetwork that includes both ATM and non-ATM

technologies.

[Next](#) [Up](#) [Previous](#)

Next: [Further Reading](#) **Up:** [Internetworking](#) **Previous:** [Summary](#)

Copyright 1996, Morgan Kaufmann Publishers

Further Reading

Not surprisingly, there have been countless papers written on various aspects of the Internet. Of these, we recommend two as must reading: the Cerf-Kahn paper is the one that originally introduced the TCP/IP architecture, and is worth reading just for its historical perspective; and the Bradner-Mankin paper gives an informative overview on how the rapidly growing Internet has stressed the scalability of the original architecture, ultimately resulting in the next generation IP. The paper by Perlman describes the spanning tree algorithm, and is the seminal paper on this topic. The next three papers discuss multicast: the first Deering paper is a seminal paper on the topic and describes the approach to multicast currently used on the MBone, while the latter two papers present more scalable solutions. Finally, the paper by Mockapetris and Dunlap describes many of the design goals behind the Domain Naming System.

- V. Cerf and R. Kahn. A Protocol for Packet Network Intercommunication. *IEEE Transactions on Communications*, COM-22(5): 637--648, May 1974.
- S. Bradner and A. Mankin. The Recommendation for the Next Generation IP Protocol. *Request for Comments 1752*, January 1995.
- R. Perlman. An Algorithm for Distributed Computation of Spanning Trees in an Extended LAN. *Proceedings of the 9th Data Communications Symposium* (Sept. 1985), 44-53.
- S. Deering and D. Cheriton. Multicast Routing in Datagram Internetworks and Extended LANs. *ACM Trans. on Comp. Syst.*, Vol. 8, No. 2 (May 1990), 85--110.
- T. Ballardie, P. Francis, and J. Crowcroft. Core Based Trees (CBT): An Architecture for Scalable Inter-Domain Multicast Routing. *Proceedings of the SIGCOMM '93 Symposium*, pages 85--95, September 1993.
- A. Thyagarajan and S. Deering. Hierarchical Distance Vector Multicast Routing for the MBone. *Proceedings of the SIGCOMM '95 Symposium*, pages 60--67, August 1995.
- P. Mockapetris and K. Dunlap. Development of the Domain Name System. *Proceedings of the SIGCOMM '88 Symposium*, pages 123--133, August 1988.

Beyond these papers, Perlman gives an excellent explanation of routing in an internet, including coverage of both bridges and routers [[Per92](#)]. The description of the spanning tree algorithm presented in Chapter 3 of Perlman's book is actually easier to understand than the paper cited in the above reading list. Also, the Lynch-Rose book gives general information on the scalability of the Internet [[Cha93](#)].

Many of the techniques and protocols developed to help the Internet scale are described in RFCs: subnetting is described in [[MP85](#)], CIDR is described in [[FLYV93](#)], RIP is defined in [[Hed88](#)] and [[Mal93](#)], OSPF is defined in [[Moy94](#)], and BGP-4 is defined in [[RL95](#)]. We observe that the OSPF specification, at 212 pages, is one of the longer RFCs around. Also, explanations of how IP and ATM can co-exist are given in [[Hei93](#),[Lau94](#),[BCDB95](#)]. A collection of RFC's related to IPv6 can be found in [[BM95](#)].

There are a wealth of papers on naming, as well as on the related issue of resource discovery---finding out what resources exist in the first place. General studies of naming can be found in [[Ter86](#),[CP89](#),[BLNS82](#),[Sal78](#),[Sho78](#),[Wat81](#)]; attribute-based (descriptive) naming systems are described in [[Pet88](#),[BPY90](#)]; and resource discovery is the subject of [[BDMS94](#)].

Finally, we recommend the following live references:

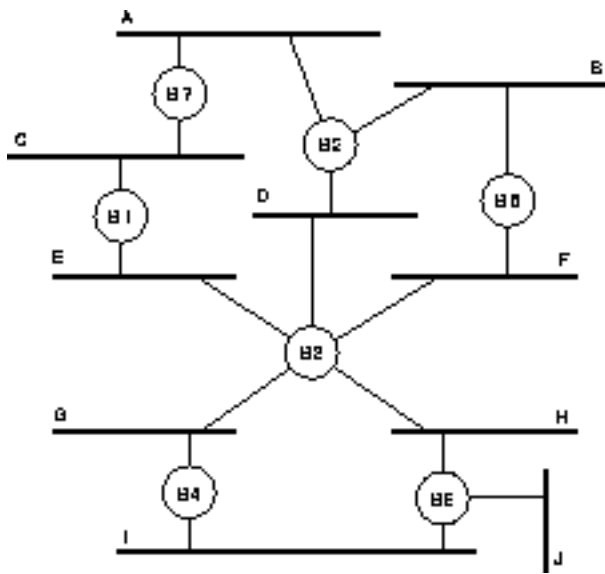
- <http://www.ietf.cnri.reston.va.us>: the IETF home page, from which you can get RFC's, internet drafts and working group charters;
- <http://playground.sun.com/pub/ipng/html/ipng-main.html>: current state of IPv6;
- <http://boom.cs.ucl.ac.uk/ietf/idmr/>: current state of the MBone;

Next	Up	Previous
----------------------	--------------------	--------------------------

Next: [Exercises](#) **Up:** [Internetworking](#) **Previous:** [Open Issue: IP](#)

Exercises

- Given the extended LAN shown below, indicate which ports are not selected by the spanning tree algorithm.



- Identify the class of each of the following IP addresses:
 - 128.36.199.3
 - 21.12.240.17
 - 183.194.76.253
 - 192.12.69.248
 - 89.3.0.1
 - 200.3.6.2
- Why does the `Offset` field in the IP header measure the offset in 8-byte units? (Hint: recall that the `Offset` field is 13 bits long.)
- Suppose a TCP message that contains 2048 bytes of data and 20 bytes of TCP header is passed to IP for delivery across two networks of the Internet (i.e., from the source host to a router to the destination host). The first network uses 14-byte headers and has a MTU of 1024 bytes; the second uses 8-byte headers with a MTU of 512 bytes. (Each network's MTU gives the total

packet size that may be sent, including the network header.) Also, recall that the IP header size is 20 bytes. Schematically depict the packets that are delivered to the network layer at the destination host.

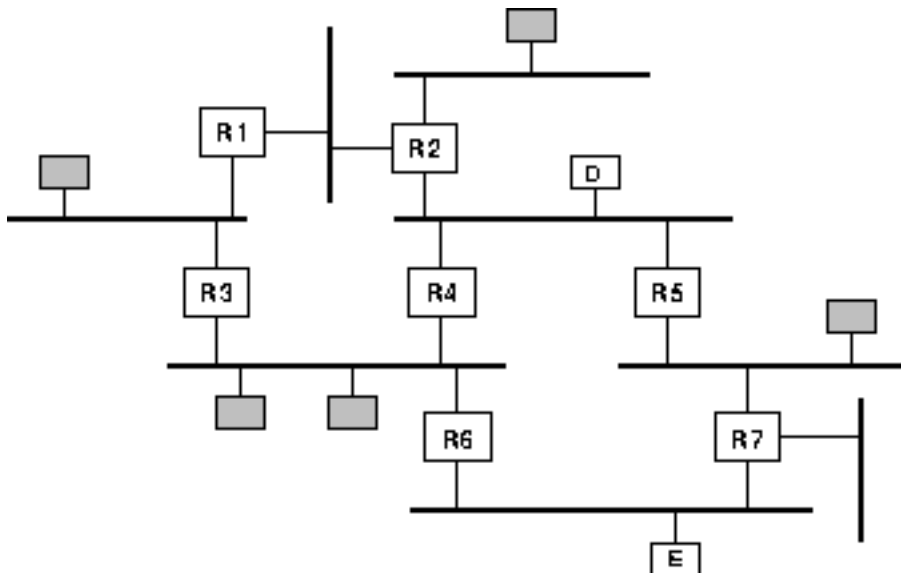
5. Give an explanation for why IP does reassembly at the destination host rather than at the routers.
6. Identify the differences in IP fragmentation/reassembly and ATM segmentation/reassembly. (Consider both AAL3/4 and AAL5.) Explain why these differences lead to different header formats.
7. In IP, why is it necessary to have an address per interface, rather than just one address per host.
8. Read the man page for the Unix utility `netstat`. Use `netstat` to display the current IP routing table on your host.
9. Use the Unix utility `ping` to determine the round-trip time to various hosts in the Internet. Read the man page for `ping` to determine how it is implemented.
10. Explain why having each entry in the ARP table timeout after 10-15 minutes is reasonable. Give the problems that occur if the timeout value is too small or too large.
11. Sketch the ARP procedure that determines if a request message just received should be used to update the ARP table. Implement the complete ARP protocol in the `x`-kernel.
12. Use the Unix utility `traceroute` to determine how many hops it is from your host to other hosts in the Internet (e.g., `cs.arizona.edu` and `thumper.bellcore.com`). How many routers do you traverse just to get out of your local site? Explain how `traceroute` is implemented. (Hint: read the man page.)
13. Suppose a router has built up the routing table given below. The router can deliver packets directly over interfaces 0 and 1, or it can forward packets to routers R2, R3 or R4. Describe what the router does with a packet addressed to each of the following destinations:

SubnetNumber	SubnetMask	NextHop
128.96.39.0	255.255.255.128	interface 0
128.96.39.128	255.255.255.128	interface 1
128.96.40.0	255.255.255.128	R2
192.4.153.0	255.255.255.192	R3
{default}		R4

1. 128.96.39.10

2. 128.96.40.12
3. 128.96.40.151
4. 192.4.153.17
5. 192.4.153.90

14. Sketch the data structures and procedures for an IP forwarding table that supports subnetting. Implement IP in the **x**-kernel.
15. This chapter claimed that IP is designed to run on top of any network technology. Explain the main complication that arises when IP is implemented on top of a connection-oriented network, such as X.25. How might the IP routers connected to such a network work around this complication?
16. What would be the disadvantage to placing the IP version number in a place other than the first byte of the header?
17. Why was the TTL field of IPv4 renamed HopLimit in IPv6?
18. Why is it unnecessary to use ARP in conjunction with IPv6?
19. Consider the example internet is shown below, in which sources D and E send packets to multicast group G, whose members are shaded in grey. Show the shortest-path multicast trees for each source.



20. Determine if your site is connected to the MBone. If so, investigate and experiment with any MBone tools, such as `sd` and `nv`.
21. The original Internet name-lookup mechanism used a central `hosts.txt` table, which was

distributed to all hosts every few days. Describe the reasons why this mechanism is no longer used.

22. When hosts maintained `hosts.txt` files, IP addresses could easily be mapped back into host names. Describe how this might be done using DNS.
23. Read the man page for `nslookup`, a Unix utility for communicating interactively with DNS servers. Determine what sequence of name servers are contacted to resolve the name `cheltenham.cs.arizona.edu`.

Next	Up	Previous
------	----	----------

Next: [End-to-End Protocols](#) **Up:** [Internetworking](#) **Previous:** [Further Reading](#)

End-to-End Protocols

- [Problem: Getting Processes to Communicate](#)
 - [Simple Demultiplexor \(UDP\)](#)
 - [Reliable Byte-Stream \(TCP\)](#)
 - [Remote Procedure Call](#)
 - [Application Programming Interface](#)
 - [Performance](#)
 - [Summary](#)
 - [Open Issue: Application-Specific Protocols](#)
 - [Further Reading](#)
 - [Exercises](#)
-

Problem: Getting Processes to Communicate

The previous three chapters have described various technologies that can be used to connect together a collection of computers: direct links (including LAN technologies like Ethernet and FDDI), packet switched networks (including cell-based networks like ATM), and internetworks. The next problem is to turn this host-to-host packet delivery service into a process-to-process communication channel. This is the role played by the *transport* level of the network architecture, which, because it supports communication between the end application programs, is sometimes called the *end-to-end* protocol.

Two forces shape the end-to-end protocol. From above, the application-level processes that use its services have certain requirements. The following lists some of the common properties that a transport protocol can be expected to provide:

- Guarantees message delivery;
- Delivers messages in the same order they are sent;
- Delivers at most one copy of each message;
- Supports arbitrarily large messages;
- Supports synchronization between the sender and the receiver;
- Allows the receiver to apply flow control to the sender;
- Supports multiple application processes on each host.

Note that this list does not include all the functionality that application processes might want from the network. For example, it does not include security, which is typically provided by protocols that sit above the transport level.

From below, the underlying network upon which the transport protocol operates has certain limitations in the level of service it provides. Some of the more typical limitations of the network are that it

- Drops messages;
- Re-orders messages;
- Delivers duplicate copies of a given message;
- Limits messages to some finite size;
- Delivers messages after an arbitrarily long delay.

Such a network is said to provide a *best-effort* level of service, as exemplified by the Internet.

The challenge, therefore, is to develop algorithms that turn the less-than-desirable properties of the underlying network into the high level of service required by application programs. Different transport protocols employ different combinations of these algorithms. This chapter looks at these algorithms in the context of three representative services---a simple asynchronous demultiplexing service, a reliable byte-stream service, and a request/reply service.

In the case of the demultiplexing and byte-stream services, we use the Internet's UDP and TCP protocols, respectively, to illustrate how these services are provided in practice. In the third case, we first give a collection of algorithms that implement the request/reply (plus other related) services, and then show how these algorithms can be combined to implement a Remote Procedure Call (RPC) protocol. This discussion is capped off with a description of a widely used RPC protocol---SunRPC---in terms of these component algorithms. Finally, this chapter concludes with a section that introduces a popular interface that application programs use to access the services of the transport protocol, and a section that discusses the performance of the different transport protocols.

Next	Up	Previous
------	----	----------

Next: [Simple Demultiplexor \(UDP\)](#) **Up:** [End-to-End Protocols](#) **Previous:** [End-to-End Protocols](#)

Simple Demultiplexor (UDP)

The simplest possible transport protocol is one that extends the host-to-host delivery service of the underlying network into a process-to-process communication service. There are likely to be many processes on any given host, so the protocol needs to add a level of demultiplexing, thereby allowing multiple application processes on each host to share the network. Aside from this, it adds no other functionality to the best-effort service provided by the underlying network. The Internet's User Datagram Protocol (UDP) is an example of such a transport protocol. So is the example ASP protocol given in Chapter [1](#).

Reliable Byte-Stream (TCP)

In contrast to a simple demultiplexing protocol like UDP, a more sophisticated transport protocol is one that offers a connection-oriented, reliable byte-stream service. Such a service has proven useful to a wide assortment of applications because it frees the application from having to worry about missing or reordered data. The Internet's Transmission Control Protocol (TCP) is probably the most widely used instance of such a protocol. It is also the most carefully optimized, which makes it an interesting protocol to study.

In terms of the properties of transport protocols given in the problem statement at the start of this Chapter, TCP guarantees the reliable, in-order delivery of a stream of bytes. It is a full-duplex protocol, meaning that each TCP connection supports a pair of byte-streams, one flowing in each direction. It also includes a flow control mechanism for each of these byte-streams that allows the receiver to limit how much data the sender can transmit. Finally, like UDP, TCP supports a demultiplexing mechanism that allows multiple application programs on any given host to simultaneously carry on a conversation with their peers.

In addition to the above features, TCP also implements a highly tuned congestion control mechanism. The idea of this mechanism is to throttle how fast TCP sends data, not for the sake of keeping the sender from over-running the *receiver*, but so as to keep the sender from overloading the *network*. A description of TCP's congestion control mechanism is postponed until Chapter [10](#), where we discuss it in the larger context of how network resources are fairly allocated.

Since many people confuse congestion control and flow control, we restate the difference. Flow control involves preventing senders from over-running the capacity of receivers. Congestion control involves preventing too much data from being injected into the network, thereby causing switches or links to become overloaded. Thus, flow control is an end-to-end issue, while congestion control is more of an issue of how hosts and networks interact.

End-to-End Issues

At the heart of TCP is the sliding window algorithm. Even though this is the same basic algorithm we saw in Section [10.1](#), because TCP runs over the Internet rather than a point-to-point link there are many important differences. This subsection identifies these differences, and explains how they complicate TCP. The following five subsections then describe how TCP addresses these complications.


First, whereas the sliding window algorithm presented in Section [10.1](#) runs over a single physical link that always connects the same two computers, TCP supports logical connections between processes running on arbitrary computers. This means that TCP needs an explicit connection establishment phase during which the two sides of the connection agree to exchange data with each other. This difference is analogous to having to "dial-up" the other party, rather than having a dedicated phone line. TCP also has an explicit connection teardown phase. One of the things that happens during connection establishment is that the two parties establish some shared state to enable the sliding window algorithm to begin.

Second, whereas a single physical link that always connects the same two computers has a fixed RTT, TCP connections have highly variable round-trip times. For example, a TCP connection between a host in Tucson and a host in New York, which are separated by several thousand kilometers, might have an RTT of 100ms, while a TCP connection between a host in Tucson and a host in Phoenix, only a few hundred kilometers away, might have an RTT of only 10ms. The same TCP protocol must be able to support both of these connections. To make matters worse, the TCP connection between hosts in Tucson and New York might have an RTT of 100ms at 3a.m., but an RTT of 500ms at 3p.m. Variations in the RTT are even possible during a single TCP connection that lasts only a few minutes. What this means to the sliding window algorithm is that the timeout mechanism that triggers retransmissions must be adaptive. (Certainly, the timeout for a point-to-point link must be a settable parameter, but it is not necessary to adapt this timer frequently.)

The third difference is also related to the variable RTT of a logical connection across the Internet, but this time is concerned with the pathological situation where a packet is delayed in the network for an extended period of time. Recall from Section [10.1](#) that the time-to-live (TTL) field of the IP header limits the number of hops that a packet can traverse. (In IPv6, the TTL field is renamed the `HopLimit` field.) TCP makes use of the fact that limiting the number of hops indirectly limits how long a packet can circulate in the Internet. Specifically, TCP assumes that each packet has a maximum lifetime of no more than 60 seconds. Keep in mind that IP does not directly enforce this 60 second value; it is simply a conservative estimate that TCP makes of how long a packet might live in the Internet. This sort of delay is simply not possible in a point-to-point link---a packet put into one end of the link must appear at the other end in an amount of time related to the speed of light. The implication of this difference is significant---TCP has to be prepared for very old packets suddenly showing up at the receiver, and potentially confusing the sliding window algorithm.

Fourth, the computers connected to a point-to-point link are generally engineered to support the link. For

example, if a link's delay \times bandwidth product is computed to be 8KB---meaning that a window size is selected so as to allow up to 8KB of data to be unacknowledged at a given time---then it is likely that the computers at either end of the link have the ability to buffer up to 8KB of data. Designing the system otherwise would be silly. On the other hand, almost any kind of computer can be connected to the Internet, making the amount of resources dedicated to any one TCP connection highly variable, especially considering that any one host can potentially support hundreds of TCP connections at the same time. This means that TCP must include a mechanism that each side uses to ``learn" what resources (e.g., how much buffer space) the other side is able to apply to the connection.

Fifth, because the transmitting side of a directly connected link cannot send any faster than the bandwidth of the link allows, and only one host is pumping data into the link, it is not possible to unknowingly congest the link. Said another way, the load on the link is visible in the form of a queue of packets on the sender. In contrast, the sending side of a TCP connection has no idea what links will be traversed to reach the destination. For example, the sending machine might be directly connected to a relatively fast Ethernet---and so, capable of sending data at a rate of 10Mbps---but somewhere out in the middle of the network, a 1.5Mbps T1 link must be traversed. What is worse, data being generated by many different sources might be trying to traverse this same slow link. This leads to the problem of network congestion. As stated before, a discussion of this topic is delayed until Chapter .


We conclude this discussion of end-to-end issues by comparing TCP's approach to providing a reliable/ordered delivery service to the approach used by X.25 networks. In TCP, the underlying IP network is assumed to be unreliable and to deliver messages out-of-order; TCP uses the sliding window algorithm on an end-to-end basis to provide reliable/ordered delivery. In contrast, X.25 networks use the sliding window protocol within the network, on a hop-by-hop basis. The assumption behind this approach is that if messages are delivered reliably and in order between each pair of nodes along the path between the source host and the destination host, then the end-to-end service also guarantees reliable/ordered delivery.

The problem with this latter approach is that a sequence of hop-by-hop guarantees do not necessarily add up to an end-to-end guarantee. First, if a heterogeneous link (say across an Ethernet) is added to one end of the path, then there is no guarantee that this hop will preserve the same service as the other hops. Second, just because the sliding window protocol guarantees that messages are delivered correctly from node A and node B, and then from node B to node C, it does not guarantee that node B behaves perfectly. For example, network nodes have been known to introduce errors into messages while transferring them from an input buffer to an output buffer. They have also been known to accidentally re-order messages. As a consequence of these small windows of vulnerability, it is still necessary to provide true end-to-end checks to guarantee reliable/ordered service, even though the lower-levels of the system also implement that functionality.

This discussion serves to illustrate one of the most important principles in system design---the end-to-end argument. In a nutshell, the end-to-end argument says that a function (in our example, providing reliable/ordered delivery), should not be provided in the lower levels of the system unless it can be

completely and correctly implemented at that level. Therefore, this rule argues in favor of the TCP/IP approach. This rule is not absolute, however. It does allow for functions to be incompletely provided at a low level as a performance optimization. This is why it is perfectly consistent with the end-to-end argument to perform error detection (e.g., CRC) on a hop-by-hop basis; detecting and retransmitting a single corrupt packet across one hop is preferred to having to retransmit an entire file end-to-end.

Segment Format

TCP is a byte-oriented protocol, which means the sender writes bytes into a TCP connection and the receiver reads bytes out of the TCP connection. Although "byte-stream" describes the service TCP offers to application processes, TCP does not, itself, transmit individual bytes over the Internet. Instead, TCP on the source host buffers enough bytes from the sending process to fill a reasonably sized packet, and then sends this packet to its peer on the destination host. TCP on the destination host then empties the contents of the packet into a receive buffer, and the receiving process reads from this buffer at its leisure. This situation is illustrated in Figure , which for simplicity, shows data flowing in only one direction. In general, remember, a single TCP connection supports byte-streams flowing in both directions.

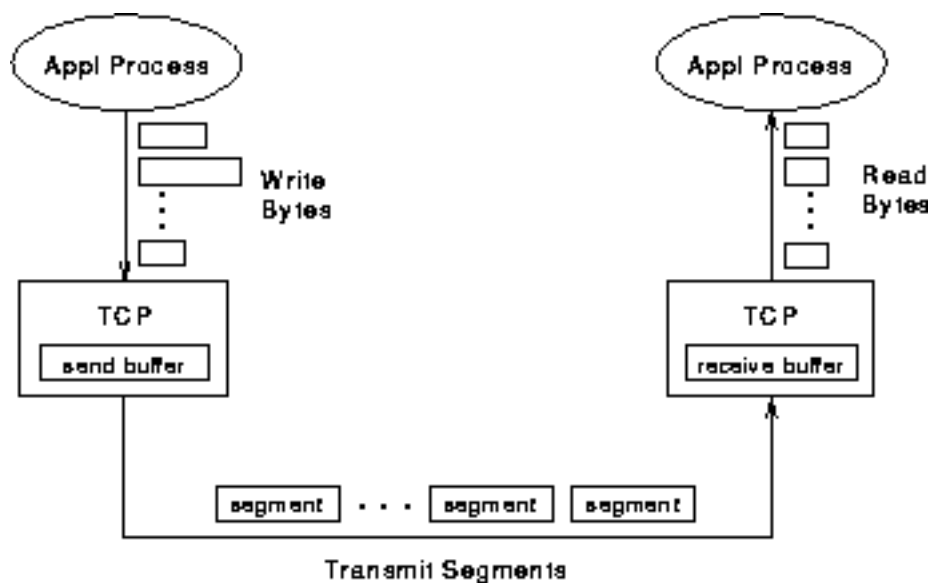




Figure: TCP managing a byte-stream.

The packets exchanged between TCP peers in Figure  are called *segments*, since each one carries a segment of the byte-stream. One question you might ask is how does TCP decide that it has enough bytes to send a segment? The answer is that TCP has three mechanisms to trigger the transmission of a segment. First, TCP maintains a threshold variable, typically called the maximum segment size (MSS), and sends a segment as soon as it has collected MSS bytes from the sending process. MSS is usually set to the size of the largest segment TCP can send without causing the local IP to fragment. That is, MSS is

set to the MTU of the directly connected network, minus the size of the TCP and IP headers. The second thing that triggers TCP to transmit a segment is that the sending process has explicitly asked it to do so. Specifically, TCP supports a *push* operation, and the sending process invokes this operation to effectively flush the buffer of unsent bytes. (This *push* operation is not the same as the the **x**-kernel's `xPush`.) This operation is used in terminal emulators like TELNET because each byte has to be sent as soon as it is typed. The final trigger for transmitting a segment is a timer that periodically fires; the resulting segment contains as many bytes as are currently buffered for transmission.

Each TCP segment contains the header schematically depicted in Figure . The relevance of most of these fields will become apparent throughout this section. For now, we simply introduce them.

Src Port		Dest Port	
SequenceNum			
Acknowledgement			
HdrLen (4)	0 (6)	Flags (6)	Advertised Window
Checksum		UrgPtr	
options (variable)			
data			

Figure: TCP header format.

The `SrcPort` and `DstPort` fields identify the source and destination ports, respectively, just as in UDP. These two fields, plus the source and destination IP addresses, combine to uniquely identify each TCP connection. That is, TCP's demux key is given by the 4-tuple:

$$\{ \text{SrcPort}, \text{SrcIPAddr}, \text{DstPort}, \text{DstIPAddr} \}.$$

Note that because TCP connections come and go, it is possible for a connection between a particular pair of ports to be established, used to send and receive data, and closed, and then at a later time, for the same pair of ports to be involved in a second connection. We sometimes refer to this situation as two different *incarnations* of the same connection.

The `Acknowledgment`, `SequenceNum`, and `AdvertisedWindow` fields are all involved in TCP's sliding window algorithm. Because TCP is a byte-oriented protocol, each byte of data has a sequence number; the `SequenceNum` field contains the sequence number for the first byte of data carried in that segment. The `Acknowledgment` and `AdvertisedWindow` fields carry information about the flow

of data going in the other direction. To simplify our discussion, we ignore the fact that data can flow in both directions, and concentrate on data with a particular `SequenceNum` flowing in one direction, and `Acknowledgment` and `AdvertisedWindow` values flowing in the opposite direction, as illustrated in Figure [1](#). The use of these three fields is described more fully in Section [2](#).

The 6-bit `Flags` field is used to relay control information between TCP peers. The possible flags include `SYN`, `FIN`, `RESET`, `PUSH`, `URG`, and `ACK`. The `SYN` and `FIN` flags are used when establishing and terminating a TCP connection, respectively. Their use is described in Section [3](#). The `ACK` flag is set any time the `Acknowledgment` field is valid, implying that the receiver should pay attention to it. The `URG` flag signifies that this segment contains urgent data. When this flag is set, the `UrgPtr` field indicates where the non-urgent data contained in this segment begins. The urgent data is contained at the front of the segment body, up to and including `UrgPtr` bytes into the segment. The `PUSH` flag signifies that the sender invoked the *push* operation, which indicates to the receiving side of TCP that it should notify the receiving process of this fact. More on these last two features in Section [4](#). Finally, the `RESET` flag signifies that the sender has become confused---for example, because it received a segment it did not expect to receive---and so wants to abort the connection.

Finally, the `Checksum` field is used in exactly the same way as in UDP---it is computed over the TCP header, the TCP data, and the pseudo-header, which is made up of the source address, destination address, and length fields from the IP header. The checksum is required for TCP in both IPv4 and IPv6. Also, since the TCP header is variable length (options can be attached after the mandatory fields), a `HdrLen` field which gives the length of the header in 32 bit words. This field is also known as the `Offset` field, since it measures the offset from the start of the packet to the start of the data.

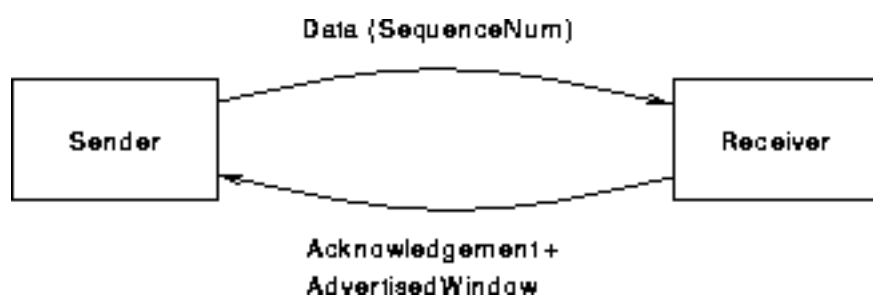



Figure: Data in one direction and ACKs in the other.

Connection Establishment and Termination

A TCP connection begins with a client (caller) doing an active open to a server (callee). Assuming the server had earlier done a passive open, the two sides engage in an exchange of messages to establish the connection. (Recall from Chapter [1](#) that a party wanting to initiate a connection performs an active open, while a party willing to accept a connection does a passive open.) Only after this connection

establishment phase is over do the two sides begin sending data. Likewise, as soon as a participant is done sending data, it closes its half of the connection, which causes TCP to initiate a round of connection termination messages. Notice that while connection setup is an asymmetric activity---one side does a passive open and the other side does an active open---connection teardown is symmetric---each side has to close the connection independently. Therefore, it is possible for one side to have done a close, meaning that it can no longer send data, but for the other side to keep its half of the bi-directional connection open and to continue sending data.

Three-Way Handshake

The algorithm used by TCP to establish and terminate a connection is called a *three-way handshake*. We first describe the basic algorithm, and then show how it is used by TCP. The three-way handshake, as the name implies, involves the exchange of three messages between the client and the server, as illustrated by the time-line given in Figure .

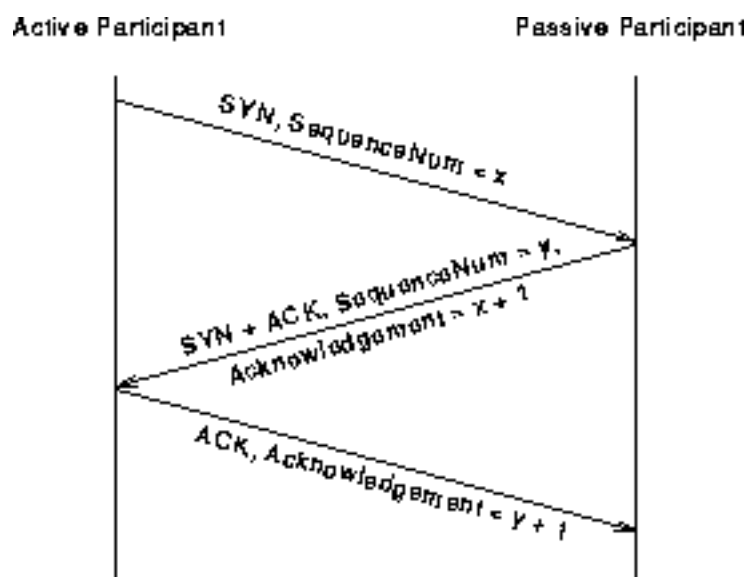


Figure: Time-line for three-way handshake algorithm.

The idea is that two parties want to agree on a set of parameters, which in the case of opening a TCP connection, are the starting sequence numbers the two sides plan to use for their respective byte-streams. In general, the parameters might be any facts that each side wants the other to know about. First, the client (the active participant) sends a segment to the server (passive participant) stating the initial sequence number it plans to use (`Flag=SYN`, `SequenceNum=x`). The server then responds with a single segment that both acknowledges the client's sequence number (`Flag=ACK`, `Ack=x+1`) and states its own beginning sequence number (`Flag=SYN`, `SequenceNum=y`). That is, both the SYN and ACK bits are set in the `Flags` field of this second message. Finally, the client responds with a third segment acknowledging the server's sequence number (`Flags=ACK`, `Ack=y+1`). The reason each side acknowledges a sequence number that is one larger than the one sent is that the `Acknowledgment` field actually identifies the "next sequence number expected", thereby implicitly acknowledging all earlier sequence numbers. Although not shown in this time-line, a timer is scheduled

for each of the first two segments, and if the expected response is not received, the segment is retransmitted.

You may be asking yourself why the client and server have to exchange starting sequence numbers with each other at connection setup time. It would be simpler if each side simply started at some "well-known" sequence number, such as zero. In fact, the TCP specification requires that each side of a connection select an initial starting sequence number *at random*. The reason for this is to protect against two incarnations of the same connection reusing the same sequence numbers too soon, that is, while there is still a chance that a segment from an earlier incarnation of a connection might interfere with a later incarnation of the connection.

State Transition Diagram

TCP is complex enough that its specification includes a state transition diagram. A copy of this diagram is given in Figure 1. This diagram shows only the states involved in opening a connection (everything above ESTABLISHED) and in closing a connection (everything below ESTABLISHED). Everything that goes on while a connection is open---i.e., the operation of the sliding window algorithm---is hidden in the ESTABLISHED state.



Figure: TCP state transition diagram.

TCP's state transition diagram is fairly easy to understand. Each circle denotes a state that any TCP connection can find itself in. All connections start in the CLOSED state. As the connection progresses, the connection moves from state to state according to the arcs. Each arc is labeled with a tag of the form *event / action*. Thus, if a connection is in the LISTEN state and a SYN segment arrives (i.e., a segment with the SYN flag set), the connection makes a transition to the SYN_RCVD state and takes the action of replying with an ACK+SYN segment.

Notice that two kinds of events trigger a state transition: (1) a segment arrives from the peer (e.g., the event on the arc from LISTEN to SYN_RCVD), and (2) the local application process invokes an operation on TCP (e.g., the *active open* event on the arc from CLOSE to SYN_SENT). In other words, TCP's state transition diagram effectively defines the *semantics* of both its *peer-to-peer* interface and its *service* interface, as defined in Section 2. The *syntax* of these two interfaces are given by the segment format (as illustrated in Figure 2), and some application programming interface (an example of which is given in Section 3), respectively.

Now let's trace the typical transitions taken through the diagram in Figure 1. Keep in mind that TCP at each end of the connection makes different transitions from state to state. When opening a connection, the server first invokes a *passive open* operation on TCP, which causes TCP to move to the LISTEN

state. At some later time, the client does an *active open*, which causes its end of the connection to send a SYN segment to the server and move to the SYN_SENT state. When the SYN segment arrives at the server, it moves to the SYN_RCVD state and responds with a SYN+ACK segment. The arrival of this segment causes the client to move to the ESTABLISHED state, and send an ACK back to the server. When this ACK arrives, the server finally moves to the ESTABLISHED state. In other words, we have just traced the three-way handshake.

There are three things to notice about the connection establishment half of the state transition diagram. First, if the client's ACK to the server is lost, corresponding to the third leg of the three-way handshake, then the connection still functions correctly. This is because the client-side is already in the ESTABLISHED state, so the local application process can start sending data to the other end. Each of these data segments will have the ACK flag set, and the correct value in the Acknowledgment field, so the server will move to the ESTABLISHED state when the first data segment arrives. This is actually an important point about TCP---every segment reports what sequence number the sender is expecting to see next, even if this repeats the same sequence number contained in one or more previous segments.

The second thing to notice about the state transition diagram is that there is a funny transition out of the LISTEN state whenever the local process invokes a *send* operation on TCP. That is, it is possible for a passive participant to identify both ends of the connection (i.e., itself and the remote participant that it is willing to have connect to it), and then change its mind about waiting for the other side and actively establish the connection. To the best of our knowledge, this is a feature of TCP that no system-specific interface allows the application process to take advantage of.

The final thing to notice about the diagram is the arcs that are not shown. Specifically, most of the states that involve sending a segment to the other side also schedule a timeout that eventually causes the segment to be resent if the expected response does not happen. These retransmissions are not depicted in the state transition diagram.

Turning our attention now to the process of terminating a connection, the important thing to keep in mind is that the application process on both sides of the connection must independently close its half of the connection. This complicates the state transition diagram because it must account for the possibility that the two sides invoke the *close* at the same time, as well as the possibility that first one side invokes *close* and then at some later time, the other side invokes *close*. Thus, on any one side there are three combinations of transitions that get a connection from the ESTABLISHED state to the CLOSED state:


- This side closes first:
ESTABLISHED \rightarrow FIN_WAIT_1 \rightarrow FIN_WAIT_2 \rightarrow TIME_WAIT \rightarrow CLOSED.
- The other side closes first:
ESTABLISHED \rightarrow CLOSE_WAIT \rightarrow LAST_ACK \rightarrow CLOSED.
- Both sides close at the same time:


ESTABLISHED → FIN_WAIT_1 → CLOSING → TIME_WAIT → CLOSED.

There is actually a fourth, although rare, sequence of transitions that lead to the CLOSED state; it follows the arc from FIN_WAIT_1 to TIME_WAIT. We leave it as an exercise for the reader to figure out what combination of circumstances leads to this fourth possibility.


The main thing to recognize about connection teardown is that a connection in the TIME_WAIT state cannot move to the CLOSED state until it has waited for two times the maximum amount of time an IP datagram might live in the Internet (i.e., 120 seconds). The reason for this is that while the local side of the connection has sent an ACK in response to the other side's FIN segment, it does not know that the ACK was successfully delivered. As a consequence, the other side might retransmit its FIN segment, and this second FIN segment might be delayed in the network. If the connection were allowed to move directly to the CLOSED state, then another pair of application processes might come along and open the same connection (i.e., use the same pair of port numbers), and the delayed FIN segment from the earlier incarnation of the connection would immediately initiate the termination of the later incarnation of that connection.


Sliding Window Revisited

We are now ready to discuss TCP's variant of the sliding window algorithm. As discussed in Section , the sliding window serves several purposes: (1) it guarantees the reliable delivery of data, (2) it ensures that data is delivered in order, and (3) it enforces flow control between the sender and the receiver.

TCP's use of the sliding window algorithm is the same as we saw in Section  in the case of the first two of these three functions. Where TCP differs from the earlier algorithm is that it folds the flow control function in as well. In particular, rather than having a fixed-sized sliding window, the receiver *advertises* a window size to the sender. This is done using the AdvertisedWindow field in the TCP header. The sender is then limited to having no more than AdvertisedWindow bytes of unacknowledged data at any given time. The receiver selects a suitable value for AdvertisedWindow based on the amount of memory allocated to the connection for the purpose of buffering data. The idea is to keep the sender from overrunning the receiver's buffer. More on this below.

Reliable and Ordered Delivery

To see how the sending and receiving sides of TCP interact with each other to implement reliable and ordered delivery, consider the situation illustrated in Figure . TCP on the sending side (pictured on the left) maintains a send buffer. This buffer is used to store data that has been sent but not yet acknowledged, as well as data that has been written by the sending application, but not transmitted. On

the receiving side (pictured on the right in Figure ) , TCP maintains a receive buffer. This buffer holds data that arrives out-of-order, as well as data that is in the correct order (i.e., there are no missing bytes earlier in the stream), but that the application process has not yet had the chance to read.

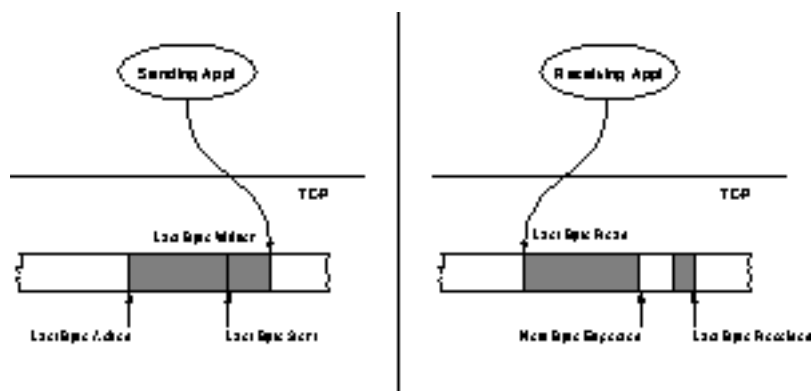


Figure: Relationship between send and receive buffers.

To make the following discussion simpler to follow, we initially ignore the fact that both the buffers and the sequence numbers are of some finite size, and so will eventually wrap around. Also, we do not distinguish between a pointer into a buffer where a particular byte of data is stored, and the sequence number for that byte.

Looking first to the sending side, three pointers are maintained into the send buffer, each with the obvious meaning: `LastByteAked`, `LastByteSent`, and `LastByteWritten`. Clearly,

$$\text{LastByteAked} \leq \text{LastByteSent}$$

since the receiver cannot have acknowledged a byte that has not yet been sent, and

$$\text{LastByteSent} \leq \text{LastByteWritten}$$

since TCP cannot send a byte that the application process has not yet written. Also note that none of the bytes to the left of `LastByteAked` need to be saved in the buffer because they have already been acknowledged, and none of the bytes to the right of `LastByteWritten` need to be buffered because they have not yet been generated.


A similar set of pointers (sequence numbers) are maintained on the receiving side: `LastByteRead`, `NextByteExpected`, and `LastByteRcvd`. The inequalities are a little less intuitive, however, because of the problem of out-of-order delivery. The first relationship

$$\text{LastByteRead} < \text{NextByteExpected}$$


is true because a byte cannot be read by the application until it is received *and* all preceding bytes have

also been received. `NextByteExpected` points to the byte immediately after the latest byte to meet this criterion. Second,

$$\text{NextByteExpected} \leq \text{LastByteRcvd} + 1$$

since, if data has arrived in order, `NextByteExpected` points to the byte after `NextByteExpected`, whereas if data has arrived out of order, `NextByteExpected` points to the start of the first gap in the data, as in Figure . Note that bytes to the left of `NextByteRead` need not be buffered because they have already been read by the local application process, and bytes to the right of `LastByteRcvd` need not be buffered because they have not yet arrived.

Flow Control

Most of the above discussion is similar to that found in Section , the only real difference being that this time we elaborated on the fact that the sending and receiving application processes are filling and emptying their local buffer, respectively. (The earlier discussion glossed over the fact that data arriving from an upstream node was filling the send buffer, and data being transmitted to a downstream node was emptying the receive buffer).

You should make sure you understand this much before proceeding, because now comes the point where the two algorithms differ more significantly. In what follows, we re-introduce the fact that both buffers are of some finite size, denoted `MaxSendBuffer` and `MaxRcvBuffer`, although we don't worry about the details of how they are implemented. In other words, we are only interested in the number of bytes being buffered, not in where those bytes are actually stored.

Recall that in a sliding window protocol, the size of the window sets the amount of data that can be sent without waiting for acknowledgment from the receiver. Thus, the receiver throttles the sender by advertising a window that is no larger than the amount of data that it can buffer. Observe that TCP on the receive side must keep

$$\text{LastByteRcvd} - \text{NextByteRead} \leq \text{MaxRcvBuffer},$$

to avoid overflowing its buffer. It therefore advertises a window size of

$$\text{AdvertisedWindow} = \text{MaxRcvBuffer} - (\text{LastByteRcvd} - \text{NextByteRead})$$

which represents the amount of free space remaining in its buffer. As data arrives, the receiver acknowledges it as long as all the preceding bytes have also arrived. In addition, `LastByteRcvd` moves to the right (is incremented), meaning that the advertised window potentially shrinks. Whether or

not it shrinks depends on how fast the local application process is consuming data. If the local process is reading data just as fast as it arrives (causing `NextByteRead` to be incremented at the same rate as `LastByteRcvd`), then the advertised window stays open (i.e., `AdvertisedWindow = MaxRcvBuffer`). If, however, the receiving process falls behind, perhaps because it performs a very expensive operation on each byte of data it reads, then the advertised window grows smaller with every segment that arrives, until it eventually goes to zero.

TCP on the send side must then adhere to the advertised window it gets from the receiver. This means that at any given time, it must ensure that

$$\text{LastByteSent} - \text{LastByteAcked} \leq \text{AdvertisedWindow}.$$

Said another way, the sender computes an *effective* window that limits how much data it can send:

$$\text{EffectiveWindow} = \text{AdvertisedWindow} - (\text{LastByteSent} - \text{LastByteAcked})$$

Clearly, `EffectiveWindow` must be greater than zero before the source can send more data. It is possible, therefore, that a segment arrives acknowledging `x` bytes, thereby allowing the sender to increment `LastByteAcked` by `x`, but because the receiving process was not reading any data, the advertised window is now `x` bytes smaller than the time before. In such a situation, the sender would be able to free buffer space, but not send any more data.

All the while this is going on, the send side must also make sure that the local application process does not overflow the send buffer, that is, that

$$\text{LastByteWritten} - \text{LastByteAcked} \leq \text{MaxSendBuffer}.$$

If the sending process tries to write `y` bytes to TCP, but

$$(\text{LastByteWritten} - \text{LastByteAcked}) + \mathbf{y} > \text{MaxSendBuffer},$$

then TCP blocks the sending process, and does not allowed it to generate more data.

It is now possible to understand how a slow receiving process ultimately stops a fast sending process. First, the receive buffer fills up, which means the advertised window shrinks to zero. An advertised window of zero means that the sending side cannot transmit any data, even though data it has previously sent has been successfully acknowledged. Finally, not being able to transmit any data means that the send buffer fills up, which ultimately causes TCP to block the sending process. As soon as the receiving process starts to read data again, the receive-side TCP is able to open its window back up, which allows the send-side TCP to transmit data out of its buffer. When this data is eventually acknowledged,

LastByteAcked is incremented, the buffer space holding this acknowledged data becomes free, and the sending process is unblocked and allowed to proceed.

There is only one remaining detail that must be resolved---how does the sending side know that the advertised window is no longer zero? As mentioned above, TCP *always* sends a segment in response to a received data segment, and this response contains the latest values for the Acknowledge and AdvertisedWindow fields, even if these values have not changed since the last time they were sent. The problem is this. Once the receive side has advertised a window size of zero, the sender is not permitted to send any more data, which means it has no way to discover that the advertised window is no longer zero at some time in the future. TCP on the receive side does not spontaneously send non-data segments; it only sends them in response to an arriving data segment.

TCP deals with this situation as follows. Whenever the other side advertises a window size of zero, the sending side persists in sending a segment with one byte of data every so often. It knows that this data will probably not be accepted, but it tries anyway, because each of these 1-byte segments triggers a response that contains the current advertised window. Eventually, one of these 1-byte probes triggers a response that reports a non-zero advertised window.

Note that the reason the sending side periodically sends this probe segment is that TCP is designed to make the receive side as simple as possible---it simply responds to segments from the sender, and never initiates any activity on its own. This is an example of a well-recognized (although not universally applied) protocol design rule, which for a lack of a better name, we call the smart sender/dumb receiver rule. Recall that we saw another example of this rule when we discussed the use of NAK's in Section



Keeping the Pipe Full

We now turn our attention to the *size* of the SequenceNum and AdvertisedWindow fields, and the implications of their sizes on TCP's correctness and performance. TCP's SequenceNum field is 32 bits long and its AdvertisedWindow field is 16 bits long, meaning that TCP has easily satisfied the requirement of the sliding window algorithm that the sequence number space be twice as big as the window size: $2^{32} \gg 2 \times 2^{16}$. However, this requirement is not the interesting thing about these two fields. Consider each field, in turn.

The relevance of the 32-bit sequence number space is that the sequence number used on a given connection might wrap around---a byte with sequence number x could be sent at one time, and then at a later time, a second byte with the same sequence number x might be sent. Once again, we assume that packets cannot survive in the Internet for more than 60 seconds. Thus, we need to make sure that the sequence number does not wrap around within a 60 second period of time. Whether or not this happens depends how fast data can be transmitted over the Internet, that is, how fast the 32-bit sequence number space can be consumed. (This discussion assumes we are trying to consume the sequence number space as fast as possible, but of course we will be if we are doing our job at keeping the pipe full.) Table



shows how long it takes for the sequence number to wrap on networks with various bandwidths.

Bandwidth	Time Until Wrap Around
T1 (1.5Mbps)	6.4 hours
Ethernet (10Mbps)	57 minutes
T3 (45Mbps)	13 minutes
FDDI (100Mbps)	6 minutes
STS-3 (155Mbps)	4 minutes
STS-12 (622Mbps)	55 seconds
STS-24 (1.2Gbps)	28 seconds

Table: Time until 32-bit sequence number space wraps around.

As you can see, the 32-bit sequence number space is adequate for today's networks, but it won't be long (STS-12) until a larger sequence number space is needed. The IETF is already working on an extension to TCP that effectively extends the sequence number space to protect against the sequence number wrapping around.

The relevance of the 16-bit advertised window fields is that it must be big enough to allow the sender to keep the pipe full. Clearly, the receiver is free to not open the window as large as the AdvertisedWindow field allows; we are interested in the situation where the receiver has enough buffer space handle as much data as the largest possible AdvertisedWindow allows.

In this case, it is not just the network bandwidth, but the delay \times bandwidth product that dictates how big the AdvertisedWindow field needs to be---the window needs to be opened far enough to allow a full delay \times bandwidth product's worth of data to be transmitted. Assuming an RTT of 100ms (a typical number for a cross-country connection in the U.S.), Table [4](#) gives the delay \times bandwidth product for several network technologies.

Bandwidth	Delay \times Bandwidth Product
T1 (1.5Mbps)	18KB
Ethernet (10Mbps)	122KB
T3 (45Mbps)	549KB
FDDI (100Mbps)	1.2MB
STS-3 (155Mbps)	1.8MB
STS-12 (622Mbps)	7.4MB
STS-24 (1.2Gbps)	14.8MB

Table: Required Window Size for 100ms RTT.

As you can see, TCP's AdvertisedWindow field is in even worse shape than its SequenceNum

field---it is not big enough to handle even a T3 connection across the continental U.S., since a 16-bit field allows us to advertise a window of only 64KB. The very same TCP extension mentioned above provides a mechanism for effectively increasing the size of the advertised window.

Adaptive Retransmission

Because TCP guarantees the reliable delivery of data, it retransmits each segment if an ACK is not received in a certain period of time. TCP sets this timeout as a function of the RTT it expects between the two ends of the connection. Unfortunately, given the range of possible RTT between any pair of hosts in the Internet, as well as the variation in RTT between the same two hosts over time, choosing an appropriate timeout value is not that easy. To address this problem, TCP uses an adaptive retransmission mechanism. We now describe this mechanism, and how it has evolved over time as the Internet community has gained more experience using TCP.

Original Algorithm

We begin with a simple algorithm for computing a timeout value between a pair of hosts. This is the algorithm originally described in the TCP specification---and the following describes it in those terms---but it could be used by any end-to-end protocol.

The idea is to keep a running average of the RTT, and then compute the timeout as a function of this RTT. Specifically, every time TCP sends a data segment, it records the time. When an ACK for that segment arrives, TCP reads the time again, and takes the difference between these two times as a `SampleRTT`. TCP then computes an `EstimatedRTT` as a weighted average of the previous estimate, and this new sample. That is,

$$\text{EstimatedRTT} = \alpha \times \text{EstimatedRTT} + \beta \times \text{SampleRTT}$$

where

$$\alpha + \beta = 1.$$

The parameters α and β are selected to *smooth* the `EstimatedRTT`. A large β tracks changes in the RTT, but is perhaps too heavily influenced by temporary fluctuations. On the other hand, a large α is more stable, but perhaps not quick enough to adapt to real changes. The original TCP specification recommended a setting of α between 0.8 and 0.9 and β between 0.1 and 0.2. TCP then uses `EstimatedRTT` to compute the timeout in a rather conservative way:

$\text{Timeout} = 2 \times \text{EstimatedRTT}$.

Karn and Partridge's Algorithm

After several years of use on the Internet, a rather obvious flaw was discovered in this simple algorithm. The problem was that an ACK does not really acknowledge a transmission, it actually acknowledges the receipt of a data. In other words, whenever a segment is retransmitted and then an ACK arrives at the sender, it is impossible to determine if this ACK should be associated with the first or the second transmission of the segment, for the purpose of measuring a sample RTT. It is necessary to know which transmission to associate it with so as to compute an accurate `SampleRTT`. As illustrated in Figure [□](#), if you assume the ACK is for the original transmission but it was really for the second, then `SampleRTT` is too large (case on left), while if you assume the ACK is for the second transmission but it is for the first, then the `SampleRTT` is too small (case on right).

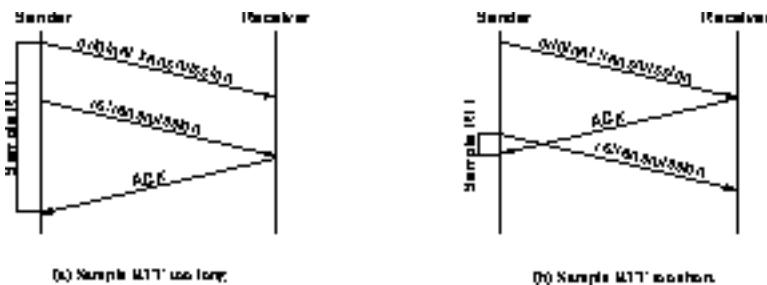


Figure: Associating the ACK with Original Transmission versus Retransmission.

The solution is surprisingly simple. Whenever TCP retransmits a segment, it stops taking samples of the RTT; it only measures `SampleRTT` for segments that have been sent only once. This fix is known as the Karn/Partridge algorithm, after its inventors. Their proposed fix also includes a second small change to TCP's timeout mechanism. Each time TCP retransmits, it sets the next timeout to be twice the last timeout, rather than basing it on the last `EstimatedRTT`. That is, Karn and Partridge proposed that TCP use exponential backoff, just as the Ethernet does.

Jacobson and Karels' New Algorithm

The Karn/Partridge algorithm was introduced at a time when the Internet was suffering from high levels of network congestion. Their approach was designed to fix some of the causes of that congestion, and although it was an improvement, the congestion was not eliminated. A couple of years later, two other researchers---Jacobson and Karels---proposed a more drastic change to TCP to battle congestion. The bulk of that proposed change is described in Chapter [□](#). Here, we focus on the aspect of that proposal related to deciding when to timeout and retransmit a segment.

As an aside, it should be clear how the timeout mechanism is related to congestion---if you timeout too soon, you may unnecessarily retransmit a segment, which only adds to the load on the network. As we will see in Chapter [□](#), the other reason for needing an accurate timeout value is that a timeout is taken

to imply congestion, which triggers a congestion control mechanism. Finally, note that there is nothing about the Jacobson/Karels' timeout computation that is specific to TCP. It could be used by any end-to-end protocol.

The main problem with the original computation is that it does not take the variance of the sample RTT's into account. Intuitively, if the variation among samples is small, then the `EstimatedRTT` can be better trusted, and there is no reason for multiplying this estimate by 2 to compute the timeout. On the other hand, a large variance in the samples suggests that the timeout value should not be too tightly coupled to the `EstimatedRTT`.

In the new approach, the sender measures a new `SampleRTT` as before. It then folds this new sample into the timeout calculation as follows:

```
Difference = SampleRTT - EstimatedRTT
EstimatedRTT = EstimatedRTT + ( $\delta \times$  Difference)
Deviation = Deviation +  $\delta (| \text{Difference} | - \text{Deviation})$ 
```

where δ is a fraction between 0 and 1. That is, we calculate both the mean RTT, and the variation in that mean.

TCP then computes the timeout value as a function of both `EstimatedRTT` and `Deviation` as follows:

```
TimeOut =  $\mu \times$  EstimatedRTT +  $\phi \times$  Deviation
```

where based on experience, μ is typically set to 1 and ϕ is set to 4. Thus, when the variance is small, `TimeOut` is close to `EstimatedRTT`, while a large variance causes the `Deviation` term to dominate the calculation.

Implementation

There are two items of note regarding the implementation of timeouts in TCP. The first is that it is possible to implement the calculation for `EstimatedRTT` and `Deviation` without using floating point arithmetic. Instead, the whole calculation is scaled by 2^n , with δ selected to be $1/2^n$. This allows us to do integer arithmetic, implementing multiplication and division using shifts, thereby achieving higher performance. The resulting calculation is given by the following code fragment, where $n=3$ (i.e., $\delta = 1/8$). Note that `EstimatedRTT` and `Deviation` are stored in their scaled up forms, while the value of `SampleRTT` at the start of the code and of `TimeOut` at the end are real, unscaled values. If you find the code hard to follow, you might want to try plugging some real numbers into it and verifying that it gives the same results as the equations above.

```

{
    SampleRTT -= (EstimatedRTT >> 3);
    EstimatedRTT += SampleRTT;
    if (SampleRTT < 0)
        SampleRTT = -SampleRTT;
    SampleRTT -= (Deviation >> 3);
    Deviation += SampleRTT;
    TimeOut = (EstimatedRTT >> 3) + (Deviation >> 1);
}

```

The second point of note is that Jacobson and Karels' algorithm is only as good as the clock used to read the current time. On a typical Berkeley Unix implementation, the clock granularity is as large as 500ms, significantly larger than the average cross-country RTT of somewhere between 100 and 200ms. To make matters worse, the Berkeley Unix implementation of TCP only checks to see if a timeout should happen every time this 500ms clock ticks, and it only takes a sample of the round-trip time once per RTT. The combination of these two factors quite often means that a timeout happens 1 second after the segment was transmitted. Once again, the proposed extensions to TCP include a mechanism that makes this RTT calculation a bit more precise.

Sidebar: TCP Extensions

We have mentioned at three different points in this section that proposed extensions to TCP help to mitigate some problem TCP is facing. These proposed extensions are designed to have as small an impact on TCP as possible. In particular, they are realized as options that can be added to the TCP header. (We glossed over this point earlier, but the reason the TCP header has a `HdrLen` field is that the header can be variable length; the variable part of the TCP header contains the options that have been added.) The significance of adding these extensions as options rather than changing the core of the TCP header is that hosts can still communicate using TCP even if they do not implement the options. Hosts that do implement the optional extensions, however, can take advantage of them. The two sides agree that they will use the options during TCP's connection establishment phase.

The first extension helps to improve TCP's timeout mechanism. Instead of measuring the RTT using a coarse-grained event, TCP can read the actual system clock when it is about to send a segment, and put this time---think of it as a 32-bit *timestamp*---in the segment's header. The receiver then echoes this timestamp back to the sender in its acknowledgment, and the sender subtracts this timestamp from the current time to measure the RTT. In essence, the timestamp option provides a convenient place for TCP to "store" the time when a segment was transmitted; it stores it in the segment itself. Note that the end-points

in the connection do not need synchronized clocks, since the timestamp is written and read at the same end of the connection.

The second extension addresses the problem of TCP's 32-bit `SequenceNum` field wrapping around too soon on a high-speed network. Rather than define a new 64-bit sequence number field, TCP uses the 32-bit timestamp just described to effectively extend the sequence number space. In other words, TCP decides whether to accept or reject a segment based on a 64-bit identifier that has the `SequenceNum` field in the low-order 32 bits and the timestamp in the high-order 32 bits. Since the timestamp is always growing, it serves to distinguish between two different incarnations of the same sequence number. Note that the timestamp is being used in this setting only to protect against wrap around; it is not treated as part of the sequence number for the purpose of ordering or acknowledging data.

The third extension allows TCP to advertise a larger window, thereby allowing it to fill larger delay \times bandwidth pipes made possible by high-speed networks. This extension involves an option that defines a *scaling factor* for the advertised window. That is, rather than interpreting the number that appears in the `AdvertisedWindow` field as indicating how many *bytes* the sender is allowed to have unacknowledged, this option allows the two sides of TCP to agree that the `AdvertisedWindow` field counts larger chunks (e.g, how many 16-byte units of data the sender can have unacknowledged). In other words, the window scaling option specifies how many bits each side should left-shift the `AdvertisedWindow` field before using its contents to compute an effective window.

Record Boundaries

As mentioned earlier in this section, TCP is a byte-stream protocol. This means that the number of bytes written by the sender is not necessarily the same as the number of bytes read by the receiver. For example, the application might write 8 bytes, then 2 bytes, then 20 bytes to a TCP connection, while on the receiving side, the application reads 5 bytes at a time inside a loop that iterates 6 times. TCP does not interject record boundaries between the 8th and 9th bytes, nor between the 10th and 11th bytes. This is in contrast to a message-oriented protocol, such as UDP, in which the message that is sent is exactly the same length as the message that is received.

Even though TCP is a byte-stream protocol, it has two different features that can be used by the sender to effectively insert record boundaries into this byte-stream, thereby informing the receiver how to break the stream of bytes into records. (Being able to mark record boundaries is useful, for example, in many database applications.) Both of these features were originally included in TCP for completely different

reasons; they have only become used for this purpose over time.

The first mechanism is the *push* operation. Originally, this mechanism was designed to allow the sending process to tell TCP that it should send whatever bytes it has collected to its peer. This was, and still is, used in terminal emulators like TELNET because each byte has to be sent as soon as it is typed. However, *push* can be used to implement record boundaries because the specification says that TCP should inform the receiving application that a *push* was performed; this is the reason for the PUSH flag in the TCP header. This act of informing the receiver of a *push* can be interpreted as marking a record boundary.

The second mechanism for inserting end-of-record markers into a byte stream is the urgent data feature, as implemented by the URG flag and the UrgPtr field in the TCP header. Originally, the urgent data mechanism was designed to allow the sending application to send *out-of-band* data to its peer. By "out-of-band" we mean data that is separate from the normal flow of data, e.g., a command to interrupt an operation already underway. This out-of-band data was identified in the segment using the UrgPtr field, and was to be delivered to the receiving process as soon as it arrived, even if that meant delivering it before data with an earlier sequence number. Over time, however, this feature has not been used, so instead of signifying "urgent" data, it has come to be used to signify "special" data, such as a record marker. This is because as with the *push* operation, TCP on the receiving side must inform the application that "urgent data" has arrived. That is, the urgent data in itself is not important. It is the fact that the sending process can effectively send a "signal" to the receiver that is important.

Of course, the application program is always free to insert record boundaries without any assistance from TCP. For example, it can send a field that indicates the length of a record that is to follow, or it can insert its own record boundary markers into the stream.

Next	Up	Previous
------	----	----------

Next: [Remote Procedure Call](#) **Up:** [End-to-End Protocols](#) **Previous:** [Simple Demultiplexor \(UDP\)](#)

Remote Procedure Call

As discussed in Chapter [1](#), a common pattern of communication used by application programs is the request/reply paradigm, also called message transaction: a client sends a request message to a server, the server responds with a reply message, and the client blocks (suspends execution) waiting for this response. Figure [1](#) illustrates the basic interaction between the client and server in such a message transaction. A transport protocol that supports the request/reply paradigm is much more than a UDP message going in one direction, followed by a UDP message going in the other direction. It also involves overcoming all of the limitations of the underlying network outlined in the problem statement at the beginning of this Chapter. While TCP overcomes these limitations by providing a reliable byte stream channel, it provides very different semantics from the request/reply abstraction that many applications need. This section looks at how we provide such an abstraction.

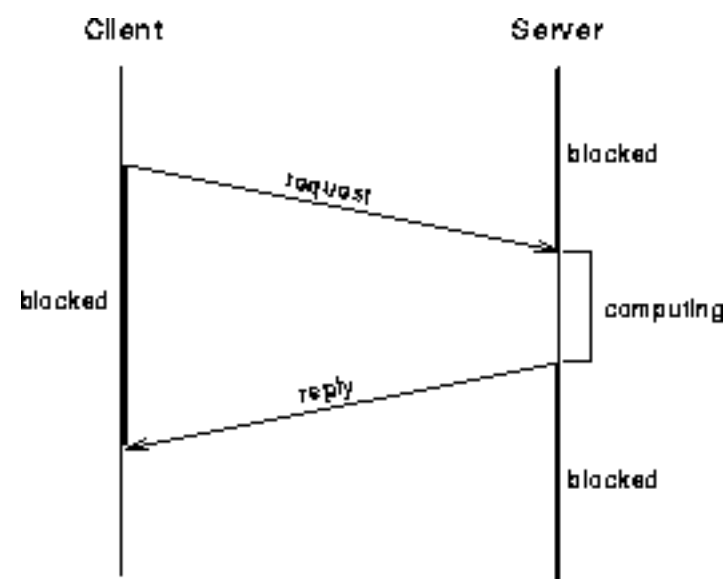


Figure: Timeline for RPC.

The request/reply communication paradigm is at the heart of a Remote Procedure Call (RPC) mechanism. RPC is a popular mechanism for structuring distributed systems because it is based on the

semantics of a local procedure call---the application program makes a call into a procedure without regard for whether it is local or remote, and blocks until the call returns. A complete RPC mechanism actually involves two major components:

- a protocol that manages the messages sent between the client and the server processes and deals with the potentially undesirable properties of the underlying network;
- programming language and compiler support to package the arguments into a request message on the client machine and then translate this message back into the arguments on the server machine (and likewise with the return value). This piece of the RPC mechanism is usually called a *stub compiler*.

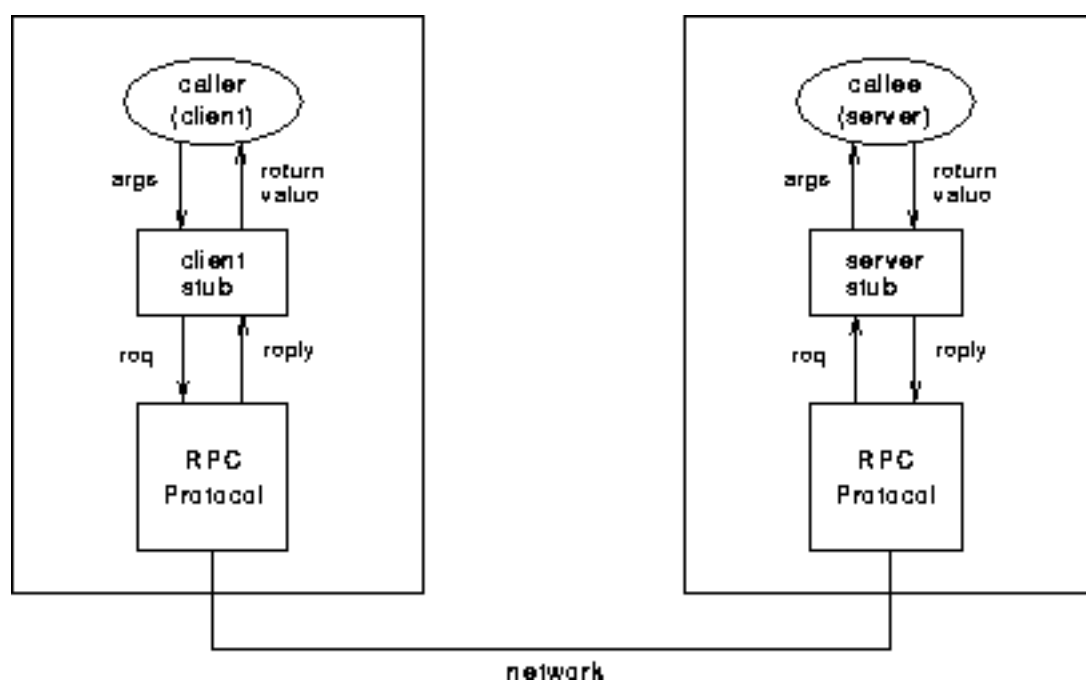





Figure: Complete RPC mechanism.

Figure  schematically depicts what happens when a client invokes a remote procedure. First, the client calls a local stub for the procedure, passing it the arguments required by the procedure. This stub hides the fact that the procedure is remote by translating the arguments into a request message, and then invoking an RPC protocol to send the request message to the server machine. At the server, the RPC protocol delivers the request message to the server stub, which translates it into the arguments to the procedure, and then calls the local procedure. After the server procedure completes, it returns the answer to the server stub, which packages this return value in a reply message that it hands off to the RPC protocol for transmission back to the client. The RPC protocol on the client passes this message up to the client stub, which translates it into a return value, and returns to the client program.

This section considers just the protocol-related aspects of an RPC mechanism. That is, it ignores the stubs, and focuses instead on the RPC protocol that transmits messages between the client and server; the transformation of arguments into messages, and vice versa, is covered in the Chapter . An RPC

protocol, as we will see, fulfills a different set of needs than either TCP or UDP, and performs a rather complicated set of functions. Indeed, even though we call this an RPC ``protocol" in the above discussion, the task being performed is complicated enough that instead of treating RPC as a single/monolithic protocol, we develop it as a ``stack" of three smaller protocols: BLAST, CHAN, and SELECT. Each of these smaller protocols, which we sometimes call a *micro-protocol*, contains a single algorithm that addresses one of the problems outlined at the start of this Chapter. In way of a brief overview:

- BLAST: fragments and reassembles large messages;
- CHAN: synchronizes request and reply messages;
- SELECT: dispatches request messages to the correct process.

These micro-protocols are complete, self-contained protocols that can be used in different combinations to provide different end-to-end services. Section  shows how they can be combined to implement RPC.

Note that BLAST, CHAN, and SELECT are not standard protocols in the sense that TCP, UDP, and IP are. They are simply protocols of our own invention, but ones that demonstrate the various algorithms needed to implement RPC. They also serve to illustrate the protocol design process, and in particular, a design philosophy in which each protocol is limited to doing just one thing and then composed with other one-function protocols to provide more complex communication services. This is in contrast to designing protocols that attempt to provide a wide-range of services to many different applications. Because this section is not constrained by the artifacts of what has been designed in the past, it provides a particularly good opportunity to examine the principles of protocol design.

Sidebar: What Layer is RPC?

Once again the ``what layer is this" issue raises its ugly head. To many people, especially those that adhere to the Internet architecture, RPC is implemented on top of a transport protocol (usually UDP), and so cannot (by definition), itself be a transport protocol. It is equally valid, however, to argue that the Internet should have an RPC protocol, since it offers a process-to-process service that is fundamentally different from that offered by TCP and UDP. The usual response to such a suggestion, however, is that the Internet architecture does not prohibit someone from implementing their own RPC protocol on top of UDP. (In general, UDP is viewed as the Internet architecture's ``escape hatch", since it effectively just adds a layer of demultiplexing to IP.) Which ever side of the ``the Internet should have an official RPC protocol" issue you come down on, the important point is that the way one implements RPC in the Internet architecture says nothing about whether RPC should be considered a transport protocol, or not.

Interestingly, there are other people that believe RPC is the most interesting protocol in the world, and that TCP/IP is just what you do when you want to go ``off site". This is the predominate view of the operating systems community, which has built countless OS kernels for distributed systems that contain exactly one protocol---you guessed it, RPC---running on top of a network device driver.

The water gets even muddier when you implement RPC as a combination of three different micro-protocols, as is the case in this section. In such a situation, which of the three is the ``transport" protocol? Our answer to this question is that any protocol that offers a process-to-process service, as opposed to a node-to-node or host-to-host service, qualifies as a transport protocol. Thus, RPC is a transport protocol, and in fact, can be implemented from a combination of micro-protocols that are themselves valid transport protocols.

Bulk Transfer (BLAST)

Request/Reply (CHAN)

Dispatcher (SELECT)

Putting it All Together (SunRPC)

Next	Up	Previous
----------------------	--------------------	--------------------------

Next: [Application Programming Interface](#) **Up:** [End-to-End Protocols](#) **Previous:** [Reliable Byte-Stream \(TCP\)](#)

Copyright 1996, Morgan Kaufmann Publishers

Application Programming Interface

We now turn our attention from the *implementation* of various end-to-end protocols, to the *interface* these protocols provide to application programs. This has not been a serious issue for the protocols covered in earlier chapters because they only interface with other protocols, and this interaction is typically buried deep in the operating system kernel. Transport protocols, however, are often directly available to application programs, and as a result, special attention is paid to how application programs gain access to the communication services they provide. This interface is generically called the *application programming interface*, or API.

Performance

Recall that Chapter [□](#) introduced the two quantitative metrics by which network performance is evaluated: latency and throughput. As mentioned in that discussion, these metrics are influenced not only by the underlying hardware (e.g., propagation delay and link bandwidth), but also by software overheads. Now that we have a complete software-based protocol graph available to us, including alternative transport protocols, we can discuss how to meaningfully measure its performance; a sort of protocol trackmeet. The importance of such measurements is that they represent the performance seen by application programs.

Experimental Method

Latency

Throughput

Summary

This chapter has described three very different end-to-end protocols, discussed the API by which application programs engage these protocols, and reported on their performance.

The first protocol we considered was a simple demultiplexor, as typified by UDP. All such a protocol does is dispatch messages to the appropriate application process based on a port number. It does not enhance the best-effort service model of the underlying network in any way, or said another way, it offers an unreliable, connectionless, datagram service to application programs.

The second is a reliable byte-stream protocol, and in particular, we looked at the details of TCP. The challenges with such a protocol are to recover from messages that may be lost by the network, to deliver messages in the same order in which they are sent, and to allow the receiver to do flow control on the sender. TCP uses the basic sliding window algorithm, enhanced with an advertised window, to implement this functionality. The other item of note for this protocol is the importance of an accurate timeout/retransmission mechanism.

The third transport protocol we looked at was a request/reply protocol that forms the basis for RPC. In this case, a combination of three different algorithms are employed to implement the request/reply service: a selective retransmission algorithm that is used to fragment and reassemble large messages, a synchronous channel algorithm that pairs the request message with the reply message, and a dispatch algorithm that causes the correct remote procedure to be invoked.

Open Issue: Application-Specific Protocols

What should be clear after reading this Chapter is that transport protocol design is a tricky business. As we have seen, getting a transport protocol right in the first place is hard enough, but changing circumstances only complicate matters. The challenge is finding ways to adapt to this change.

One thing that can change is our experience using the protocol. As we saw with TCP's timeout mechanism, experience led to a series of refinements in how TCP decides to retransmit a segment. None of these changes affected the format of the TCP header, however, and so they could be incorporated into TCP one implementation at a time. That is, there was no need for everyone to upgrade their version of TCP on the same day.

Another thing that can change is the characteristics of the underlying network. For many years, TCP's 32-bit sequence number and a 16-bit advertised window were more than adequate. Recently, however, higher bandwidth networks have meant that the sequence number is not large enough to protect against wrap around, and the advertised window is too small to allow the sender to fill the network pipe. While an obvious solution would have been to redefine the TCP header to include a 64-bit sequence number field and a 32-bit advertised window field, this would have introduced the very serious problem of how 5 million Internet hosts would make the transition from the current header to this new header. While such transitions have been performed on production networks, including the telephone network, they are no trivial matter. It was decided, therefore, to implement the necessary extensions as options, and to allow hosts to negotiate with each other as to whether or not they will use the options for each connection.

This approach will not work indefinitely, however, since the TCP header has room for only 44 bytes of options. (This is because the `HdrLen` field is 4 bits long, meaning that the total TCP header length cannot exceed 16×32 -bit words, or 64 bytes.) Of course, a TCP option that extends the space available for options is always a possibility, but one has to wonder how far it is worth going for the sake of backwards compatibility.

Perhaps the hardest changes to accommodate are those to the level of service required by application programs. It is inevitable that some application will have a good reason for wanting a slight variation from the standard services. For example, some applications want RPC most of the time, but occasionally want to be able to send a stream of request messages without waiting for any of the replies. While this is

no longer technically the semantics of RPC, a common scenario is to modify an existing RPC protocol to allow this flexibility. As another example, because video is a stream-oriented application, it is tempting to use TCP as the transport protocol. Unfortunately, TCP guarantees reliability, which is not important to the video application. In fact, a video application would rather drop a frame (segment) than wait for it to be retransmitted. Rather than invent a new transport protocol from scratch, however, some have proposed that TCP support an option that effectively turns off its reliability feature. It seems that such a protocol could hardly be called TCP any more, but we are talking about the pragmatics of getting an application to run.

How to develop transport protocols that can evolve to satisfy diverse applications, many of which have not yet been imagined, is a hard problem. It is possible that the ultimate answer to this problem is the one-function-per-protocol style promoted by the *x*-kernel, or some similar mechanism by which the application programmer is allowed to program, configure, or otherwise stylize the transport protocol.

Next	Up	Previous
----------------------	--------------------	--------------------------

Next: [Further Reading](#) **Up:** [End-to-End Protocols](#) **Previous:** [Summary](#)

Further Reading

There is no doubt that TCP is a complex protocol, and in fact, has subtleties not illuminated in this Chapter. Therefore, the recommended reading list for this Chapter includes the original TCP specification. Our motivation for including this specification is not so much to fill in the missing details, but to expose the reader to what an honest-to-goodness protocol specification looks like. The other two papers in the recommended reading list focus on RPC. The paper by Birrell and Nelson is the seminal paper on the topic, while the O'Malley-Peterson paper describes the *x*-kernel's one-function-per-protocol design philosophy in more detail.

- USC-ISI. Transmission Control Protocol. *Request for Comments 793*, September 1981.
- A. Birrell and B. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1): 39--59, February 1984.
- S. O'Malley and L. Peterson. A Dynamic Network Architecture. *ACM Transactions on Computer Systems*, 10(2): 110--143, May 1992.

Beyond the protocol specification, the most complete description of TCP, including its implementation in Berkeley Unix, can be found in [\[Ste94b\]](#). Also, the third volume of Comer's TCP/IP series of books describes how to write client/server applications on top of TCP and UDP using both the socket interface [\[CS93\]](#), and the System V Unix TLI interface [\[CS94\]](#).

Several papers evaluate the performance of different transport protocols at a very detailed level. For example, [\[CJRS89\]](#) measures the processing overheads of TCP, while [\[TL93\]](#) and [\[SB89\]](#) examine RPC's performance in great detail.

The original TCP timeout calculation was described in the TCP specification (see above), while the Karn/Partridge extension was described in [\[KP91\]](#) and the Jacobson/Karels algorithm was proposed in [\[Jac88\]](#). The TCP extensions are defined in [\[JBB92\]](#), while [\[OP91\]](#) argues that extending TCP in this way is not the right approach to solving the problem.

Finally, there are several example distributed operating systems that have defined their own RPC

protocol. Notable examples include the V system [[CZ85](#)], Sprite [[OCD88](#)], and Amoeba [[Mul90](#)]. The latest version of SunRPC, which is also known as ONC, is defined in [[Sri95a](#)].

Next	Up	Previous
----------------------	--------------------	--------------------------

Next: [Exercises](#) **Up:** [End-to-End Protocols](#) **Previous:** [Open Issue: Application-Specific](#)

[Next](#) [Up](#) [Previous](#)

Next: [End-to-End Data](#) **Up:** [End-to-End Protocols](#) **Previous:** [Further Reading](#)

Exercises

Copyright 1996, Morgan Kaufmann Publishers

End-to-End Data

-
- [Problem: What do we do with the data?](#)
 - [Presentation Formatting](#)
 - [Data Compression](#)
 - [Security](#)
 - [Summary](#)
 - [Open Issue: Presentation---Layer of the 90's](#)
 - [Further Reading](#)
 - [Exercises](#)
-

Problem: What do we do with the data?

Our focus up to this point has been on the hierarchy of protocols that deliver messages from one process to another. What we sometimes forget is that these messages carry data---information that is somehow meaningful to the application processes using the network. While this data is by nature application-specific, there are certain common transformations that application programs apply to their data before they send it. This chapter considers three such transformations: *presentation formatting*, *compression*, and *encryption*. These transformations are sometimes called *data manipulations*.

Presentation formatting has to do with translating data from the representation used internally by the application program into a representation that is suitable for transmission over a network. It is concerned with how different data types (e.g., integers, floating point numbers, character strings) are encoded in messages, and the fact that different machines typically use different encodings. It also must deal with the representation of complex data structures, which may include pointers that have only local significance, in a way that they can be transmitted to a remote machine.

Compression is concerned with reducing the number of bits it takes to represent a particular piece of data, with the goal of being able to transmit the data using as little network bandwidth as possible. Data is typically compressed before it is given to the transport protocol on the sending side, and then decompressed on the receiving side before the application begins processing it.

The third transformation, encryption, is used to ensure that anyone that might be eavesdropping on the network---that is, snoop packets as they fly by on a shared access network or as they sit in a router queue---is not able to read the data in the message. Encryption is also used to authenticate participants and to ensure message integrity.

One of the important aspects of the data manipulations studied in this chapter is that they involve processing every byte of data in the message. This is in contrast to most of the protocols we have seen up to this point, which process a message without ever looking at its contents. Because of this need to read, compute on, and write every byte of data in a message, data manipulations strongly affect the end-to-end throughput one can achieve over a network. In fact, they are often the limiting factor.

Note that there is a fourth common function that processes every byte of data in a message---error

checking. We considered error checking in an earlier Chapter (Section [□](#)) because it is usually implemented at lower levels of the protocol hierarchy; it is typically not implemented above the end-to-end protocol.

[Next](#) [Up](#) [Previous](#)

Next: [Presentation Formatting](#) **Up:** [End-to-End Data](#) **Previous:** [End-to-End Data](#)

Presentation Formatting

From the network's perspective, application programs send messages to each other. From the application's perspective, however, these messages contain various kinds of *data*---arrays of integers, video frames, lines of text, digital images, and so on. For some of this data, well established formats have been defined. For example, video is typically transmitted in MPEG format, still images are usually transmitted in JPEG or GIF format, multimedia documents are transmitted in HTML or MIME format, and text is typically transmitted in ASCII format. ☒ For other types of data, however, there is no universal agreement about the format; each computer defines its own representation. This is certainly true for the kinds of data that regular programs compute on, things like integers, floating point numbers, character strings, arrays, and structures.

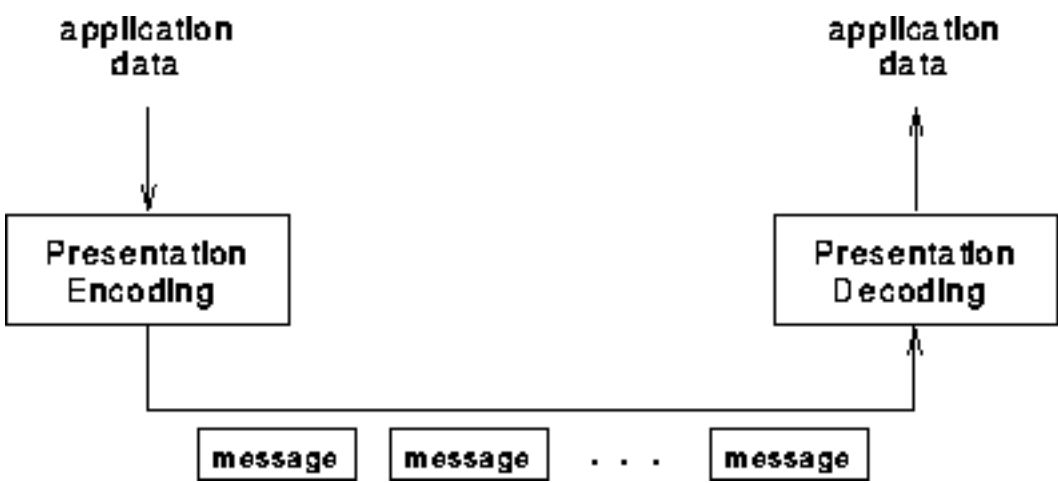


Figure: Presentation formatting involves encoding and decoding application data.

One of the most common transformations of network data is from the representation used by the application program into a form that is suitable for transmission over a network, and vice versa. This transformation is typically called *presentation formatting*. As illustrated in Figure ☐, the sending program translates the data it wants to transmit from the representation that it uses internally, into a message that can be transmitted over the network; that is, the data is *encoded* in a message. On the receiving side, the application translates this arriving message into a representation that it can then

[Next](#) [Up](#) [Previous](#)

Next: [Data Compression](#) **Up:** [End-to-End Data](#) **Previous:** [Problem: What do](#)

Data Compression

It is sometimes the case that application programs need to send more data than the bandwidth of the network supports in a timely fashion. For example, a video application might have a 10Mbps video stream that it wants to transmit, but has only a 1Mbps network available to it. In cases like this, the data can first be *compressed* at the sender, then transmitted over the network, and finally *decompressed* at the receiver.

The field of data compression has a rich history, dating back to Shannon's pioneer work on Information Theory in the 1940s. If you think of Information Theory as the study of techniques for encoding data, then compression is about the efficiency of those encodings---how few bits are needed. In a nutshell, data compression is concerned with removing *redundancy* from the encoding.

In many ways, compression is inseparable from data encoding. That is, in thinking about how to encode a piece of data in a set of bits, one may just as well think about how to encode the data in the fewest set of bits possible. For example, if you have a block of data that is made up of the symbols A through Z, and if all of these symbols have an equal chance of occurring in the data block you are encoding, then encoding each symbol in 5 bits is the best you can do. If, however, the symbol R occurs 50% of the time, then it would be a good idea to use less bits to encode the R than any of the other symbols. In general, if you know the relative probability that each symbol will occur in the data, then you can assign a different number of bits to each possible symbol in a way that minimizes the number of bits it takes to encode a given block of data. This is the essential idea of *Huffman codes*, which is one of the important early results in data compression.

For our purposes, there are two classes of compression algorithms. The first, called *loss-less compression*, ensures that the data recovered from the compression/decompression process is exactly the same as the original data. A loss-less compression algorithm is used to compress file data, such as executable code, text files, and numeric data. This is because programs that process such file data cannot tolerate mistakes in the data. In contrast, *lossy compression* does not promise that the data received is exactly the same as the data sent. This is because a lossy algorithm removes information that it cannot later restore. Hopefully, however, the lost information will not be missed by the receiver. Lossy algorithms are used to compress digital imagery, including video. This makes sense because such data

often contains more information than the human eye can perceive, and for that matter, may already contains errors and imperfections that the human eye is able to compensate for. It is also the case that lossy algorithms typically achieve much better compression ratios than do their loss-less counterparts; they can be as much as an order of magnitude better.

...text deleted...

Loss-Less Compression Algorithms

Image Compression (JPEG)

Video Compression (MPEG)

Next	Up	Previous
------	----	----------

Next: [Security](#) **Up:** [End-to-End Data](#) **Previous:** [Presentation Formatting](#)

Security

Computer networks are typically a shared resource used by many applications for many different purposes. Sometimes the data transmitted between application processes is confidential, and the applications would prefer that others not be able to read it. For example, when purchasing a product over the World-Wide-Web, users sometimes transmit their credit card numbers over the network. This is a dangerous thing to do since it is easy for someone to eavesdrop on the network and read all the packets that fly by. Therefore, a third transformation made to data sent over an end-to-end channel is to *encrypt* it, with the goal of keeping anyone that is eavesdropping on the channel from being able to read the contents of the message.

The idea of encryption is simple enough: the sender applies an *encryption* function to the original *plaintext* message, the resulting *ciphertext* message is sent over the network, and the receiver applies a reverse function (called *decryption*) to recover the original plaintext. The encryption/decryption process generally depends on a secret *key* shared between the sender and receiver. When a suitable combination of key and encryption algorithm is used, it is sufficiently difficult for an eavesdropper to break the ciphertext, and so the sender and receiver can rest assured that their communication is secure.

This familiar use of cryptography is said to ensure *privacy*---preventing the unauthorized release of information. Privacy, however, is not the only service that cryptography provides. It can also be used to support other equally important services, including *authentication* (verifying the identity of the remote participant), and message *integrity* (making sure the message has not been altered). This section first introduces the basic idea of cryptography---including a description of the two most common encryption algorithms, DES and RSA---and then shows how these algorithms can be used to provide authentication and integrity services.

One thing to keep in mind while reading this section is that the various algorithms and protocols for privacy, authentication, and integrity are being described in isolation. In practice, constructing a secure system requires an intricate combination of just the right set of protocols and algorithms. This is a challenging task because each protocol is vulnerable to a different set of attacks. To make matters worse, determining when a security protocol is ``good enough" is as much art and politics as science. A thorough analysis of these different attacks, and how one might build a complete system that minimizes the risk of compromise, is beyond the scope of this book.

Secret Key versus Public Key Cryptography



Broadly speaking, there are two types of cryptographic algorithms: *secret key* algorithms and *public key* algorithms. Secret key algorithms are symmetric in the sense that both participants  in the communication share a single key. Figure  illustrates the use of secret key encryption to transmit data over an otherwise insecure channel. The Data Encryption Standard (DES) is the best known example of a secret key encryption function.



Figure: Secret Key Encryption.


In contrast to a pair of participants sharing a single secret key, *public key* cryptography involves each participant having a *private* key that it shares with no one else, and a *public* key that is published so everyone knows it. To send a secure message to this participant, one encrypts the message using the widely known public key. The participant then decrypts the message using its private key. This scenario is depicted in Figure . RSA---named after its inventors, Rivest, Shamir, and Adleman---is the best known public key encryption algorithm.



Figure: Public Key Encryption.

There is actually a third type of cryptography algorithm, called a *hash* or *message digest* function. Unlike the preceding two types of algorithms, cryptographic hash functions involve the use of *no* keys. Instead, the idea of a cryptographic hash function is to map a potentially large message into a small fixed-length number, analogous to the way a regular hash function maps values from a large space into values from a small space.

The best way to think of a cryptographic hash function is that it computes a *cryptographic checksum* over a message. That is, just as a regular checksum protects the receiver from accidental changes to the message, a cryptographic checksum protects the receiver from malicious changes to the message. This is because all cryptographic hash algorithms are carefully selected to be one-way functions---given a cryptographic checksum for a message, it is virtually impossible to figure out what message produced that checksum. Said another way, it is not computationally feasible to find two messages that hash to the same cryptographic checksum. The relevance of this property is that if you are given a checksum for a message and you are able to compute exactly the same checksum for that message, then it is highly likely this message produced the checksum you were given.

Although we will not describe it in detail, the most widely used cryptographic checksum algorithm is *Message Digest version 5* (MD5). An important property of MD5, in addition to those outlined in the previous paragraph, is that it is much more efficient to compute than either DES or RSA. We will see the relevance of this fact later in this section.

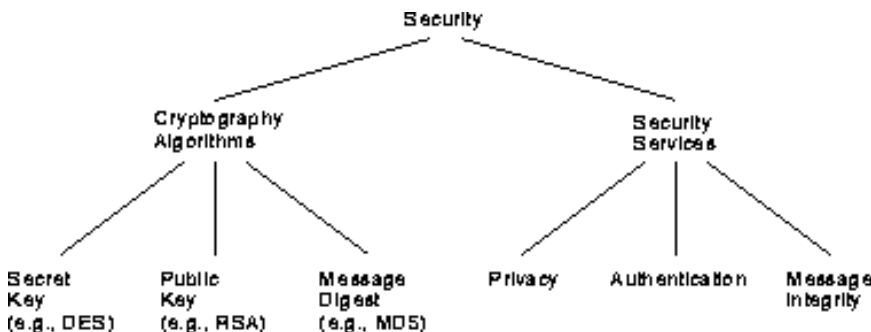



Figure: Taxonomy of Network Security.

To re-emphasize, cryptography algorithms like DES, RSA, and MD5 are just building blocks from which a secure system

can be constructed. Figure  gives a simple taxonomy that illustrates this point. In looking at these services and building blocks, we should consider the following question: How did the participants get the various keys in the first place? This is the key distribution problem, one of the central problems in security, as we will see in the following subsections.

Encryption Algorithms (DES,RSA)

Before showing how encryption algorithms are used to build secure systems, we first describe how the two best known algorithms---DES and RSA---work. We will also give some insight into *why* they work, but there is only so much we can do on this front since the design principles that underlie DES are not public knowledge. In the case of RSA, a deep explanation for why it works would require a background in number theory that is beyond the scope of this book, but we can provide some intuition into the underlying principles. Before looking at either algorithm, however, let's step back and ask what we want from an encryption algorithm.

Requirements

The basic requirement for an encryption algorithm is that it be able to turn plaintext into ciphertext in such a way that only the intended recipient---the holder of the decryption key---can recover the plaintext. What this means is that the encryption method should be safe from attacks by people who do not hold the key. As a starting point, we should assume that the encryption algorithm itself is known, and that only the key is kept secret. The reason for this is that if you depend on the algorithm being secret, then you have to throw it out when you believe it is no longer secret. This means potentially frequent changes of algorithm, which is problematic, since it takes a lot of work to develop a new algorithm. Also, one of the best ways to know that an algorithm is effective is to use it for a long time---if no-one breaks it, it's probably secure. (Fortunately, there are plenty of people who will try to break algorithms and let it be widely known when they have succeeded, so no news is generally good news.) Thus, there is considerable risk in deploying a new algorithm. Therefore, our first requirement is that secrecy of the key, and not the algorithm itself, is the only thing that is needed to ensure the privacy of the data.

It is important to realize that when someone receives a piece of ciphertext, they may have more information at their disposal than just the ciphertext itself. For example, they may know that the plaintext was written in English, which means that the letter 'e' occurs more often in the plaintext than any other letter; the frequency of many other letters and common letter combinations can also be predicted. This information can greatly simplify the task of finding the key. Similarly, they may know something about the likely contents of the message; e.g., the word "login" is likely to occur at the start of a remote login session. This may enable a "known plaintext" attack, which has a much higher chance of success than a "ciphertext only" attack. Even better is a "chosen plaintext" attack, which may be enabled by feeding some information to the sender that you know the sender is likely to transmit---such things have happened in wartime.

The best cryptographic algorithms, therefore, can prevent the attacker from deducing the key even when he knows both the plaintext and the ciphertext. One approach, which is the one taken in DES, is to make the algorithm so complicated that virtually none of the structure of the plaintext remains in the ciphertext. This leaves the attacker with no choice but to search the space of possible keys exhaustively. This can be made infeasible by choosing a suitably large key space and by making the operation of checking a key reasonably costly. As we will see, DES is now becoming only marginally secure on that basis.

DES

DES encrypts a 64-bit block of plaintext using a 64-bit key. The key actually contains only 56 usable bits---the last bit of each of the 8 bytes in the key is a parity bit for that byte. Also, messages larger than 64 bits can be encrypted using DES,

as described below.

DES has three distinct phases: (1) the 64 bits in the block are permuted (shuffled), (2) 16 rounds of an identical operation are applied to the resulting data and the key, and (3) the inverse of the original permutation is applied to the result. This high-level outline of DES is depicted in Figure 1.

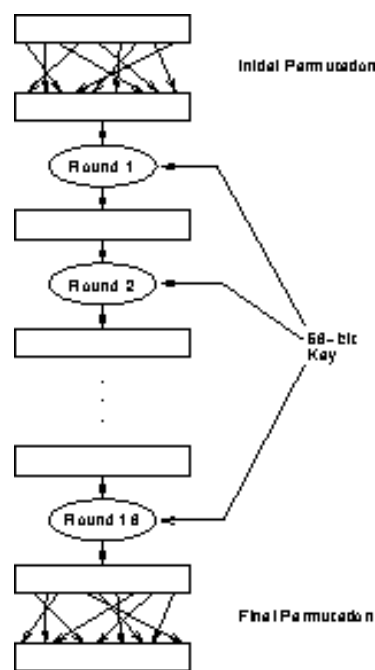


Figure: High-level outline of DES.

A table representing part of the initial permutation is shown in Table 1. The final permutation is the inverse (e.g., bit 40 would be permuted to bit position 1). It is generally agreed that these two permutations add nothing to the security of DES. Some speculate that they were included to make the computation take longer, but it is just as likely that they are an artifact of the initial hardware implementation, involving some restriction of pin layout, for example.

Input Position.	1	2	3	4	5	...	60	61	62	63	64
Output Position	40	8	48	16	56	...	9	49	17	57	25

Table: Initial (and final) DES permutation.

During each round, the 64-bit block is broken into two 32-bit halves, and a different 48 bits are selected from the 56-bit key. If we denote the left and right halves of the block at round i as L_i and R_i , respectively, and the 48-bit key at round i as K_i , then these three pieces are combined during round i according to the following rule:

$$\begin{aligned} L_i &= R_{i-1} \\ R_i &= L_{i-1} \circ F(R_{i-1}, K_i) \end{aligned}$$

where F is a combiner function described below and \circ is the exclusive-OR (XOR) operator. Figure 2 illustrates the basic operation of each round. Note that L_0 and R_0 correspond to the left and right halves of the 64-bit block that results from the initial permutation, and that L_{16} and R_{16} are combined back together to form the 64-bit block to which the final inverse permutation is applied.

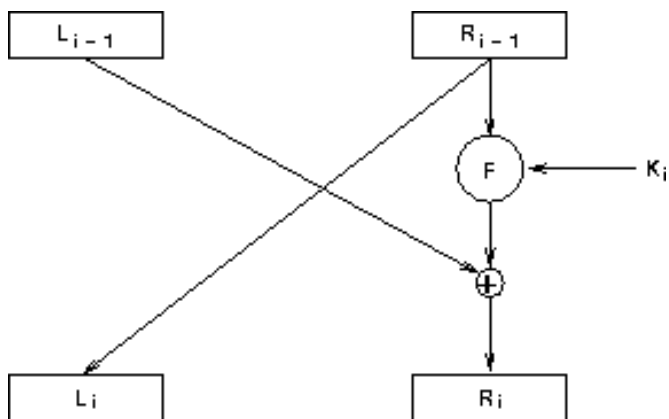


Figure: Manipulation at each round of DES.

We now need to define function **F** and show how each K_i is derived from the 56-bit key. We start with the key. Initially, the 56-bit key is permuted according to Table [1](#). Note that every eighth bit is ignored (e.g. bit 64 is missing from the table), reducing the key from 64 bits to 56. Then for each round, the current 56 bits are divided into two 28-bit halves, and each half is independently shifted left either one or two digits, depending on the round. The number of bits shifted in each round is given in Table [2](#). The 56 bits that result from this shift are used as both input for the next round (i.e., the preceding shift is repeated), and to select the 48 bits that make up the key for the current round. Table [3](#) shows how 48 of the 56 bits are selected; note that they are simultaneously selected and permuted. For example, the bit in position 9 is not selected because it is not in the table.

Input Position.	1	2	3	4	5	...	59	60	61	62	63
Output Position	8	16	24	56	52	...	17	25	45	37	29

Table: DES key permutation.

Round	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Shift	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

Table: DES key shifts per round.

Input Position	1	2	3	4	5	6	7	8	10	11	12	13	14	15	16	17
Output Position	5	24	7	16	6	10	20	18	12	3	15	23	1	9	19	2

Input Position	19	20	21	23	24	26	27	28	29	30	31	32	33	34	36	37
Output Position	14	22	11	13	4	17	21	8	47	31	27	48	35	41	46	28

Input Position	39	40	41	42	44	45	46	47	48	49	50	51	52	53	55	56
Output Position	39	32	25	44	37	34	43	29	36	38	45	33	26	42	30	40

Table: DES compression permutation.

Function **F** combines the resulting 48-bit key for round **i** (K_i) with the right half of the data block after round **i-1** (R_{i-1}) as follows. To simplify our notation, we refer to K_i , R_{i-1} as **K** and **R**, respectively. First, function **F** expands **R** from 32 bits into 48 bits so it can be combined with the 48-bit **K**. It does this by breaking **R** into eight 4-bit chunks, and expanding each chunk into 6 bits by stealing the rightmost and leftmost bit from the left and right adjacent 4-bit chunks, respectively.

This expansion is illustrated in Figure [10.10](#), where **R** is treated as circular in the sense that the first and last chunks get their extra bit from each other.

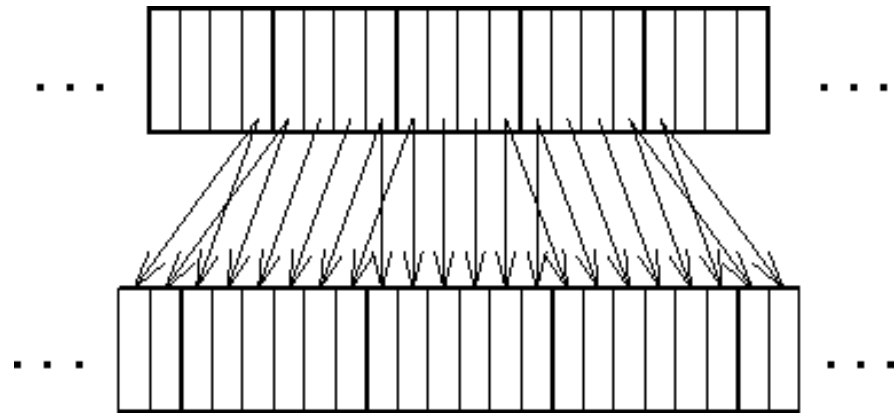


Figure: Expansion phase of DES.

Next, the 48-bit **K** is divided into eight 6-bit chunks, and each chunk is XOR'ed with the corresponding chunk resulting from the previous expansion of **R**. Finally, each resulting 6-bit value is fed through something called a *substitution box* (S box) which reduces each 6-bit chunk back into 4 bits. There are actually eight different S boxes, one for each of the 6-bit chunks. You can think of an S box just performing a many-to-one mapping from 6 bit numbers to 4 bit numbers. Table [10.11](#) gives part of the S box function for the first chunk. We are now done with round **i**.

Input	000000	000001	000010	000011	000100	000101	...
Output	1110	0100	1101	0001	0010	1111	...

Input	...	111010	111011	111100	111101	111110	111111
Output	...	0011	1110	1010	0000	0110	1101

Table: Example DES S-box (bits 1-6).

Notice that the preceding description does not distinguish between encryption and decryption. This is because one of the nice features of DES is that both sides of the algorithm work exactly the same. The only difference is that the keys are applied in the reverse order; i.e., $K_{16}, K_{15}, \dots, K_1$.

Also keep in mind that the preceding discussion is limited to a single 64-bit data block. To encrypt a longer message using DES, a technique known as *cipher block chaining* (CBC) is typically used. The idea of CBC is simple: the ciphertext for block **i** is XOR'ed with the plaintext for block **i+1** before running it through DES. An *initialization vector* (IV) is used in lieu of the non-existent ciphertext for block 0. This IV, which is a random number generated by the sender, is sent along with the message so that the first block of plaintext can be retrieved. CBC on the encryption side is shown in Figure [10.12](#) for a 256-bit (four block) message. Decryption works in the expected way since XOR is its own inverse, with the process starting with the last block and moving towards the front of the message.

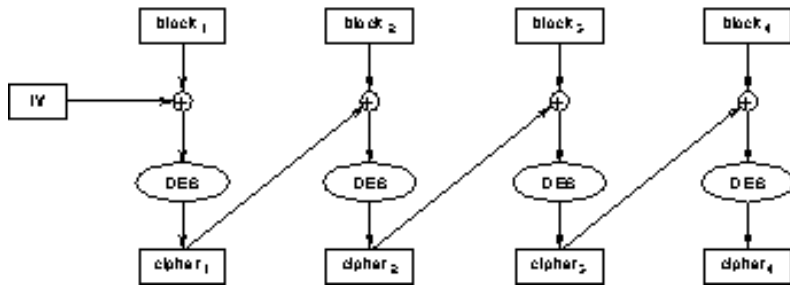


Figure: Cyclic Block Chaining (CBC) for large messages.

We conclude by noting that there is no mathematical proof that DES is secure. What security it achieves it does through the application to two techniques: confusion and diffusion. (Having just plowed through the algorithm, you should now have a deep understanding of these two techniques.) What we can say is that the only known way to break DES is to exhaustively search all possible 2^{56} keys, although on average you would expect to have to search only half of the key space, or $2^{55} = 3.6 \times 10^{16}$ keys. On a 175MHz Alpha workstation, it is possible to do one encryption in $4\mu\text{s}$, meaning it would take $1.4 \times 10^{17}\mu\text{s}$ to break a key. This is approximately 4,500 years. While that seems like a long time, keep in mind that searching a key space is a *highly* parallelizable task, meaning if you could throw 9,000 Alphas at the job, it would take only six months to break a key.

This amount of time is considered borderline-secure in many circles, especially considering that processor speeds are doubling every 18 months. For this reason, many applications are starting to use triple-DES, that is, encrypting the data three times. This can be done with three separate keys, or with two keys: the first is used, then the second key is used, and finally the first key is used again.

RSA

RSA is a much different algorithm, not only because it involves a different key for encryption (public key) and decryption (private key), but also because it is grounded in number theory. In fact, the essential aspect of RSA comes down to how these two keys are selected. The act of encrypting or decrypting a message is expressed as a simple function, although this function requires enormous computational power. In particular, RSA commonly uses a key length of 512 bits, making it much more expensive to compute than DES; more on this below.

The first item of business is to generate a public and private key. To do this, choose two large prime numbers **p** and **q**, and multiply them together to get **n**. Both **p** and **q** should be roughly 256 bits long. Next, choose the encryption key **e**, such that **e** and $(p-1) \times (q-1)$ are relatively prime. (Two numbers are relatively prime if they have no common factor greater than one.) Finally, compute the decryption key **d** such that

$$d = e^{-1} \text{mod } ((p-1) \times (q-1))$$

The public key is constructed from the pair $\{e, n\}$ and the private key is given by the pair $\{d, n\}$. The original primes **p** and **q** are no longer needed. They can be discarded, but must not be disclosed.

Given these two keys, encryption and decryption are defined by the following two formula, respectively:

$$c = m^e \bmod n$$

and

$$m = c^d \bmod n$$

where **m** is the plaintext message and **c** is the resulting ciphertext. Note that **m** must be less than **n**, which implies that it is no more than the 512 bits long. A larger message is simply treated as the concatenation of multiple 512-bit blocks.

Notice that when two participants want to encrypt data they are sending each other using a public key algorithm like RSA, then a pair of public/private key pairs are required. It doesn't work to encrypt with your private key and let the other side decrypt with the public key because everyone has access to the public key, and so could decrypt the message. In other words, participant **A** encrypts data it sends to participant **B** using **B**'s public key and **B** uses its private key to decrypt this data, while **B** encrypts data it sends to **A** using **A**'s public key and **A** decrypts this message using its private key. Observe that **A** cannot decrypt a message that it has sent to **B**; only **B** has the requisite private key.

RSA security comes from the premise that factoring large numbers is a computationally expensive proposition. In particular, if you could factor **n**, you could recover **p** and **q**, which would compromise **d**. The speed at which large numbers can be factored is a function of both the available processor speed and the factoring algorithm being used. It is estimated that 512-bit numbers will be factorable in the next few years, and in fact, people are already starting to use 768 and 1024-bit keys. Keep in mind that while we are concentrating on the security of data as it moves through the network---i.e., the data is vulnerable for only a short time period of time---in general, security people have to consider the vulnerability of data that needs to be stored in archives for tens of years.

Sidebar: Breaking RSA

In 1977, a challenge was issued to break a 129-digit (430-bit) message encrypted using RSA. It was believed that the code was impregnable, requiring 40 quadrillion years of computation using the currently known algorithms for factoring large numbers. In April of 1994, a mere 17 years later, four scientists reported that they had broken the code. The hidden message:

THE MAGIC WORDS ARE SQUEAMISH OSSIFRAGE

The task was accomplished using a factoring method that requires approximately 5000 MIP-years. This was done over an eight month period of time by dividing the problem into smaller pieces, and shipping these pieces, using email, to computers all over the world.

Keep in mind that it doesn't always take 500 MIP-years to break a key, especially when the key is poorly chosen. For example, a security hole was recently exposed in a WWW browser that used RSA to encrypt credit card numbers that were being sent over the Internet. The problem was that the system use a highly predictable method (a combination of process id plus time-of-day) to generate a random number that was, in turn, used to generate a private and public key. Such keys were easily broken.

Implementation and Performance

DES is two to three orders of magnitude faster than RSA, depending on whether the implementations are done in hardware or software. When implemented in software, we have measured DES to process data at 36Mbps and RSA to process data at only 1Kbps on a 175MHz Alpha workstation. For comparison, a software implementation of MD5 can process data at a rate of 80Mbps on the same workstation. When implemented in hardware, that is, by custom VLSI chips, it has been reported that DES can achieve rates approaching 1Gbps and RSA can achieve a whopping 64Kbps.

Perhaps surprisingly, DES is the more likely of the two algorithms to be implemented in hardware on a given computer. This is because even when implemented in hardware, RSA is still too slow to be of any practical use in encrypting data messages. Instead, RSA is typically used to encrypt secret keys that are passed between two participants. These participants then use DES, possibly implemented in hardware, to secure their communication using this key. Because the secret key would only be used for as long as these two participants want to communicate, it is called a *session* key. More on this in following subsections.

Authentication Protocols (Kerberos)

Before two participants are likely to establish a secure channel between themselves---i.e., use DES or RSA to encrypt messages they exchange---they must each first establish that the other participant is who it claims to be. This is the problem of authentication. If you think about authentication in the context of a client/server relationship, say a remote file system, then it is understandable that the server would want to establish the identity of the client---if the client is going to be allowed to modify and delete John's file, then the server is obligated to make sure that the client is, in fact, John. It is also the case, however, that the client often wants to verify the identity of the server. After all, you would not want to start writing sensitive data to what you thought was a file server, only to later discover that it was an impostor process.

This section describes three common protocols for implementing authentication. The first two use secret key cryptography (e.g., DES), while the third uses public key cryptography (i.e., RSA). Note that it is often during the process of authentication that the two participants establish the session key that is going to be used to ensure privacy during subsequent communication. The following includes a discussion of how this process gets bootstrapped.

Simple Three-Way Handshake

A simple authentication protocol is possible when the two participants that want to authenticate each other---think of them as a client and a server---already share a secret key. This situation is analogous to a user (the client) having an account on a computer system (the server), where both the client and server know the password for the account.

The client and server authenticate each other using a simple three-way handshake protocol similar to the one described in Section [4.1](#). In the following, we use $E(m, k)$ to denote the encryption of message m with key k , and $D(m, k)$ to denote the decryption of message m with key k .

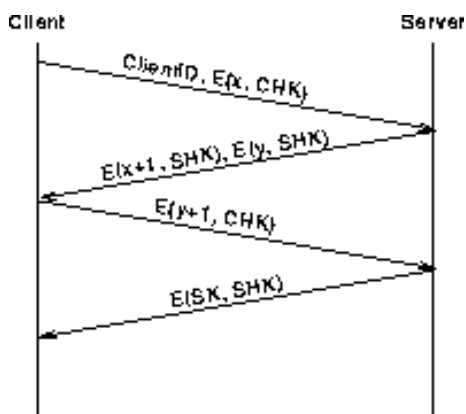




Figure: Three-way handshake protocol for authentication.

As illustrated in Figure , the client first selects a random number x and encrypts it using its secret key, which we denote as CHK (client handshake key). It then sends $E(x, CHK)$, along with an identifier ($ClientID$) for itself to the server. The server uses the key it thinks corresponds to client $ClientID$ (call it SHK for server handshake key) to decrypt the random number. The server adds one to the number it recovers, and sends the result back to the client. It also sends back a random number y that has been encrypted with SHK . Next, the client decrypts the first half of this message and if the result is one more than the random number x that it sent to the server, it knows that the server possesses its secret key. At this point, the client has authenticated the server. The client also decrypts the random number the server sent it (this should yield y), encrypts this number plus one, and sends the result to the server. If the server is able to recover $y+1$ then it knows the client is legitimate.

After the third message, each side has authenticated itself to the other. The fourth message in Figure  corresponds to the server sending the client, encrypted using SHK (which is equal to CHK), a session key (SK). Typically, the client and server use SK to encrypt any future data they send to each other. The advantage of using a session key is that it means that the permanent secret key is only used for a small number of messages, making it harder for an attacker to gather data that might be used to determine the key.

This only begs the question of where the client and server handshake keys came from in the first place. One possibility is that they correspond to a password that a user entered; the $ClientID$ could be the login identifier in this situation. Because a user-selected password might not make a suitable secret key, a transformation is often performed to turn it into a legitimate 56-bit DES key, for example.

Trusted Third Party

A more likely scenario is that the two participants know nothing about each other, but both trust a third party. This third party is sometimes called an *authentication server*, and it uses a protocol to help the two participants authenticate each other. There are actually many different variations of this protocol. The one we describe is the one used in Kerberos, a TCP/IP-based security system developed at MIT.

In the following, we denote the two participants that want to authenticate each other as **A** and **B**, and we call the trusted authentication server **S**. The Kerberos protocol assumes that **A** and **B** each share a secret key with **S**; we denote these two keys as K_A and K_B , respectively. As before, $E(m, k)$ denotes message m encrypted with key k .

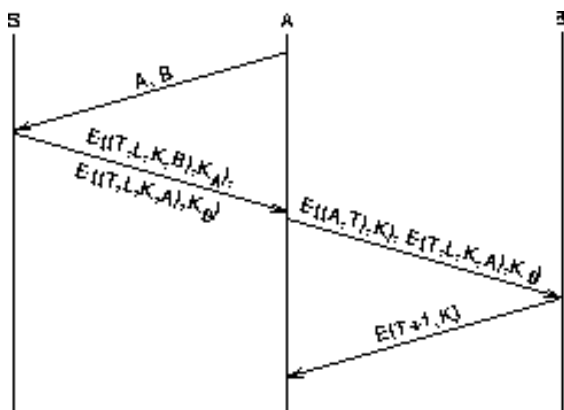




Figure: Third party authentication in Kerberos.

As illustrated in Figure , participant A first sends a message to server S that identifies both itself and B. The server then generates a timestamp T , a lifetime L , and a new session key K . Timestamp T is going to serve much the same purpose as the random number in the simple three-way handshake protocol given above, plus it is used in conjunction with L to limit the amount of time that session key K is valid. Participants A and B will have to go back to server S to get a new session key when this time expires. The idea here is to limit the vulnerability of any one session key.

Server S then replies to A with a two part message. The first part encrypts the three values T , L , and K , along with the identifier for participant B, using the key that the server shares with A. The second part encrypts the three values T , L , and K , along with the participant A's identifier, but this time using the key that the server shares with B. Clearly, when A receives this message, it will be able to decrypt the first part, but not the second part. A simply passes this second part on to B, along with the encryption of A and T using the new session key K . (A was able to recover T and K by decrypting the first part of the message it got from S). Finally, B decrypts the part of the message from A that was originally encrypted by S, and in doing so, recovers T , K , and A. It uses K to decrypt the half of the message encrypted by A, and upon seeing the A and T are consistent in the two halves of the message, replies with a message that encrypts $T+1$ using the new session key K .

A and B can now communicate with each other using the shared secret key K to ensure privacy.

Public Key Authentication

Our final authentication protocol uses public key cryptography; e.g., RSA. The public key protocol is a good one because the two sides need not share a secret key; they only need to know the other side's public key. As shown in Figure , participant A encrypts a random number x using participant B's public key, and B proves it knows the corresponding private key by decrypting the message and sending x back to A. A could authenticate itself to B in exactly the same way.

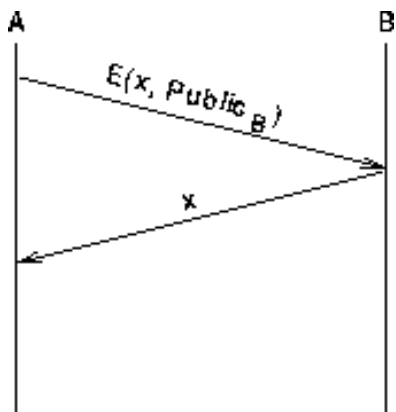


Figure: Public key authentication.

As to the question of where participants learn the others' public key, it is tempting to believe that individual participants should be allowed to post their public keys on a bulletin board. This approach does not work, however, because it would be possible for participant **A** to post its own public key and claim that it is the key for participant **B**. This would allow **A** to masquerade as **B**. Instead, public keys must be retrieved from a trusted source, which is typically known as a *certificate authority* (CA). That is, you go to the trusted CA to get some other participant's public key, and you have to prove through some external mechanism that you are who you say you are when you register your public key with the CA. This reduces the problem to one of every participant having to know just one public key---that of the CA. Note that in general, it is possible to build a hierarchy of CA's: you start at the root CA to get the public key for a second-level CA, which gives you the public key for the third-level CA, and so on, until you learn the public key for the participant you want to communicate with. The root CA would be published in some other trusted source, for example, the *New York Times*.

An alternative to a tree-structured hierarchy is to build a non-hierarchical mesh of trust. We defer discussion of this approach until we have discussed digital signatures, on which this approach depends.


Sidebar: Security Support in IPv6

IPv6 includes hooks to support authentication, integrity, and encryption. First, IPv6 provides an Authentication extension header, which contains only two fields of interest: a Security Parameters Index (SPI), and some amount of Authentication Data. This provides a very general way to support a wide range of authentication algorithms and key management strategies without needing to redefine the header format. It is assumed that any two parties (hosts or routers) that wish to authenticate packets from each other will be able to agree on everything needed to provide authentication, notably an algorithm and a key. Once they have done this, the sending host uses the SPI to identify the appropriate set of agreements to the receiver. The sender then uses the agreed algorithm and key to calculate a checksum over the entire datagram, and the receiver uses the same algorithm and key to verify the checksum. If the packet has been modified in any way, the checksum will fail, thus we achieve integrity. If someone other than the purported sender tries to send the packet, since they do not know the key to the algorithm, they will be unable to forge correct checksum.

A similarly general approach is used for encryption. Specifically, another extension header, called the Encapsulating Security Payload (ESP) header, is used. All headers that precede the ESP header, most notably the IPv6 header, are unencrypted, so that routers do not need to decrypt packets before forwarding them. The only mandatory field of the ESP header is the SPI, which again provides an identifier by which the receiver can look up the algorithm and key needed to decrypt the data. Everything that follows the ESP header in the packet is then encrypted.

Message Integrity Protocols

Sometimes two communicating participants do not care whether an eavesdropper is able read the messages they are sending each other, but they are worried about the possibility of an impostor sending messages that claims to be from one of them. That is, the participants want to ensure the integrity of their messages.

One way to ensure the integrity of a message is to encrypt it using DES with CBC, and then use the CBC *residue* (the last block output by the CBC process) as a *message integrity code* (MIC). (For the CBC example given in Figure , "cipher 4" is the CBC residue.) The plaintext message plus the MIC would be transmitted to the receiver, with the MIC acting as a sort of checksum---if the receiver could not reproduce the attached MIC using the secret key it shares with the sender, then the message was either not sent by the sender, or it was modified since it was transmitted. Note that you would not want to use DES with CBC to both encrypt the message for privacy and to generate the MIC for integrity. This is because you would simply end up transmitting the CBC encrypted message with the last block repeated. Thus, anyone that wanted to tamper with the CBC encrypted message could take the value of the final block they wanted to send, and send it twice.

This section looks at three alternatives for ensuring message integrity. The first uses RSA to produce a digital signature. RSA used on its own tends to be slow, but it can be used in combination with MD5 to yield a much more efficient technique. The second and third approaches use MD5 in conjunction with RSA to guarantee message integrity.

Digital Signature Using RSA

A *digital signature* is a special case of a message integrity code, where the code can only have been generated by one participant. The easiest digital signature algorithm to understand is an RSA signature, which works in the obvious way---since a given participant is the only one that knows its own private key, the participant uses this key to produce the signature. Any other participant can verify this signature using the corresponding public key. In other words, to sign a message, you encrypt it using your private key, and to verify a signature, you decrypt it using the public key of the purported sender. Clearly, this means producing an RSA signature is as slow as RSA, which we have already seen is two or three orders of magnitude slower than DES. Observe that the use of keys is exactly reversed relative to their use for privacy: the sender encrypts with the sender's private key rather than the receiver's public key, and the receiver decrypts with the sender's public key, not the receiver's private key.

Note that the NIST has proposed a digital signature standard known as DSS that is similar to the approach just described, except it uses an alternative algorithm, called El Gamel, instead of RSA.

Keyed MD5

Recall that MD5 produces a cryptographic checksum for a message. This checksum does not depend on a secret key, so it does not prevent an importer from creating a message that claims to be from someone else, and computing an MD5 checksum for that message. However, there are two ways to use MD5 in combination with a public key algorithm like RSA to implement message integrity. Both approaches overcome the performance problems inherent in using RSA alone.

The first method, which is commonly referred to as *keyed MD5*, works as follows. The sender generates a random key **k** and runs MD5 over the concatenation of the message (denoted **m**) and this key. In practice, the random key is attached to either the front or back of the message for the purpose of running MD5; **k** is then removed from the message once MD5 is finished. The random key is then encrypted using RSA and the sender's private key. The original message, the MD5 checksum, and the encrypted version of the random key are then packaged together and sent to the receiver. The following summarizes the complete message transmitted by the sender:

$$\mathbf{m} + \text{MD5}(\mathbf{m} + \mathbf{k}) + \text{E}(\mathbf{k}, \text{private})$$

where $\text{MD5}(\mathbf{s})$ represents applying the MD5 algorithm to string **s**, and **a** + **b** denotes the concatenation of strings **a** and **b**.

The receiver recovers the random key using the purported sender's public RSA key, and then applies MD5 to the concatenation of this random key and the body of the message. If the result matches the checksum sent with the message, then the message must have been sent by the participant that generated the random key.

MD5 with RSA Signature

The second method for using MD5 in combination with RSA works as follows. The sender runs MD5 over the original message it wants to protect, producing an MD5 checksum. It then signs this checksum with its own private RSA key. That is, the sender does not sign the entire message, it just signs the checksum. The original message, the MD5 checksum, and the RSA signature of the checksum are then transmitted. Using the same notation as above, this means the sender transmits:

$$\mathbf{m} + \text{MD5}(\mathbf{m}) + \text{E}(\text{MD5}(\mathbf{m}), \text{private}).$$

The receiver verifies the message by making sure the signature is correct, that is, by decrypting the signature with the sender's public key and comparing the result with the MD5 checksum sent with the message. The two should be equal.

Using Digital Signatures for Key Distribution

We mentioned above that digital signatures could be used to build a mesh of trust by which keys can be distributed, as an alternative to a strict hierarchy. Now that we know how digital signatures work, providing us with a way to say "this message was generated by participant **X** and has not been altered", we can describe the mesh approach.

To begin, suppose participants **A** and **B** exchange public keys while they are in the same room; that is, they can verify that the keys really belong to the right people. At some later time, **A** exchanges public keys with **C**. **A** can now use his private key to sign the public key of **C** and send it to **B**. In other words, **A** sends a message that says, in effect, "I was in a room with **C** and I believe that this is **C**'s public key" and signs this message using his own private key. Because **B** has a trustworthy copy of **A**'s public key, **B** can verify the signature on this message. As long as **B** trusts that **A** was not fooled into mistakenly signing **C**'s key, **B** now has a trustworthy copy of **C**'s public key.

Thus, a reasonable strategy is for **C** to get his key signed by lots of people, and with luck, anyone with whom **C** wants to communicate will be able to find one person they trust in the set of people who have signed **C**'s public key. Note that there can be arbitrarily many links in the chain of trust, but if one person in the chain signs a key when they are not sure it really belongs to the right person, then the trust is broken. As an aside, note that key-signing parties---these parties involve a lot of **A**'s, **B**'s and **C**'s in a room with their laptops---have become a regular feature of IETF meetings. This approach to key management is used by the Pretty Good Privacy (PGP) suite of software.

[Next](#) [Up](#) [Previous](#)

Next: [Summary](#) **Up:** [End-to-End Data](#) **Previous:** [Data Compression](#)

Summary

This chapter has described three different data manipulations that applications apply to their data before sending it over a network. Unlike the protocols described earlier in this book, which you can think of as processing *messages*, these three transformations process *data*.

The first data manipulation is presentation formatting, where the problem is encoding the different types of data that application programs compute on---e.g., integers, floats, arrays, structures---into packets that can be transmitted over a network. This involves both translating between machine and network byte order, and linearizing compound data structures. We outlined the design space for presentation formatting, and discussed three specific mechanisms that fall on different points in this design space: XDR, ASN.1, and NDR.

The second data manipulation is compression, which is concerned with reducing the bandwidth required to transmit different types of data. Compression algorithms can be either loss-less or lossy, with lossy algorithms being most appropriate for image and video data. JPEG and MPEG are example lossy compression protocols for still image and video data, respectively.

The third data manipulation is encryption, which is used to protect the privacy of data sent over a network. We looked at two basic encryption algorithms: shared key algorithms (e.g., DES) and public key algorithms (e.g., RSA). As we have seen, encryption algorithms actually play a much bigger role in network security than just supporting privacy. They also form the basis for protocols used to ensure authentication and message integrity.

Open Issue: Presentation---Layer of the 90's

At one time, data manipulations such as the ones discussed in this chapter were treated as an afterthought by networking people. There were various reasons for this. In the case of presentation formatting, the network was so slow that no one cared how bad the encoding strategy was. In the case of compression, it is the recent advance of CD-ROM and graphical I/O devices (i.e., the growing popularity of multimedia) that has made the compression of images a fruitful area to pursue. In the case of security, it is the recent explosion of the World-Wide-Web, and the implications of using the Web for commerce, that has made people more security conscious. (Of course, the military has always been interested in security.) Today, however, one could argue that the data manipulation level of the network architecture is where the action is. In short, it is the set of data manipulation functions available to application programs that is limiting what we can do with computer networks.

Network security is a good example of services that are just around the corner. Using the basic cryptography algorithms like DES, RSA, and MD5, people are designing security protocols that do much more than ensure privacy, authentication, and message integrity. For example, there are now protocols for anonymous digital cash, certified mail, secure elections, and simultaneous contract signing.

In the case of data compression, formats like MPEG were not specifically designed for use on a network; they were designed to provide a format for storing video on disk. It is not completely clear that MPEG makes sense for sending video over a network. For example, if a B or P frame is dropped by the network, then it is possible to simply replay the previous frame without seriously compromising the video; one frame out of 30 is no big deal. On the other hand, a lost I frame has serious consequences---none of the subsequent B and P frames can be processed without it. Thus, losing an I frame would result in losing multiple frames of the video. While one could retransmit the missing I frame, the resulting delay would probably not be acceptable in a real-time video conference. One interesting avenue of research is to have the application ``teach" the network how to treat different types of messages; for example, it is OK to drop messages carrying B and P frames, but not messages carrying I frames. Note, however, that this is a significant departure from the current model in which all messages are treated equally by the network.

One final issue to consider is the relationship among these three transformations. In most cases, one would perform presentation encoding before compression or encryption---the application first packs and linearizes a data structure before trying to compress or encrypt it. Between compression and encryption,

you would always compress before encrypting. This is for two reasons. First, if the encryption algorithm is a good one, then there is no obvious redundancy for the compression algorithm to remove. The second reason is that by first compressing the data, you are able to make the ciphertext resulting from encryption even more obscure, and therefore, more difficult to break.

Next	Up	Previous
------	----	----------

Next: [Further Reading](#) **Up:** [End-to-End Data](#) **Previous:** [Summary](#)

Copyright 1996, Morgan Kaufmann Publishers

Further Reading

Our recommended reading list for this chapter includes two papers on compression, and two papers on network security. The two compression papers give an overview of the JPEG and MPEG standards, respectively. Their main value is in explaining the various factors that shaped the standards. The two security-related papers, taken together, give a good overview of the topic. The Lampson paper contains a formal treatment of security, while the Satyanarayanan paper gives a nice description of how a secure system is designed in practice.

The final paper on the reading list discusses the importance of data manipulations to network performance, and gives two architectural principles that should guide the design of future protocols.

- G. Wallace. The JPEG Still Picture Compression Standard. *Communications of the ACM*, 34(1): 30--44, April 1991.
- D. Le Gall. MPEG: A Video Compression Standard for Multimedia Applications. *Communications of the ACM*, 34(1): 46--58, April 1991.
- B. Lampson, et. al. Authentication in Distributed Systems: Theory and Practice. *ACM Transactions on Computer Systems*, 10(4): 265--310, November 1992.
- M. Satyanarayanan. Integrating Security in a Large Distributed System. *ACM Transactions on Computer Systems*, 7(3): 247--280, August 1989.
- D. Clark and D. Tennenhouse. Architectural Considerations for a New Generation of Protocols. *Proceedings of the SIGCOMM '91 Symposium*, 200--208", September 1991.

Unfortunately, there is no single paper that gives a comprehensive treatment of presentation formatting. Aside from the XDR, ASN.1/BER, and NDR specifications (see [[Sri95b](#),[X.292a](#),[X.292b](#),[Ope94](#)]), three other papers cover topics related to presentation formatting: [[OPM94](#)], [[Lin93](#)], and [[CLNZ89](#)]. All three discuss performance-related issues.

On the topic of compression, a good place to start is with Huffman encoding, which was originally

defined in [Huf52]. The original LZ algorithm is presented in [ZL77], and an improved version of that algorithm can be found in [ZL78]. Both of these papers are of a theoretical nature. The work that brought the LZ approach into wide-spread practice can be found in [Wel84]. For a more complete overview of the topic of compression, [Nel92] is recommended. You can also learn about compression in any of several recent books on multimedia. We recommend [WMC94], which has an extremely high science-to-hype ratio, and [Buf94], which is a collection of contributed chapters that span the range of multimedia topics.

On the topic of network security, there are several good books to choose from. We recommend two: [Sch94] and [KPS95]. The former gives a comprehensive treatment of the topic, including sample code, while the latter gives a very readable overview of the subject.

Finally, we recommend the following two live references:

- <http://roger-rabbit.cs.berkeley.edu/mpeg/index.html>: a collection of MPEG-related programs, some of which are used in the following exercises;
- <ftp://cert.org/pub>: a collection of security-related notices posted by the Computer Emergency Response Team (CERT).

Next	Up	Previous
------	----	----------

Next: [Exercises](#) **Up:** [End-to-End Data](#) **Previous:** [Open Issue: Presentation---Layer](#)

[Next](#) [Up](#) [Previous](#)

Next: [Congestion Control](#) **Up:** [End-to-End Data](#) **Previous:** [Further Reading](#)

Exercises

Copyright 1996, Morgan Kaufmann Publishers

Congestion Control

- [Problem: Allocating Resources](#)
 - [Issues](#)
 - [Queuing Disciplines](#)
 - [TCP Congestion Control](#)
 - [Congestion Avoidance Mechanisms](#)
 - [Virtual Clock](#)
 - [Summary](#)
 - [Open Issue: Inside versus Outside the Network](#)
 - [Further Reading](#)
 - [Exercises](#)
-

Problem: Allocating Resources

Now that we have seen all the layers that make up a network architecture, we return to an issue that spans the entire protocol hierarchy---how to effectively and fairly allocate the available resources among a collection of competing users. The resources being shared include the bandwidth of the links, and the buffers on the routers (switches) where packets are queued waiting to be transmitted over these links. Packets *contend* at a router for the use of a link, with each contending packet placed in a queue waiting its turn to be transmitted over the link. When enough packets are contending for the same link, the queue overflows and packets have to be dropped. It is at this stage that the network is said to be *congested*. Most networks provide a *congestion control* mechanism to deal with just such a situation.

Congestion control and resource allocation are two sides of the same coin. On the one hand, if the network takes a proactive role in allocating resources---for example, scheduling which virtual circuit gets to use a given physical link during a certain period of time---then congestion can be avoided, thereby making congestion control unnecessary. Allocating network resources with any precision is extremely difficult to do, however, because the resources in question are distributed throughout the network; multiple links connecting a series of routers need to be scheduled. On the other hand, one can always let packet sources send as much data as they want, and then recover from congestion should it occur. This is the easier approach, but can be disruptive because it is likely that many packets will be discarded by the network before congestion can be controlled. There are also solutions in the middle, whereby inexact allocation decisions are made, but congestion can still occur and so some mechanism is still needed to recover from it. Whether you call such a mixed solution congestion control or resource allocation does not really matter. In some sense, it is both.

For our purposes, we discuss the problem in terms of congestion control, which has two points of implementation. At each router, it defines a queuing discipline. This queuing discipline specifies the order in which packets get transmitted, and as a consequence, affects which packets get dropped when congestion occurs. The queuing discipline also has the potential to segregate traffic, that is, to keep one user's packets from unduly affecting another user's packets. At the end hosts, the congestion control mechanism paces how fast sources are allowed to send packets. This is done in an effort to keep congestion from occurring in the first place, and should it occur, to help eliminate the congestion.

This chapter starts by giving a general overview of congestion control. The second section then discusses different queuing disciplines that can be implemented on the routers inside the network, and

the third section describes how TCP implements congestion control on the hosts at the edges of the network. The next section then explores various techniques that shift the burden of congestion control between the routers and the end hosts in an effort to avoid congestion before it becomes a problem. The chapter concludes with a discussion of a more proactive approach to congestion control, in which resources are reserved on behalf of an end-to-end flow of data.

Next	Up	Previous
----------------------	--------------------	--------------------------

Next: [Issues](#) **Up:** [Congestion Control](#) **Previous:** [Congestion Control](#)

Issues

Congestion control is a complex issue, and one that has been a subject of much study ever since the first network was designed. In fact, it is still an active area of research. One of the factors that makes congestion control complex is that it is not isolated to one single level of a protocol hierarchy. It is partially implemented in the routers or switches inside the network, and partially in the transport protocol running on the end hosts. One of the main goals of this chapter is to define a framework in which these mechanisms can be understood, as well as give the relevant details about a representative sample of mechanisms.

Network Model

Taxonomy

Evaluation Criteria

Queuing Disciplines

Regardless of how simple or how sophisticated the rest of the congestion control mechanism, each router must implement some queuing discipline that governs how packets are buffered waiting to be transmitted. The queuing algorithm can be thought of as allocating both bandwidth (which packets get transmitted) and buffer space (which packets get discarded). This section introduces two common queuing algorithms---FIFO and Fair Queuing---and identifies several variations that have been proposed.

FIFO

Fair Queuing

TCP Congestion Control


This section describes the predominant example of end-to-end congestion control in use today, that implemented by TCP. The essential strategy of TCP is to send packets into the network without a reservation, and then react to observable events that occur. TCP assumes only FIFO queuing in the network's routers, but also works with fair queuing.

TCP congestion control was introduced into the Internet in the late 1980's by Van Jacobson, roughly eight years after the TCP/IP protocol stack had become operational. Immediately preceding this time, the Internet was suffering from congestion collapse---hosts would send their packets into the Internet as fast as the advertised window would allow, congestion would occur at some router causing packets to be dropped, and the hosts would timeout and retransmit their packets, resulting in even more congestion.

Broadly speaking, the idea of TCP congestion control is for each source to determine how much capacity is available in the network, so it knows how many packets it can safely have in transit. Once a given source has this many packets in transit, it uses the arrival of an ACK as a signal that one of its packets has left the network, and it is therefore safe to insert a new packet into the network without adding to the level of congestion. By using ACKs to pace the transmission of packets, TCP is said to be *self clocking*. Of course, determining the available capacity in the first place is no easy task. To make matters worse, because other connections come and go, the available bandwidth changes over time, meaning that any given source must be able to adjust the number of packets it has in transit. The rest of this section describes the algorithms used by TCP to address these, and other, problems.

Note that although we describe these mechanisms one at a time, thereby giving the impression that we are talking about three independent mechanisms, it is only when they are taken as a whole that we can define TCP congestion control.

Additive Increase/Multiplicative Decrease

TCP maintains a new state variable for each connection-- `CongestionWindow`---which is used by the source to limit how much data it is allowed to have in transit at a given time. The congestion window is congestion control's counterpart to flow control's advertised window. TCP is modified to have no more than the minimum of the congestion window and the advertised window bytes of unacknowledged data. Thus, using the variables defined in Section , TCP's effective window is revised as follows:

```
MaxWindow = MIN(CongestionWindow, AdvertisedWindow)
EffectiveWindow = MaxWindow - (LastByteSent - LastByteAcked).
```

That is, `MaxWindow` replaces `AdvertisedWindow` in the calculation of `EffectiveWindow`. Thus, a TCP source is allowed to send no faster than the slowest component---the network or the destination host---can accommodate.

The problem, of course, is how TCP comes to learn an appropriate value for `CongestionWindow`. Unlike the `AdvertisedWindow`, which is sent by the receiving side of the connection, there is no one to send a suitable `CongestionWindow` to the sending side of TCP. The answer is that the TCP source sets the `CongestionWindow` based on the level of congestion it perceives to exist in the network. This involves decreasing the congestion window when the level of congestion goes up, and increasing the congestion window when the level of congestion goes down. Taken together, the mechanism is commonly called *additive increase/multiplicative decrease*.

The key question, then, is how does the source determine that the network is congested and it should decrease the congestion window? The answer is based on the observation that the main reason packets are not delivered, and a timeout results, is that a packet was dropped due to congestion. It is rare that a packet is dropped because of an error during transmission. Therefore, TCP interprets timeouts as a sign of congestion, and reduces the rate at which it is transmitting. Specifically, each time a timeout occurs, the source sets `CongestionWindow` to half of its previous value. This halving of the `CongestionWindow` for each timeout corresponds to the "multiplicative decrease" part of the mechanism.

Although `CongestionWindow` is defined in terms of bytes, it is easiest to understand multiplicative decrease if we think in terms of whole packets. For example, suppose the `CongestionWindow` is currently set to 16 packets. If a loss is detected, `CongestionWindow` is set to 8. (Normally, a loss is detected when a timeout occurs, but as we see below, TCP has another mechanism to detect dropped packets.) Additional losses cause `CongestionWindow` to be reduced to 4, then 2, and finally 1 packet. `CongestionWindow` is not allowed to fall below the size of a single packet, or in TCP terminology, the *maximum segment size* (`MSS`).

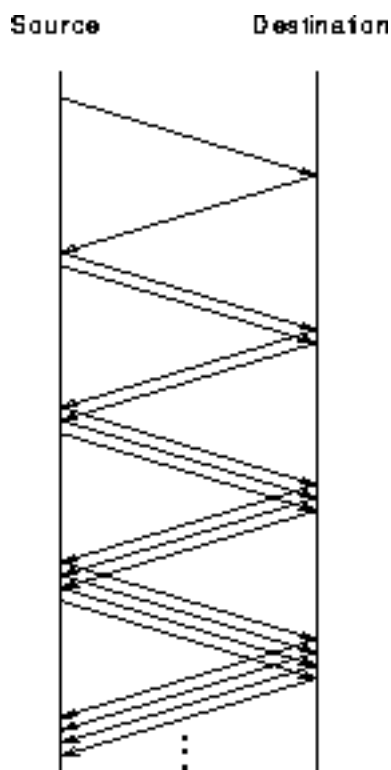




Figure: Packets in transit during additive increase: add one packet each RTT.

A congestion control strategy that only decreases the window size is obviously too conservative. We also need to be able to increase the congestion window to take advantage of newly available capacity in the network. This is the "additive increase" part of the mechanism, and it works as follows. Every time the source successfully sends a `CongestionWindow`'s worth of packets---that is, each packet sent out during the last RTT has been ACK'ed---it adds the equivalent of one packet to `CongestionWindow`. This linear increase is illustrated in Figure . Note that in practice, TCP does not wait for an entire window's worth of ACKs to add one packet's worth to the congestion window, but instead increments `CongestionWindow` by a little for each ACK that arrives. Specifically, the congestion window is incremented as follows each time an ACK arrives:

```
Increment = (MSS * MSS) / CongestionWindow
CongestionWindow += Increment
```

That is, rather than incrementing `CongestionWindow` by an entire MSS each RTT, we increment it by a fraction of MSS every time an ACK is received. Assuming each ACK acknowledges the receipt of MSS bytes, then that fraction is `MSS / CongestionWindow`.

This pattern of continually increasing and decreasing the congestion window continues throughout the lifetime of the connection. In fact, if you plot the current value of `CongestionWindow` as a function of time, you get a "sawtooth" pattern, as illustrated in Figure . The important thing to understand about additive increase/multiplicative decrease is that the source is willing to reduce its congestion window at a much faster rate than it is willing to increase its congestion window. This is in contrast to an additive increase/additive decrease strategy in which the window is incremented by 1 packet when an

ACK arrives and decreased by 1 when a timeout occurs. It has been shown that additive increase/multiplicative decrease is a necessary condition for a congestion control mechanism to be stable.

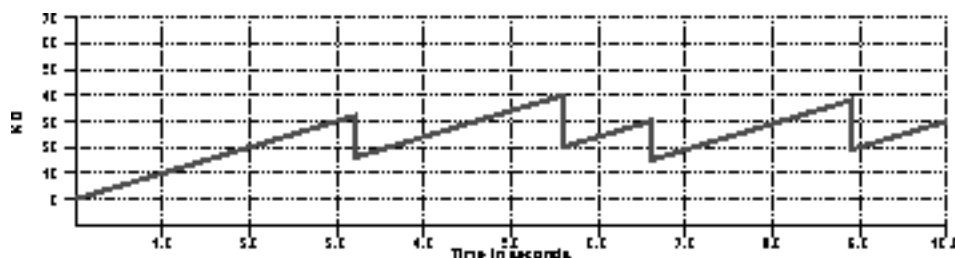


Figure: Typical TCP sawtooth pattern.

Finally, since a timeout is an indication of congestion, triggering multiplicative decrease, TCP needs the most accurate timeout mechanism it can afford. We already covered TCP's timeout mechanism in Section [□](#), and so we do not repeat it here. The two main things to remember about that mechanism are that (1) timeouts are set as a function of both the average RTT and the standard deviation in that average, and (2) due to the cost of measuring each transmission with an accurate clock, TCP only samples the round trip time once per RTT (rather than once per packet) using a coarse-grain (500ms) clock.

Slow Start

The additive increase mechanism just described is the right thing to do when the source is operating close to the available capacity of the network, but it takes too long to ramp up a connection when it is starting from scratch. TCP therefore provides a second mechanism, ironically called *slow start*, that is used to increase the congestion window rapidly from a cold start. Slow start effectively increases the congestion window exponentially, rather than linearly.

Specifically, the source starts out by setting `CongestionWindow` to one packet. When the ACK for this packet arrives, TCP adds one packet to `CongestionWindow` and then sends two packets. Upon receiving the corresponding two ACKs, TCP increments `CongestionWindow` by two---one for each ACK---and next sends four packets. The end result is that TCP effectively doubles the number of packets it has in transit every RTT. Figure [□](#) shows the growth in the number of packets in transit during slow start. Compare this to the linear growth of additive increase illustrated in Figure [□](#).

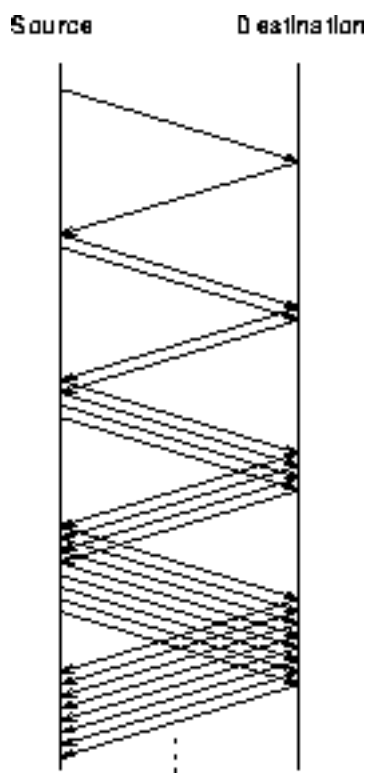


Figure: Packets in transit during slow start.

Why any exponential mechanism would be called "slow" is puzzling at first, but can be explained if put in the proper historical context. We need to compare slow start not against the linear mechanism of the previous subsection, but against the original behavior of TCP. Consider what happens when a connection is established and the source first starts to send packets; i.e., it currently has no packets in transit. If the source sends as many packets as the advertised window allows---which is exactly what TCP did before slow start was developed---then even if there is a fairly large amount of bandwidth available in the network, the routers may not be able to consume this burst of packets. It all depends on how much buffer space is available at the routers. Slow start was therefore designed to *space* packets out so that this burst does not occur. In other words, even though its exponential growth is faster than linear growth, slow start is much "slower" than sending an entire advertised window's worth of data all at once.

There are actually two different situations in which slow start runs. The first is at the very beginning of a connection, at which time the source has no idea how many packets it is going to be able to have in transit at a given time. (Keep in mind that TCP runs over everything from 9600bps links to 2.4Gbps links, so there is no way for the source to know the network's capacity.) In this situation, slow start continues to double `CongestionWindow` each RTT until there is a loss, at which time a timeout causes multiplicative decrease to divide `CongestionWindow` by two.

The second situation where slow start is used is a bit more subtle; it occurs when the connection goes dead waiting for a timeout to occur. Recall how TCP's sliding window algorithm works---when a packet is lost, the source eventually reaches a point where it has sent as much data as the advertised window allows, and so it blocks waiting for an ACK that will not arrive. Eventually, a timeout happens, but by this time there are no packets in transit, meaning that the source will receive no ACKs to "clock" the

transmission of new packets. The source will instead receive one big cumulative ACK that reopens the entire advertised window, but as explained above, the source uses slow start to restart the flow of data rather than dumping a window's worth of data on the network all at once.

Although the source is using slow start again, it now knows more information than it did at the beginning of a connection. Specifically, the source has the current value of `CongestionWindow`, which because of the timeout, has already been divided by two. Slow start is used to rapidly increase the sending rate up to this value, and then additive increase is used beyond this point. Notice that we have a tiny bookkeeping problem to take care of, in that we want to remember the ``target" congestion window resulting from multiplicative decrease, as well as the ``actual" congestion window being used by slow start. To address this problem, TCP introduces a temporary variable, typically called `CongestionThreshold`, that is set equal to the `CongestionWindow` resulting from multiplicative decrease. Variable `CongestionWindow` is then reset to one packet, and it is incremented by one packet for every ACK that is received until it reaches `CongestionThreshold`, at which point it is incremented by one packet per RTT.

In other words, TCP increases the congestion window as defined by the following code fragment:

```
{
    u_int    cw = state->CongestionWindow;
    u_int    incr = state->maxseg;

    if (cw > state->CongestionThreshold)
        incr = incr * incr / cw;
    state->CongestionWindow = MIN(cw + incr, TCP_MAXWIN);
}
```

where `state` represents the state of a particular TCP connection and `TCP_MAXWIN` defines an upper bound on how large the congestion window is allowed to grow.

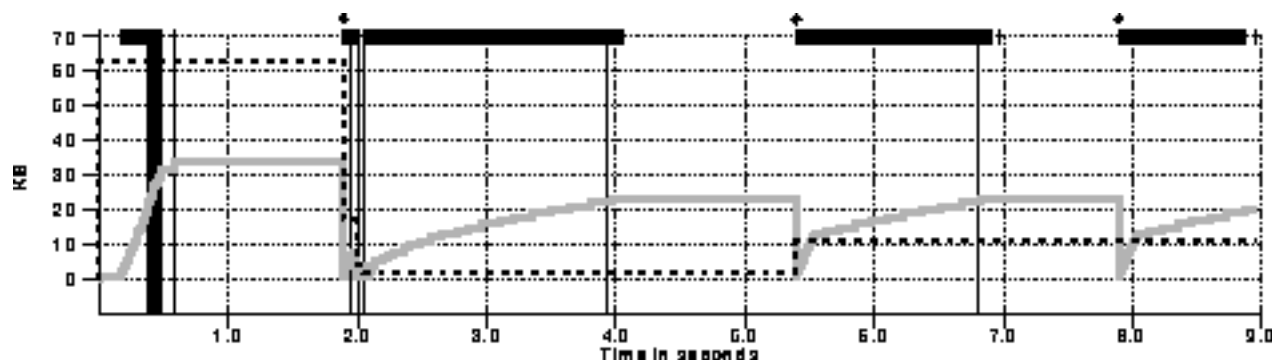



Figure: Behavior of TCP congestion control.

Figure  traces how TCP's `CongestionWindow` increases and decreases over time, and serves to illustrate the interplay of slow start and additive increase/multiplicative decrease. This trace was taken

from an actual TCP connection, and traces the current value of `CongestionWindow`---the thick grey line---over time. The graph also depicts other information about the connection:

- the vertical bars show when a packet that was eventually retransmitted was first transmitted,
- the small hash marks at the top of the graph show the time when each packet is transmitted, and
- the circles at the top of the graph show when a timeout occurs.

There are several things to notice about this trace. The first is the rapid increase in the congestion window at the beginning of the connection. This corresponds to the initial slow start phase. The slow start phase continues until several packets are lost at about 0.4 seconds into the connection, at which time `CongestionWindow` flattens out at about 34KB. (Why so many packets are lost during slow start is discussed below.) The reason the congestion window flattens is that there are no ACKs arriving, due to the fact that several packets were lost. In fact, no new packets are sent during this time, as denoted by the lack of tick marks at the top of the graph. A timeout eventually happens at approximately 2 seconds, at which time the congestion window is divided by two (i.e., cut from approximately 34KB to around 17KB), and `CongestionThreshold` is set to this value. Slow start then causes `CongestionWindow` to be reset to one packet, and start ramping up.

There is not enough detail in the trace to see exactly what happens when a couple of packets are lost just after 2 seconds, so we jump ahead to the linear increase in the congestion window that occurs between 2 and 4 seconds. This corresponds to additive increase. At about 4 seconds, `CongestionWindow` flattens out, again due to a lost packet. Now, at about 5.5 seconds

- a timeout happens, causing the congestion window to be divided by two, dropping it from approximately 22KB to 11KB, and `CongestionThreshold` is set to this amount;
- `CongestionWindow` is reset to one packet, as the sender enters slow start;
- slow start causes `CongestionWindow` to grow exponentially until it reaches `CongestionThreshold`;
- `CongestionWindow` then grows linearly.

The same pattern is repeated at around 8 seconds when another timeout occurs.

We now return to the question of why so many packets are lost during the initial slow start period. What TCP is attempting to do here is learn how much bandwidth is available on the network. This is a very difficult task. If the source is not aggressive at this stage, for example only increasing the congestion window linearly, then it takes a long time for it to discover how much bandwidth is available. This can have a dramatic impact on the throughput achieved for this connection. On the other hand, if the source

is aggressive at this stage, as is TCP during exponential growth, then the source runs the risk of having half a window's worth of packets dropped by the network.

To see what can happen during exponential growth, consider the situation where the source was just able to successfully send 16 packets through the network, and then doubles its congestion window to 32. Suppose, however, that the network just happens to have enough capacity to support only 16 packets from this source. The likely result is that 16 of the 32 packets sent under the new congestion window will be dropped by the network; actually, this is the worst case outcome, since some of the packets will be buffered in some router. This problem will become increasingly severe as the delay \times bandwidth product of networks increases. For example, a delay \times bandwidth product of 500KB means that each connection has the potential to lose up to 500KB of data at the beginning of each connection. Of course, this assumes both the source and destination implement the "big windows" extension.

Some have proposed alternatives to slow start whereby the source tries to estimate the available bandwidth through more clever means of sending out a bunch of packets and seeing how many make it through. A technique called *packet-pair* is representative of this general strategy. In simple terms, the idea is to send a pair of packets with no spacing between them. Then, the source sees how far apart the ACKs for those two packets are. The gap between the ACKs is taken as a measure of how much congestion there is in the network, and therefore, how much increase in the congestion window is possible. The jury is still out on the effectiveness of approaches such as this, although the results are promising.

Fast Retransmit and Fast Recovery

The mechanisms described so far were part of the original proposal to add congestion control to TCP. It was soon discovered, however, that the coarse-grain implementation of TCP timeouts led to long periods of time during which the connection went dead waiting for a timer to expire. Because of this, a new mechanism, called *fast retransmit*, was added to TCP. Fast retransmit is a heuristic that sometimes triggers the retransmission of a dropped packet sooner than the regular timeout mechanism. The fast retransmit mechanism does not replace regular timeouts, it just enhances that facility.

The idea of fast retransmit is straightforward. Every time a data packet arrives at the receiving side, the receiver responds with an acknowledgement, even if this sequence number has already been acknowledged. Thus, when a packet arrives out of order---that is, TCP cannot yet acknowledge the data it contains because earlier data has not yet arrived---TCP resends the same acknowledgement it sent last time. This second transmission of the same acknowledgement is called a *duplicate ACK*. When the sending side sees a duplicate ACK, it knows that the other side must have received a packet out of order, which suggests that an earlier packet might have been lost. Since it is also possible that the earlier packet has only been delayed rather than lost, the sender waits until it sees some number of duplicate ACKs, and then retransmits the missing packet. In practice, TCP waits until it has seen three duplicate ACKs

before retransmitting the packet.

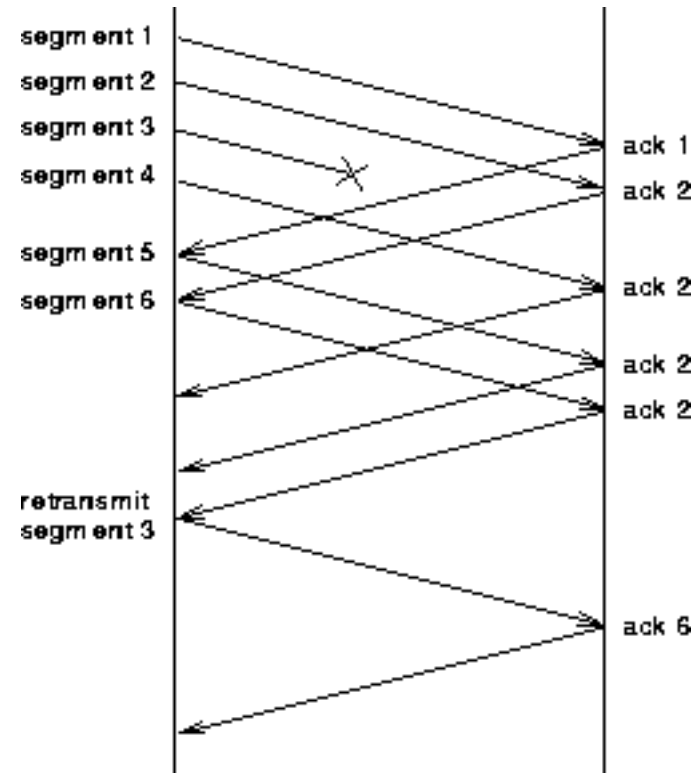



Figure: Fast retransmit based on duplicate ACKs.

Figure  illustrates how duplicate ACKs lead to a fast retransmit. In this example, the destination receives packets 1 and 2, but packet 3 is lost in the network. Thus, the destination will send a duplicate ACK for packet 2 when packet 4 arrives, again when packet 5 arrives, and so on. (To simplify this example, we think in terms of packet 1, 2, 3, and so on, rather than worrying about the sequence numbers for each byte.) When the sender sees the third duplicate ACK for packet 2---the one sent because the receiver had gotten packet 6---it retransmits packet 3. Note that when the retransmitted copy of packet 3 arrives at the destination, it then sends a cumulative ACK for everything up to and including packet 6 back to the source.

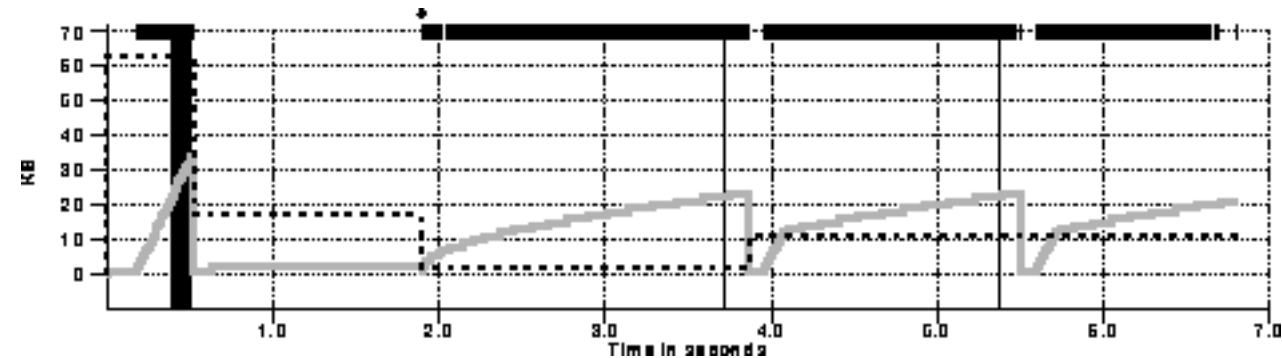





Figure: Trace of TCP with fast retransmit.

Figure  illustrates the behavior of a version of TCP with the fast retransmit mechanism. It is

interesting to compare this trace with that given in Figure , where fast retransmit was not implemented---the long periods during which the congestion window stays flat and no packets are sent has been eliminated. In general, this technique is able to eliminate about half of the coarse-grain timeouts on a typical TCP connection, which results in roughly a 20% improvement in the throughput over what could have otherwise been achieved. Notice, however, that the fast retransmit strategy does not eliminate all coarse-grained timeouts. This is because for a small window size, there will not be enough packets in transit to cause enough duplicate ACKs to be delivered. Given enough lost packets---for example, as happens during the initial slow start phase---the sliding window algorithm eventually blocks the sender until a timeout occurs. Given the current 64KB maximum advertised window size, TCP's fast retransmit mechanism is able to detect up to three dropped packets per window in practice.

Finally, there is one last improvement we can make. When the fast retransmit mechanism signals congestion, rather than drop the congestion window all the way back to one packet and run slow start, it is possible to use the ACKs that are still in the pipe to clock the sending of packets. This mechanism, which is called *fast recovery*, effectively removes the slow start phase that happens between when fast retransmit detects a lost packet and additive increase begins. For example, fast recovery avoids the slow start period between 3.8 and 4 seconds in Figure , and instead simply cuts the congestion window in half (from 22KB to 11KB) and resumes additive increase. In other words, slow start is only used at the beginning of a connection and whenever a coarse-grain timeout occurs. At all other times, the congestion window is following a pure additive increase/multiplicative decrease pattern.

[Next](#) [Up](#) [Previous](#)

Next: [Congestion Avoidance Mechanisms](#) **Up:** [Congestion Control](#) **Previous:** [Queuing Disciplines](#)

Congestion Avoidance Mechanisms

It is important to understand that TCP's strategy is to control congestion once it happens, as opposed to trying to avoid congestion in the first place. In fact, TCP repeatedly increases the load it imposes on the network in an effort to find the point at which congestion occurs, and then backs off from this point. Said another way, TCP *needs* to create losses to find the available bandwidth of the connection. An appealing alternative, but one that has not yet been widely adopted, is to predict when congestion is about to happen, and to reduce the rate at which hosts send data just before packets start being discarded. We call such a strategy congestion *avoidance*, to distinguish it from congestion *control*.

This section describes three different congestion avoidance mechanisms. The first two take a similar approach---they put a small amount of additional functionality into the router to assist the end node in the anticipation of congestion. The third mechanism is very different from the first two---it attempts to avoid congestion purely from the end nodes.

DECbit

RED Gateways

Source-Based Congestion Avoidance

Virtual Clock

We conclude our discussion of congestion control by considering a mechanism--- *virtual clock*---that is very different from those described in the previous sections. Instead of using feedback and being window-based, the virtual clock algorithm allows the source to make a reservation using a rate-based description of its needs. This approach takes us beyond the best-effort delivery model of the current Internet by allowing the source to ask the routers of the network to guarantee it a certain amount of bandwidth. Note that we are using the virtual clock algorithm as a representative example of rate/reservation-based systems currently being designed and evaluated. Section [4.4](#) discusses related efforts in this area.

The idea of the virtual clock algorithm is quite elegant. It is modeled after time division multiplexing (TDM), in which each flow is given a slice of time during which it can transmit on a shared link. The advantage of TDM is that each flow is guaranteed its share of the link's bandwidth, and flows are not allowed to interfere with each other. The disadvantage, however, is that TDM is not able to give one flow's unused bandwidth to another flow that has data to send. Since many applications do not transmit data at a constant bit rate, but instead have bursty transmissions, this is overly conservative. The virtual clock algorithm mimics the TDM behavior, but does so for a statistically multiplexed network by using a virtual definition of time to pace the transmission of data, rather than the real clock used by TDM.

Defining Rate

Virtual Clock as a Queuing Discipline

Virtual Clock as a Flow Meter

Other Rate-Based Mechanisms (ABR)

Copyright 1996, Morgan Kaufmann Publishers

Summary

As we have just seen, the issue of congestion control is not only central to computer networking, but it is also a very hard problem. Unlike flow control, which is concerned with the source not overrunning the receiver, congestion control is about keeping the source from over loading the network. In this context, most of our focus has been on mechanisms targetted at the best-effort service model of today's Internet, where the primary responsibility for congestion control falls on the end nodes of the network. Typically, the source uses feedback---either implicitly learned from the network or explicitly sent by a router---to adjust the load it places on the network; this is precisely what TCP's congestion control mechanism does.

Independent of exactly what the end nodes are doing, the routers implement a queuing discipline that governs what packets get transmitted and what packets get dropped. Sometimes this queuing algorithm is sophisticated enough to segregate traffic (e.g., FQ), and in other cases, the router attempts to monitor its queue length and signal the source host when congestion is about to occur (e.g., RED and DECbit).

As suggested throughout the chapter, it is also possible to design a rate-based congestion control mechanism in which source hosts reserve a desired quality of service from the network. The virtual clock mechanism is an example of a congestion control mechanism designed in this mold, but it is not appropriate for a best-effort service model. We will continue this discussion in Chapter [10](#), where we introduce a new service model that supports multiple qualities of service, including the ability to reserve network bandwidth.

Open Issue: Inside versus Outside the Network

Perhaps the larger question we should be asking is how much can we expect from the network, and how much responsibility will ultimately fall to the end hosts. The emerging reservation-based strategies certainly have the advantage of providing for more varied qualities of service than today's feedback-based schemes, and being able to support different qualities of service is a strong reason to put more functionality into the network's routers. Does this mean that the days of TCP-like end-to-end congestion control are numbered? The answer is not so obvious. It is actually a fairly unlikely scenario that all the routers in a world-wide, heterogeneous network like the Internet will implement precisely the same resource reservation algorithm. Ultimately, it seems that the end-points are going to have to look out for themselves, at least to some extent. After all, we should not forget the sound design principle underlying the Internet---do the simplest possible thing in the routers, and put all the smarts at the edges where you can control it. How this all plays out in the next few years will be very interesting, indeed.

Beyond this basic question, there is another thorny issue lurking in the weeds. While in the very first section of this chapter we claimed that congestion control in a network and in an internet are the same problem---and they are---the issue becomes messy when the problem is being solved at *both* the network level and the internet level. Consider the case of running IP over ATM. If we try to program our IP routers to do something clever regarding resource allocation, then we could be in trouble if one of the underlying links is really implemented by an ATM virtual path, and the ATM network is sharing that path with other traffic. This is because it is impossible for IP to promise some share of a link's bandwidth to a flow if the total bandwidth available on the link is variable. In a sense, there is a tug-of-war between ATM and IP over who owns the resource allocation problem.

Further Reading

The recommended reading list for this chapter is long, reflecting the breadth of interesting work being done in congestion control. It includes the original papers introducing the various mechanisms discussed in this chapter. In addition to a more detailed description of these mechanisms, including thorough analysis of their effectiveness and fairness, these papers are must reading because of the insights they give into the interplay of the various issues related to congestion control. In addition, the first paper gives a nice overview of some of the early work on this topic (i.e., it summarizes the past), and the last paper surveys several emerging rate-based mechanisms (i.e., it gives a possible view of the future).

- M. Gerla and L. Kleinrock. Flow Control: A Comparative Survey. *IEEE Transactions on Communications*, COM-28(4): 553--573, April 1980.
- A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. *Proceedings of the SIGCOMM '89 Symposium*, pages 1--12, September 1989.
- V. Jacobson. Congestion Avoidance and Control. *Proceedings of the SIGCOMM '88 Symposium*, pages 314--329, August 1988.
- K. Ramakrishnan and R. Jain. A Binary Feedback Scheme for Congestion Avoidance in Computer Networks with a Connectionless Network Layer. *ACM Transactions on Computer Systems*, pages 158--181, May 1990.
- S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4), pages 397--413, August 1993.
- L. Brakmo and L. Peterson. TCP Vegas: End-to-End Congestion Avoidance on a Global Internet. *IEEE Journal of Selected Areas in Communication*, 13(8), pages 1465--1480, October 1995.
- L. Zhang. Virtual Clock: A New Traffic Control Algorithm for Packet Switching Networks. *ACM Transactions on Computer Systems*, 9(2): 101--124, May 1991.
- H. Zhang and S. Keshav. Comparison of Rate-Based Service Disciplines. *Proceedings of the*

Beyond these recommended papers, there is a wealth of other valuable material on congestion control. For starters, two early papers by Kleinrock and Jaffe set the foundation for using power as a measure of congestion control effectiveness [[Kle79,Jaf81](#)]. Also, [[Jai91](#)] gives a thorough discussion of various issues related to performance evaluation, including a description of Jain's fairness index.

More details on the various congestion avoidance techniques introduced in Section [\[\]](#) can be found in [[WC92](#)], [[Jai89](#)], and [[WC91](#)], with the first paper giving an especially nice overview of congestion avoidance based on a common understanding of how the network changes as it approaches congestion. Also, the packet-pair technique briefly discussed in Section [\[\]](#) is more carefully described in [[Kes91](#)], and the *partial packet drop* technique suggested in Section [\[\]](#) is described in [[RF94](#)].

Next	Up	Previous
------	----	----------

Next: [Exercises](#) **Up:** [Congestion Control](#) **Previous:** [Open Issue: Inside](#)

[Next](#) [Up](#) [Previous](#)

Next: [High-Speed Networking](#) **Up:** [Congestion Control](#) **Previous:** [Further Reading](#)

Exercises

Copyright 1996, Morgan Kaufmann Publishers

High-Speed Networking

- [Problem: What Breaks When We Go Faster?](#)
 - [Latency Issues](#)
 - [Throughput Issues](#)
 - [Integrated Services](#)
 - [Summary](#)
 - [Open Issue: Realizing the Future](#)
 - [Further Reading](#)
 - [Exercises](#)
-

Problem: What Breaks When We Go Faster?

Performance has been an important aspect of nearly every topic covered in this book. This is for good reason, since it is often improvements in network performance that enable new classes of applications. With this continual push towards higher and higher speed networks, one has to wonder if the abstractions, mechanisms, architectures, and protocols we have been designing for the past 20 years will continue to work, or if instead, some part of our fundamental infrastructure breaks when we go faster.

First, we need to define what we mean by “faster”. While it is likely that multi-terabit networks will one day be a reality, we aim our sights a bit lower by focusing on the *gigabit* networks that are now on the horizon. When you consider that most of us are still using 10Mbps Ethernets and 1.5Mbps long-haul T1 links, 1Gbps easily qualifies as “high-speed”. At three orders of magnitude faster than a T1 link, 1Gbps is enough bandwidth to force us to start thinking in terms of “what happens in the limit”, that is, what happens when an infinite amount of bandwidth is available.

So, what does happen at gigabit speeds? For one thing, the tradeoff between bandwidth and latency changes. While it is tempting to think of “high-speed” as meaning that latency improves at the same rate as bandwidth, we cannot forget about the limitation imposed by the speed of light---the round-trip time for a 1Gbps link is exactly the same as the round-trip time for a 1Mbps link. Instead of being bandwidth poor, a situation that sometimes causes us to ignore the effects of round-trip times on the application, we are bandwidth rich and each RTT becomes more precious. The first section discusses the resulting shift in focus from preserving bandwidth to tolerating and hiding latency.

A second thing that happens is that the bandwidth available on the network starts to rival the bandwidth available *inside* the computers that are connected to the network. What we mean by this is that the network bandwidth is within an order of magnitude of the memory and bus bandwidth of the host. If we are not careful in how we design the end hosts---including both the host's architecture and the software running on that host---then we are not going to be able to deliver the gigabit bandwidth that the network provides all the way to the *end* applications that want to use it. The second section discusses this situation in more detail, and introduces various techniques that have been proposed to deal with this problem.

A final thing that happens as the network becomes faster is that new applications, some never before imagined, start to emerge. While this is good news, quite often these applications need services from the

network beyond the ability to deliver bits at high rates. For example, high-speed networks are making video applications possible, but video needs performance guarantees, not just raw performance. New demands such as these often stress the network's underlying service model, a model that was developed for applications that ran at lower speeds. The third section of this chapter describes a new service model that is just now evolving, and may one day replace the current best-effort service model.

[Next](#) [Up](#) [Previous](#)

Next: [Latency Issues](#) **Up:** [High-Speed Networking](#) **Previous:** [High-Speed Networking](#)

Copyright 1996, Morgan Kaufmann Publishers

Latency Issues

Although gigabit networks bring a dramatic change in the bandwidth available to applications---as much as 1000 times the bandwidth available on today's networks---in many respects their impact on how we think about networking comes in what does *not* change as bandwidth increases: the speed of light. That is, ``high-speed" does not mean that latency improves at the same rate as bandwidth; the cross-country RTT of a 1Gbps link is the same 100ms as it is for a 1Mbps link. This section considers the ramifications of this law of nature.

Latency/Throughput Tradeoff

Implications

Throughput Issues

Being able to deliver data at 622Mbps or 1.2Gbps across the links and switches of a high-speed network is not anything to brag about if that data comes trickling into the application program on the destination host at Ethernet speeds. Why might this happen? Because there are several things that can go wrong between the time network data hits the back of a host, and an application program running on that host is able to read and process that data.

While it may seem that computers are getting faster as fast as networks are getting faster, and therefore delivering data to the application should not be a problem, it is important to remember that it is the computer's processor that is getting faster, not its memory. In fact, all of the problems in turning good network throughput into equally good application-to-application throughput can be traced to the host's *memory bandwidth*---the rate at which data can be moved between the host's memory and its CPU registers. This section first explains the memory bandwidth problem in more detail, and then introduces several techniques that have been proposed to help mitigate the effects of limited memory bandwidth. In many respects, the issues discussed in this section lie at the intersection of computer architecture, operating systems, and computer networks.

Memory Bottleneck

Techniques for Avoiding Data Transfers

Integrated Services

One of the things that has happened as networks have become faster is that people have started to build applications that make use of the speed, not just to do the same old things faster, but to do completely new things. In particular, increased bandwidths have enabled applications that send voice and video information. Video is a particularly high consumer of network bandwidth, while voice or audio information can be tolerably transmitted over 64Kbps channels---not high speed by today's standards, but faster than early wide-area network links. Also, both video and audio have increasing bandwidth requirements as the quality goes up, so that CD-quality audio might require closer to a megabit per second, while one can imagine using tens to hundreds of megabits per second to transmit very high quality video.

There is more to transmitting audio and video over a network than just providing sufficient bandwidth, however. Participants in a telephone conversation, for example, expect to be able to converse in such a way that one person can respond to something said by the other and be heard almost immediately. Thus, the timeliness of delivery can be very important. We refer to applications that are sensitive to the timeliness of data as *real-time* applications. Voice and video applications tend to be the canonical examples, but there are others such as industrial control---you would like a command sent to a robot arm to reach it before the arm crashes into something.

The most important thing about real-time applications is that they need some sort of assurance *from the network* that data is likely to arrive on time (for some definition of ``on time"). Whereas a non-real-time application can use an end-to-end retransmission strategy to make sure that data arrives *correctly*, such a strategy cannot provide timeliness: it only adds to total latency if data arrives late. Timely arrival must be provided by the network itself (by the routers), not just at the networks edges (by the hosts). What we conclude, therefore, is that the best-effort model, in which the network tries to deliver your data but makes no promises, and leaves the cleanup operation to the edges, is not sufficient for real-time applications. What we need is a new service model, in which applications that need higher assurances can ask the network for it. The network may then respond by providing an assurance that it will do better, or perhaps by saying that it cannot promise anything better at the moment. Note that such a service model is a superset of today's: applications that are happy with best effort service should be able to use the new service model; their requirements are just less stringent. This implies that the network

will treat some packets differently from others, something that is not done in the best-effort model.

Model

At the time of writing, the IETF is working towards standardization of a set of extensions to the best effort service model of today's Internet. The idea behind standardizing a new service model is to enable networking equipment manufacturers to implement the new model and application writers to build new applications that use the new model, where everyone can have a common set of expectations about what network provides. The rest of this section motivates and develops the new model, while Section [1.4](#) looks at the mechanisms that will be required to implement the model.

The first part of the model is that we now have two different types of applications: real-time and non-real-time. The latter are sometimes called "traditional data" applications, since they have traditionally been the major applications found on data networks. These include most popular applications like Telnet, FTP, e-mail, Web-browsing, and so on. All of these applications can work without guarantees of timely delivery of data. Another term for this non-real-time class of applications is *elastic*, since they are able to stretch gracefully in the face of increased delay. Note that these applications can benefit from short delays, but that they do not become unusable as delays increase. Also note that their delay requirements vary from the interactive applications like Telnet to more asynchronous ones like e-mail, with interactive bulk transfers like FTP in the middle.

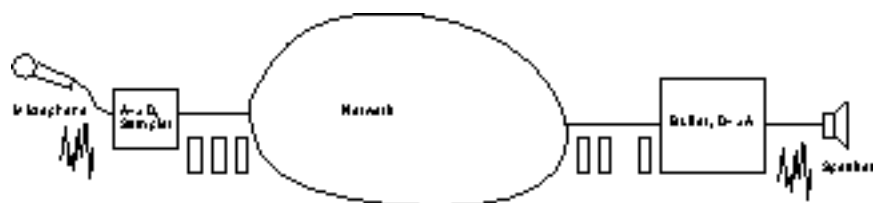



Figure: An audio application

Let's look more closely at the real-time applications. As a concrete example, we consider an audio application, as illustrated in Figure [1.4](#). Data is generated by collecting samples from a microphone and digitizing them using an analog-to-digital (A→D) converter. The digital samples are placed in packets which are transmitted across the network and received at the other end. At the receiving host, the data must be *played back* at some appropriate rate. For example, if the voice samples were collected at a rate of one per 125 μ s, they should be played back at the same rate. Thus, we can think of each sample as having a particular *playback time*: the point in time at which it is needed in the receiving host. In the voice example, each sample has a playback time that is 125 μ s later than the preceding sample. If data arrives after its appropriate playback time, either because it was delayed in the network or because it was dropped and subsequently retransmitted, it is essentially useless. It is the complete worthlessness of late data that characterizes real-time applications. In elastic applications, it might be nice if data turns up on time, but we can still use it when it does not.

One way to make our voice application work would be to make sure that all samples take exactly the same amount of time to traverse the network. Then, since samples are injected at a rate of one per 125 μ s, they will appear at the receiver at the same rate, ready to be played back. However, it is generally difficult to guarantee that all data traversing a packet switched network will experience exactly the same delay. Packets encounter queues in switches or routers, and the lengths of these queues vary with time, meaning that the delays tend to vary with time, and as a consequence, are potentially different for each packet in the audio stream. The way to deal with this at the receiver is to buffer up some amount of data in reserve, thereby always providing a store of packets waiting to be played back at the right time. If a packet is delayed by a small amount, it goes in the buffer until its playback time arrives. If it gets delayed by a large amount, then it will not need to be stored for very long in the receiver's buffer before being played back. Thus, we have effectively added a constant offset to the playback time of all packets as a form of insurance. We call this offset the *playback point*. The only time we run into trouble is if packets get delayed in the network by such a large amount that they arrive after their playback time, causing the playback buffer to be drained.

The operation of a playback buffer is illustrated in Figure . The left-hand line shows packets being generated at a steady rate. The wavy line shows when the packets arrive, some variable amount of time after they were sent, depending on what they encountered in the network. The right hand-line shows the packets being played back at a steady rate, after sitting in the playback buffer for some period of time. As long as the playback line is far enough to the right in time, the variation in network delay is never noticed by the application. However, if we move the playback line a little to the left, then some packets would have arrived too late to be useful.

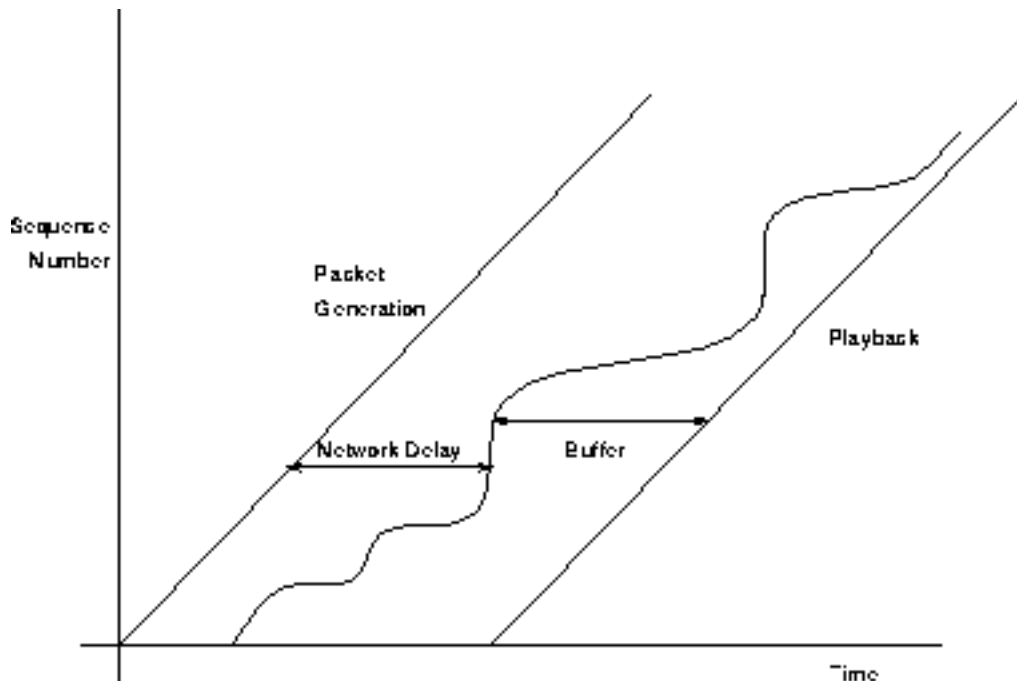


Figure: A playback buffer

For our audio application, there are limits to how far we can delay playing back data. It is hard to carry

on a conversation if the time between when you speak and when your listener hears you is more than three hundred milliseconds. Thus, what we want from the network in this case is a guarantee that all our data will arrive within 300ms. If data arrives early we buffer it until its correct playback time. If it arrives late, we have no use for it, and must discard it.

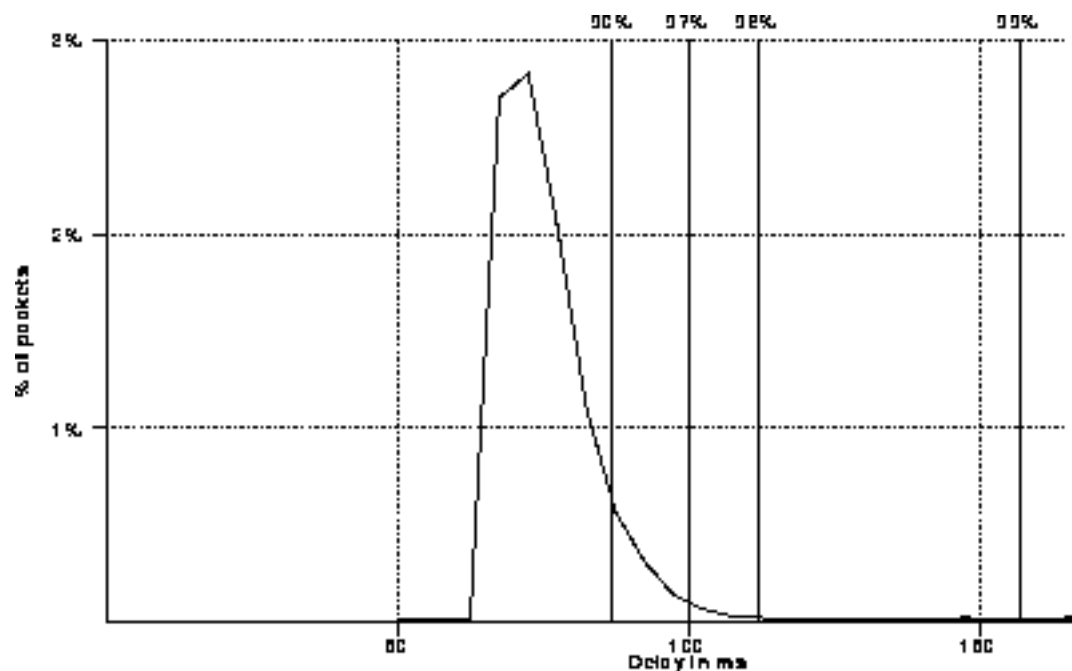



Figure: Example distribution of delays for an Internet connection.

To get a better appreciation of how variable network delay can be, Figure  shows the measured one-way delay across the Internet on one particular day in 1995. As denoted by the cumulative percentages given across the top of the graph, 97% of the packets have a latency of 100ms or less. This means that if our example audio application were to set the playback point at 100ms, then on average, three out of every 100 packets would arrive too late to be of any use. One important thing to notice about this graph is that the *tail* of the curve---how far it extends to the right---is very long. One would have to set the playback point at over 200ms to ensure that all packets arrived in time.

Taxonomy of Real-time Applications

Now that we have a concrete idea of how real-time applications work, we can look at some different classes of applications, which serve to motivate our service model. The following taxonomy owes much to the work of Clark, Shenker and Zhang, whose original paper on this subject can be found in the reading list for this Chapter.

The first characteristic by which we can categorize applications is their tolerance of loss of data, where "loss" might occur because a packet arrived too late to be played back. On the one hand, one lost audio sample can be interpolated from the surrounding samples with relatively little effect on the perceived audio quality. It is only as more and more samples are lost that quality declines to the point that the speech becomes incomprehensible. On the other hand, a robot control program is likely to be an

example of a real-time application that cannot tolerate loss---losing the packet that contains the command instructing the robot arm to stop is not a good thing. Thus, we can categorize real-time applications as *tolerant* or *intolerant* depending on whether they can tolerate occasional loss. As an aside, note that many real-time applications are more tolerant of occasional loss than non-real-time applications. For example, compare our audio application to FTP, where the uncorrected loss of one bit might render a file completely useless.

A second way to characterize real-time applications is by their adaptability. For example, an audio application might be able to adapt to the amount of delay that packets experience as they traverse the network. If at one point in time, we notice that packets are almost always arriving within 300 ms of being sent, then we could set our playback point accordingly, buffering any packets that arrive in less than 300 ms. Suppose that we subsequently observe that all packets are arriving within 100 ms of being sent. If we moved up our playback point to 100 ms, then the users of the application would probably perceive an improvement. The process of shifting the playback point would require us actually to play out samples at an increased rate for some period of time. With a voice application, this can be done in a way that is barely perceptible, simply by shortening the silences between words. Thus, playback point adjustment is fairly easy in this case, and has been effectively implemented for several voice applications such as the audio teleconferencing program known as VAT. Note that playback point adjustment can happen in either direction, but that doing so actually involves distorting the played back signal during the period of adjustment, and that the effects of this distortion will very much depend on how the end user uses the data. Intolerant applications will not in general be able to tolerate this distortion any more than they can tolerate loss.

Observe that if we set our playback point on the assumption that all packets will arrive within 100ms and then find that some packets are arriving slightly late, we will have to drop them, whereas we would not have had to drop them if we had left the playback point at 300ms. Thus, we should advance the playback point only when it provides a perceptible advantage and only when we have some evidence that the number of late packets will be acceptably small. We may do this because of observed recent history or because of some assurance from the network.

We call applications that can adjust their playback point *delay-adaptive* applications. Another class of adaptive applications are *rate-adaptive*. For example, many video coding algorithms can trade off bit-rate versus quality. Thus, if we find that the network can support a certain bandwidth, we can set our coding parameters accordingly. If more bandwidth later becomes available, we could change parameters to increase the quality. While intolerant applications will not tolerate the distortion of delay-adaptivity, they may be able to take advantage of rate-adaptivity.

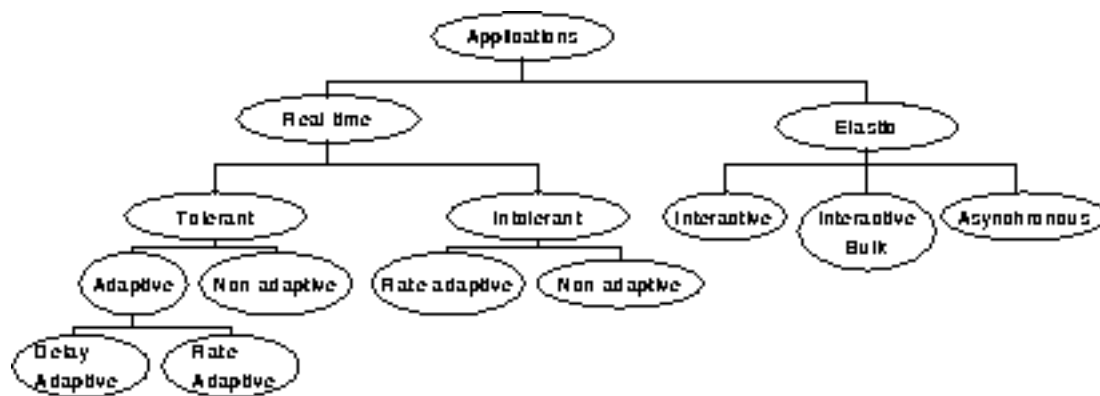


Figure: Taxonomy of applications

To summarize, we have the following taxonomy of applications, as illustrated in Figure [1](#). First, we have the elastic and the real-time, with a range of target delays for elastic applications. Within real-time, we have the intolerant, which cannot accept loss or lateness of data, and the tolerant. We also find adaptive and non-adaptive real-time applications, which may in turn be rate-adaptive or delay-adaptive. What the Internet and most other networks provide today is a service model that is adequate only for elastic applications. What we need is a richer service model to meet the needs of any application in this taxonomy. This leads us to a service model with not just one class (best-effort), but with several classes, each to meet the needs of some set of applications.

Service Classes

For starters, consider a service class to support intolerant applications. These applications require that a packet never turn up late. The network should guarantee that the maximum delay that any packet will experience has some specified value, and the application can then set its playback point so that no packet will ever arrive after its playback time. We assume that early arrival of packets can always be handled by buffering. This service is generally referred to as a *guaranteed* service.

Now consider an application that is tolerant of occasional lateness or loss, but is not adaptive. This application may be willing to take a statistical gamble that some fraction of its packets arrive late, since it knows that it can still deliver acceptable performance. Nevertheless, it needs some information about the likely delay that packets will experience in the network. Unlike the guaranteed service, this is an "honest estimate" of maximum delay, which may occasionally be exceeded. It is not a hard upper bound. The intention of such a service is that it might allow more efficient use of network resources if an occasional late packet is allowed. This service class is often called *predictive*, since the maximum delay is predicted (perhaps wrongly on occasions) rather than guaranteed.


Delay-adaptive applications can also use predictive service. They might initially set their playback point based on an estimate of maximum delay provided by the network and then adapt to the observed behavior as time goes by. We can imagine that by adapting in this way, the application might perform better by moving to lower delays when the specified bound is too conservative, but backing off to longer delays when it is necessary.

We can also imagine another service class for delay-adaptive applications. Since they can adapt to observed delay, we could use a service that offers no quantitative specification of delay, but simply attempts to control it in such a way that a delay-adaptive service will function reasonably well. The main criterion for such a service is that delay is controlled in such a way that the tail of the delay distribution is not too long, that is, we would like the vast majority of our packets to turn up within a bound that is useful to the application. At the time of writing, the IETF is standardizing such a *controlled delay* service which will offer three levels of delay service; the goal is to provide a choice of delay distributions where the maximum delay for each class is about an order of magnitude different from the neighboring class(es).

Finally, we need a service class for rate-adaptive applications. At the time of writing, this is an area of ongoing research.

Mechanisms (RSVP)

Now that we have augmented our best-effort service model with several new service classes, the next question is how we implement a network that provides these services to applications. This section outlines the key mechanisms. Keep in mind while reading this section that the mechanisms being described are still rough; they are still being hammered out by the Internet community. The main thing to take away from the discussion is a general understanding of the pieces involved in supporting the service model outlined above.

First, whereas with a best effort service we can just tell the network where we want our packets to go and leave it at that, a real-time service implies that we tell the network something more about the sort of service we require. We may give it qualitative information such as "use a controlled delay service" or quantitative information such as "I need a maximum delay of 100ms." As well as saying what we want, we need to tell the network something about what we are going to inject into it, since a low bandwidth application is going to require fewer network resources than a high bandwidth application. The set of information that we provide to the network is referred to as a *flowspec*. This name comes from the notion that a set of packets associated with a single application and which share common requirements is called a flow. This is consistent with our use of the term "flow" in Chapter .

Second, when we ask the network to provide us with a particular service, the network needs to decide if it can in fact provide that service. For example, if 10 users ask for a service where each will consistently use 2 Mbps of link capacity, and they all share a link with 10 Mbps capacity, the network will have to say no to some of them. The process of deciding when to say no is called *admission control*.

Third, we need a mechanism by which the users of the network and the components of the network itself exchange information such as requests for service, flowspecs, and admission control decisions. This is

called *signalling* in the ATM world, but since this word has several meanings, we refer to this process as *resource reservation*, and it is achieved using a resource reservation protocol.

Finally, when flows and their requirements have been described, and admission control decisions have been made, the network switches and routers need to meet the requirements of the flows. A key part of meeting these requirements is managing the way packets are queued and scheduled for transmission in the switches and routers. The last mechanism we discuss is *packet scheduling*.

Flowspecs

There are two separable parts to the flowspec: the part that describes the flow's traffic characteristics (called the *Tspec*) and the part that describes the service requested from the network (the *Rspec*). The *Rspec* is very service specific and relatively easy to describe. For example, with a controlled delay service, the *Rspec* might just be a number describing the level of delay required (1, 2 or 3). With a guaranteed or predictive service, one could specify a delay target or bound. (In the IETF's current guaranteed service specification, one specifies not a delay but another quantity from which delay can be calculated.)

The *Tspec* is a little more complicated. As our example above showed, we need to give the network enough information about the bandwidth used by the flow to allow intelligent admission control decisions to be made. For most applications, however, the bandwidth is not a single number; it is something that varies constantly. A video application, for example, will generally generate more bits per second when the scene is changing rapidly than when it is still. Just knowing the long term average bandwidth is not enough, as the following example illustrates. Suppose we have 10 flows that arrive at a switch on separate input ports and that all leave on the same 10 Mbps link. Assume that over some suitably long interval each flow can be expected to send no more than 1 Mbps. You might think that this presents no problem. However, if these are variable bit-rate applications, such as compressed video, then they will occasionally send more than their average rates. If enough sources send at above their average rates, then the total rate at which data arrives will be greater than 10 Mbps. This excess data will be queued before it can be sent on the link. The longer this condition persists, the longer the queue gets. At the very least, while data is sitting in a queue, it is not getting closer to its destination, so it is being delayed. If delayed long enough, the service that was requested will not be provided. In addition, as the queue length grows, at some point we run out of buffer space, and packets must be dropped.

Exactly how we manage our queues to control delay and avoid dropping is something we discuss below. However, note here that we need to know something about how the bandwidth of our sources varies with time. One way to describe the bandwidth characteristics of sources is called a *token bucket* filter. Such a filter is described by two parameters: a token rate, **r**, and a bucket depth, **B**. It works as follows: to be able to send a byte, I must have a token. To send a packet of length **n**, I need **n** tokens. I start with no tokens and I accumulate them at a rate of **r** per second. I can accumulate no more than **B** tokens. What this means is that I can send a burst of as many as **B** bytes into the network as fast as I want, but that over a sufficiently long interval, I can't send more than **r** bytes per second. It turns out that this

information is very helpful to the admission control algorithm when it tries to figure out whether it can accommodate a new request for service.

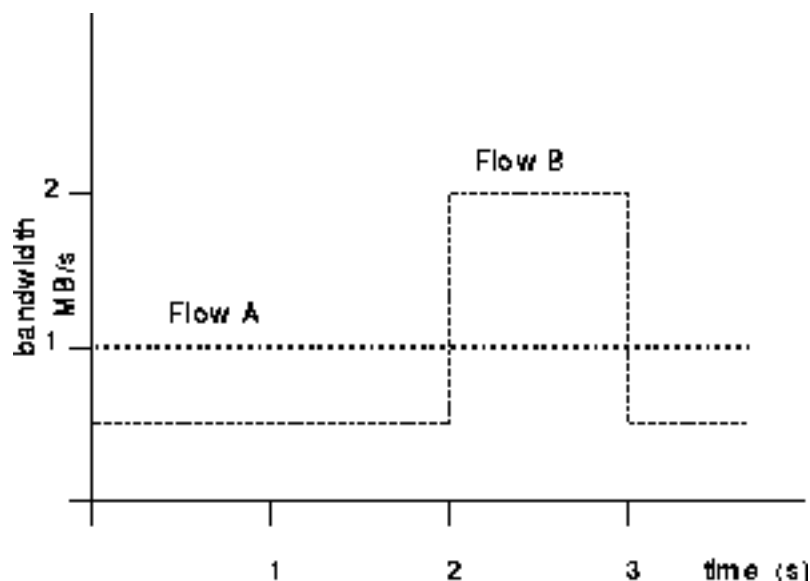



Figure: Two flows with equal average rates but different token bucket descriptions


Figure  illustrates how a token bucket can be used to characterize a flow's bandwidth requirements. For simplicity, assume that each flow can send data as individual bytes, rather than as packets. Flow A, represented by the horizontal line, generates data at a steady rate of 1 MBps, so it can be described by a token bucket filter with a rate $r = 1\text{MBps}$ and a bucket depth of 1 byte. This means that it receives tokens at a rate of 1MBps but cannot store more than 1 token---it spends them immediately, Flow B also sends at a rate which averages out to 1MBps over the long term, but does so by sending at 0.5MBps for 2 seconds and at 2MBps for 1 second. Since the token bucket rate r is, in a sense, a long term average rate, Flow B can be described by a token bucket with a rate of 1MBps. Unlike Flow A, however, Flow B needs a bucket depth B of at least 1MB, so that it can store up tokens while it sends at less than 1MBps, to be used when it sends at 2MBps. For the first 2 seconds in this example, it receives tokens at a rate of 1MBps but spends them at only 0.5 MBps, so it can save up $2 \times 0.5 = 1\text{MB}$ of tokens, which it then spends in the third second (along with the new tokens that continue to accrue in that second) to send data at 2MBps. At the end of the third second, having spent the excess tokens, it starts to save them up again by sending at 0.5 MBps again.

It is interesting to note that a single flow can be described by many different token buckets. As a trivial example, Flow A could be described by the same token bucket as Flow B, with a rate of 1MBps and a bucket depth of 1MB. The fact that it never actually needs to accumulate tokens does not make that an inaccurate description, but it does mean that we have failed to convey some useful information to the network---the fact that Flow A is actually very consistent in its bandwidth needs.

Admission Control

The idea behind admission control is simple: when some new flow wants to receive a particular level of

service, admission control looks at the Tspec and Rspec of the flow and tries to decide if the desired service can be provided to that amount of traffic given the currently available resources, without causing any previously flow to receive worse service than it requested. If it can, the flow is admitted; if not, then it is denied. The hard part is figuring out when to say yes and when to say no.

Admission control is very dependent on the type of requested service, and on the queuing discipline employed in routers; we discuss the latter topic later in this section. For a guaranteed service, you need a good algorithm to make a definitive yes/no decision. This is fairly straightforward if weighted fair queuing (WFQ), as discussed in Section , is used at each router. For a predictive or controlled delay service, the decision may be based on heuristics, like "the last time I allowed a flow with this Tspec into this class, the delays for the class exceeded the acceptable bound, so I'd better say no" or "my current delays are so far inside the bounds that I should be able to admit another flow without difficulty."

Admission control should not be confused with *policing*. The former is a per-flow decision to admit a new flow or not. The latter is a function applied on a per-packet basis to make sure that a flow conforms to the Tspec that was used to make the reservation. If a flow does not conform to its Tspec---for example, because it is sending twice as many bytes per second as it said it would---this is likely to interfere with the service provided to other flows, so some corrective action must be taken. There are several options, the obvious one being to drop offending packets. However, one option would be to check if the packets really are interfering with the service of others. If not, they could be sent on after being marked with a tag that says, in effect, "this is a non-conforming packet---drop me first if you need to drop any packets."

Reservation Protocol

While connection-oriented networks have always needed some sort of setup protocol to establish the necessary virtual circuit state in the switches, connectionless networks like the Internet have had no such protocols. As this section has indicated, however, we need to provide a lot more information to our network when we want a real-time service from it. While there have been a number of setup protocols proposed for the Internet, the one on which most current attention is focused is called RSVP (Resource reSerVation Protocol). It is particularly interesting because it differs so substantially from conventional signalling protocols for connection-oriented networks.

One of the key assumptions underlying RSVP is that it should not detract from the robustness that we find in today's connectionless networks. Because connectionless networks rely on little or no state being stored in the network itself, it is possible for routers to crash and reboot and links to go up and down while end-to-end connectivity is still maintained. RSVP tries to maintain this robustness by using the idea of *soft state* in the routers. Soft state---in contrast to the hard state found in connection-oriented networks---does not need to be explicitly deleted when it is no longer needed. Instead, it times out after some fairly short period (say a minute) if it is not periodically refreshed. We will see later how this helps robustness.

Another important characteristic of RSVP is that it treats multicast as a first-class citizen. This is not surprising, since the multicast applications found on the MBone, such as VAT and NV are obvious early candidates to benefit from real-time services. One of the insights of RSVP's designers is that most multicast applications have many more receivers than senders, as typified by the large audience and one speaker for a lecture carried on the MBone. Also, receivers may have different requirements. For example, one receiver might want to receive data from only one sender, while others might wish to receive data from all senders. Rather than have the senders keeping track of a potentially large number of receivers, it makes more sense to let receivers keep track of their own needs. This suggests the *receiver-oriented* approach adopted by RSVP. In contrast, connection-oriented networks usually leave resource reservation to the sender, just as it is normally the originator of a phone call who causes resources to be allocated in the phone network.

The soft-state and receiver-orientation of RSVP give it a number of nice properties. One nice property is that it is very straightforward to increase or decrease the level of resource allocation provided to a receiver. Since each receiver periodically sends refresh messages to keep the soft state in place, it is easy to send a new reservation which asks for the new level of resources. In the event of a host crash, resources allocated by that host to a flow will naturally time out and be released. To see what happens in the event of a router or link failure, we need to look a little more closely at the mechanics of making a reservation.

Initially, consider the case of one sender and one receiver trying to get a reservation for traffic flowing between them. There are two things that need to happen before a receiver can make the reservation. First, the receiver needs to know what traffic the sender is likely to send so that it can make an appropriate reservation. That is, it needs to know the sender's Tspec. Second, it needs to know what path packets will follow from sender to receiver, so that it can establish a resource reservation at each router on the path. Both of these requirements can be met by sending a message from the sender to the receiver that contains the Tspec. Obviously, this gets the Tspec to the receiver. The other thing that happens is that each router looks at this message, called a PATH message, as it goes past, and figures out the *reverse path* that will be used to send reservations from a receiver back to a sender in an effort to get the reservation to each router on the path. Building the multicast tree in the first place is done by mechanisms such as those described in Section [4.1](#).

Having received a PATH message, the receiver sends a reservation back "up" the multicast tree in a RESV message. This message contains the sender's Tspec and an Rspec describing the requirements of this receiver. Each router on the path looks at the reservation request and tries to allocate the necessary resources to satisfy it. If the reservation can be made, the RESV request is passed on to the next router. If not, an error message is returned to the receiver who made the request. (Actually, a range of options involving forwarding messages that failed at one or more routers are being considered by the IETF at present.) If all goes well, the correct reservation is installed at every router between the sender and the receiver. As long as the receiver wants to retain the reservation, it sends the same RESV message about once per 30 seconds.

Now we can see what happens when a router or link fails. Routing protocols will adapt to the failure and create a new path from sender to receiver. PATH messages are sent about every 30 seconds, so the first one after the new route stabilizes will reach the receiver over the new path. The receiver's next RESV message will follow the new path and (hopefully) establish a new reservation on the new path. Meanwhile, the routers that are no longer on the path will stop getting RESV messages and their reservations, and these reservations will time out and be released.

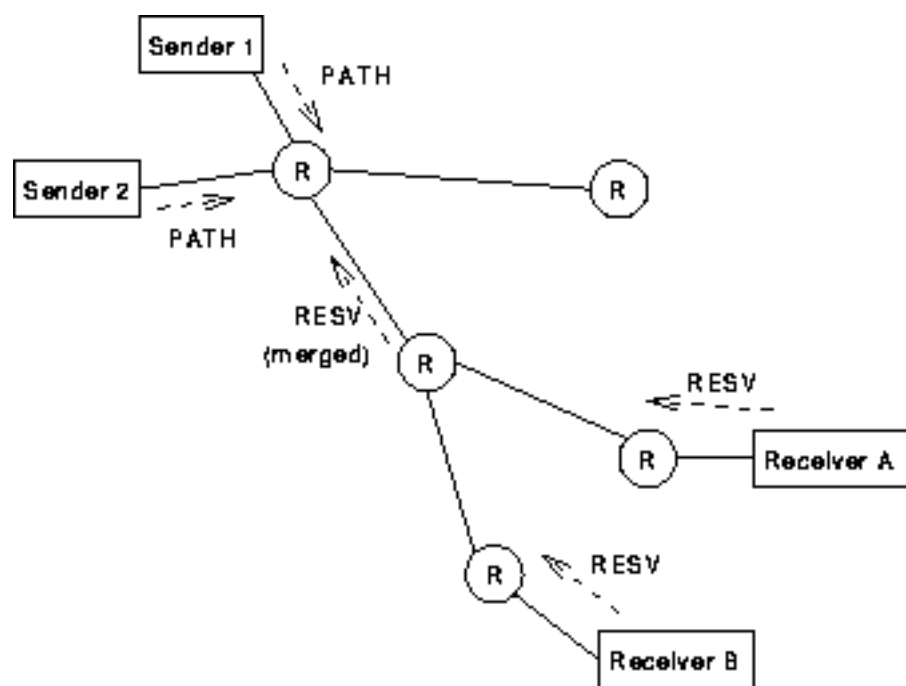


Figure: Making reservations on a multicast tree



The next thing we need to consider is how to cope with multicast, where there are multiple senders to a group, and multiple receivers. This situation is illustrated in Figure [10.10](#). First, let's deal with multiple receivers for a single sender. As a RESV message travels up the multicast tree, it is likely to hit a piece of the tree where some other receiver's reservation has already been established. It may be the case that the resources reserved upstream of this point are adequate to serve both receivers. For example, if receiver A has already made a reservation that provides for a guaranteed delay of less than 100ms, and the new request from receiver B is for a delay of less than 200ms, then no new reservation is required. On the other hand if the new request is for a delay of less than 50ms, the router would first need to see if it can accept the request, and if so, send the request on upstream. The next time receiver A asks for 100ms delay, the router would not need to pass this request on. In general, reservations can be merged in this way to meet the needs of all receivers downstream of the merge point. The only catch is that the rules for merging can be complicated.

If there are also multiple senders to the tree, receivers need to collect the Tspecs from all senders and make a reservation that is large enough to accommodate the traffic from all senders. However, this may not mean that the Tspecs need to be added up. For example, in an audio conference with ten speakers, there is not much point in allocating enough resources to carry ten audio streams, since the result of ten people speaking at once would be incomprehensible. Thus, we could imagine a reservation that is large

enough to accommodate two speakers, and no more. Calculating the correct Tspec from all the sender Tspecs is clearly application-specific. Also, one may only be interested in hearing from a subset of all possible speakers; RSVP has different reservation ``styles" to deal with such options as ``reserve resources for all speakers", ``reserve resources for any **n** speakers", and ``reserve resources for speakers A and B only".

Sidebar: RSVP and ATM

Now that we've seen some highlights of RSVP, it is interesting to compare it with a more ``conventional" signalling protocol from a connection-oriented network. Note that, at a high level, the goals of a connection-oriented signalling protocol and RSVP are the same: to install some state information in the network nodes that forward packets so that packets get handled correctly. However, there are not many similarities beyond that high level goal.

Table  compares RSVP with the ATM Forum's current signalling protocol, which is derived from the ITU-T protocol Q.2931. (Recall from Section  that Q.2931 defines how a virtual circuit is routed across the network, as well as how resources are reserved for that circuit.) The differences stem largely from the fact that RSVP starts with a connectionless model and tries to add functionality without going all the way to traditional connections, whereas ATM starts out from a connection-oriented model. RSVP's goal of treating multicast traffic as a first-class citizen is also apparent in the receiver-driven approach, which aims to provide scalability for multicast groups with large numbers of receivers.

RSVP	ATM
Receiver generates reservation	Sender generates connection request
Soft state (refresh/timeout)	Hard state (explicit delete)
Separate from route establishment	Concurrent with route establishment
QoS can change dynamically	QoS is static for life of connection
Receiver heterogeneity	Uniform QoS to all receivers

Table: Comparison of RSVP and ATM Signalling

Packet Classifying and Scheduling

Once we have described our traffic and our desired network service, and installed a suitable reservation at all the routers on the path, the only thing that remains is for the routers to actually deliver the requested service to the data packets. There are two things that need to be done:

- associate each packet with the appropriate reservation so that it can be handled correctly, a process known as *classifying* packets;
- manage the packets in the queues so that they receive the service that has been requested, a process known as packet *scheduling*.

The first part is done by examining up to four fields in the packet: the source address, destination address, source port and destination port. (In IPv6, it is possible that the `FlowLabel` field in the header could be used to enable lookup to be done based on a single, shorter key.) Based on this information, the packet can be placed in the appropriate class. For example, it may be classified into one of three controlled delay classes, or it may be part of a guaranteed delay flow that needs to be handled separately from all other guaranteed flows. In short, there is a mapping from the flow-specific information in the packet header to a single class identifier that determines how the packet is handled in the queue. For guaranteed flows, this might be a one-to-one mapping, while for other services, it might be many to one. The details of classification are closely related to the details of queue management.

It should be clear that something as simple as a FIFO queue in a router will be inadequate to provide many different services and to provide different levels of delay within each service. Several more sophisticated queue management disciplines were discussed in Section [4.4](#), and some combination of these is likely to be used in a router.

The details of packet scheduling ideally should not be specified in the service model. Instead, this is an area where implementors try to do creative things to realize the service model efficiently. In the case of guaranteed service, it has been established that a weighted fair queueing discipline, in which each flow gets its own individual queue with a certain share of the link, will provide a guaranteed end-to-end delay bound that can readily be calculated. For controlled and predictive delay services, other schemes are being tested. One possibility includes a simple priority scheme in which all flows in a class are placed in one queue, there is one priority level per class, and the lowest delay class has the highest priority. The problem is made harder when you consider that in a single router, many different services are likely to be provided concurrently, and that each of these may require a different scheduling algorithm. Thus, some overall queue management algorithm is needed to manage the resources between the different services.

Copyright 1996, Morgan Kaufmann Publishers

Summary

We asked the question ``what breaks when we go faster" at the beginning of this chapter. The answer is that three things break, but they can all be fixed. The first is that latency, rather than bandwidth, becomes our primary focus of concern. This is because while network bandwidth is improving at an astounding rate, the speed-of-light has remained constant. Fortunately, we can adjust for this by trading bandwidth for latency.

A second thing that breaks is our conception that computers are fast and I/O is slow. Gigabit networks make it necessary to design computer architectures and operating system software with the network foremost in our minds, rather than treating the network as a fast serial line. This is because network speeds are improving to the point that they are within the same order of magnitude of memory speeds. This situation is sometimes referred to as the memory bottleneck, and we examined several techniques to help mitigate its effects.

A third thing that breaks is the current best-effort service model. As networks have become faster, new applications such as real-time video have become possible. Often, however, these new applications need more than just raw speed. They also need a richer service model, for example, one that provides for different qualities of service. This is perhaps the most drastic change that is required. In this context, we looked at the early stages of an effort to extend the Internet to support multiple qualities of service.

Open Issue: Realizing the Future

Although not explicitly stated in this chapter, there is much about the network architectures that have evolved over the past 20 years that does not break when networks move into the gigabit regime. Connectivity is still achieved through a hierarchical interconnection of machines, routers and switches still forward packets from one link to another, and end-to-end protocols still provide high-level communication abstractions to application programs.

What is apparent, though, is that computer networks will continue to evolve. Higher speed technologies will be developed at the bottom of the network architecture, new applications will be introduced at the top of the network architecture, and the protocol layers in the middle will adapt to connect these new applications to the underlying technologies. Perhaps even more predictably, network designers will continue to argue about what functionality belongs at what layer. If nothing else, hopefully this book has imparted the message that computer networking is an active and exciting topic to study, and that it is a sound grasp of system design principles that will allow us to tackle each new challenge.

Further Reading

Our recommended read list for this chapter is long, once again reflecting what an active area of research high-speed networking has become. The first paper gives an excellent discussion of how drastically high-speed networks have changed the latency/bandwidth tradeoff. The next five papers then discuss the issue of turning good host-to-host throughput into equally good application-to-application throughput. The first of these five (Druschel et. al.) gives an overview of the general problem, the next two discuss the host/network interface, and the last two focus on OS techniques to avoid data copying. Finally, the last paper discusses various aspects of the new service model being designed for the Internet.

- L. Kleinrock. The Latency/Bandwidth Tradeoff in Gigabit Networks. *IEEE Communications*, 30 (4): 36--40, April 1992.
- P. Druschel et. al. Network Subsystem Design. *IEEE Network* (Special Issue on End-System Support for High Speed Networks), 7(4): 8--17, July 1993.
- P. Druschel, L. Peterson, and B. Davie. Experience with a High-Speed Network Adaptor: A Software Perspective. *Proceedings of the SIGCOMM '94 Symposium*, pages 2--13, August 1994.
- A. Edwards et. al. User-Space Protocols Deliver High-Bandwidth Performance to Applications on a Low-Cost Gb/s LAN. *Proceedings of the SIGCOMM '94 Symposium*, pages 14--23, August 1994.
- P. Druschel and L. Peterson. Fbufs: A high-bandwidth Cross-Domain Transfer Facility. *Proceedings of the Fourteenth Symposium on Operating Systems Principles*, pages 202--215, December 1993.
- M. Abbott and L. Peterson. Increasing Network Throughput by Integrating Protocol Layers. *IEEE/ACM Transactions on Networking*, 1(5): 600--610, October 1993.
- R. Braden, D. Clark, and S. Shenker. Integrated Services in the Internet Architecture: An Overview. *RFC 1633*, July 1994.

In addition to these papers, Partridge gives an overview of the field of high-speed networking, covering many of the same topics discussed in this (and earlier) chapters [[Par94](#)].

There are several papers that examine OS issues related to efficiently implementing network software on end hosts. Three examples in addition to those enumerated above include [[MB93](#),[TNML93](#),[PAM94](#)].

There are also an assortment of papers that examine the network interface and how it interacts with the end host; examples include [[DPD94](#),[Dav91](#),[TS93](#),[Ram93](#),[EWL94](#),[Met93](#),[KC88](#),[CFFD93](#),[Ste94a](#)].

To learn more on the new service model, see [[Cla92](#)]. For more information on alternative models for handling real-time traffic, see [[Fer90](#),[FV90](#),[FV93](#)].

Finally, we recommend the following live reference:

- <http://netlab.itd.nrl.navy.mil/onr.html>: list of network-related research projects, many of which focus on high-speed networks.

Next	Up	Previous
----------------------	--------------------	--------------------------

Next: [Exercises](#) **Up:** [High-Speed Networking](#) **Previous:** [Open Issue: Realizing](#)

[Next](#) [Up](#) [Previous](#)

Next: [Appendix: Network Management](#) **Up:** [High-Speed Networking](#) **Previous:** [Further Reading](#)

Exercises

Copyright 1996, Morgan Kaufmann Publishers

Appendix: Network Management

A network is a complex system, both in terms of the number of nodes that are involved, and in terms of the suite of protocols that can be running on any one node. Even if you restrict yourself to worrying about the nodes within a single administrative domain, such as a campus, there might be dozens of routers and hundreds---or even thousands---of hosts to keep track of. If you think about all the state that is maintained and manipulated on any one of those nodes---e.g., address translation tables, routing tables, TCP connection state, and so on---then it is easy to become depressed about the prospect of having to manage all of this information.

It is easy to imagine wanting to know about the state of various protocols on different nodes. For example, one might want to monitor the number of IP datagram reassemblies that have been aborted, so as to determine if the timeout that garbage collects partially assembled datagrams needs to be adjusted. As another example, one might want to keep track of the load on various nodes (i.e., the number of packets sent or received) so as to determine if new routers or links need to be added to the network. Of course, one also has to be on the watch for evidence of faulty hardware and mis-behaving software.

What we have just described is the problem of network management, an issue that pervades the entire network architecture. This appendix gives a brief overview of the tools that are available to system administrators to manage their networks.

-
- [SNMP Overview](#)
 - [MIB Variables](#)
-

[Next](#) [Up](#) [Previous](#)

Next: [MIB Variables](#) **Up:** [Appendix: Network Management](#) **Previous:** [Appendix: Network Management](#)

SNMP Overview

[Next](#) [Up](#) [Previous](#)

Next: [Glossary](#) **Up:** [Appendix: Network Management](#) **Previous:** [SNMP Overview](#)

MIB Variables

[Next](#) [Up](#) [Previous](#)

Next: [References](#) **Up:** [Computer Networks: A Systems](#) **Previous:** [MIB Variables](#)

Glossary

Copyright 1996, Morgan Kaufmann Publishers

References

Bar95

John Perry Barlow. Electronic frontier: Death from above. *Communications of the ACM*, 38 (5):17--20, May 1995.

Bat68

K. E. Batcher. Sorting networks and their applications. In *Proc. 1968 Spring AFIPS Joint Computer Conference*, volume 32, pages 307--314, 1968.

BCDB95

M. Borden, E. Crawley, B. Davie, and S. Batsell. Integration of real-time services in an ip-atm network architecture. Request for Comments 1821, August 1995.

BDMS94

C. Mic Bowman, Peter B. Danzig, Udi Manber, and Michael F. Schwartz. Scalable internet resource discovery: Research problems and approaches. *Communications of the ACM*, 37(8):98--107, August 1994.

BG92

Dimitri Bertsekas and Robert Gallager. *Data Networks*. Prentice--Hall, Inc., Englewood Cliffs, NJ, second edition, 1992.

BG93

Mats Bjorkman and Per Gunningberg. Locking effects in multiprocessor implementations of protocols. In *Proceedings of the SIGCOMM '93 Symposium*, pages 74--83, September 1993.

Bla87

Richard E. Blahut. *Principles and Practice of Information Theory*. Addison--Wesley, Reading, MA, 1987.

BLNS82

A. Birrell, R. Levin, R. Needham, and M. Schroeder. Grapevine: An exercise in distributed computing. *Communications of the ACM*, 25:250--273, 1982.

BM95

Scott Bradner and Alison Mankin, editors. *IPng: Internet Protocol Next Generation*. Addison-Wesley, 1995.

Boo95

Paulina Boorsook. How anarchy works. *Wired*, 3(10):110--118, October 1995.

BPY90

Mic Bowman, Larry L. Peterson, and Andrey Yeatts. Univers: An attribute-based name server. *Software---Practice and Experience*, 20(4):403--424, April 1990.

BS88

Lubomir Bic and Alan C. Shaw. *The Logical Design of Operating Systems*. Printice--Hall Inc., Englewood Cliffs, NJ, 1988.

Buf94

John F. Koegel Buford. *Multimedia Systems*. ACM Press and Addison--Wesley, Reading, MA, 1994.

CFFD93

Danny Cohen, Gregory Finn, Robert Felderman, and Annette DeSchon. ATOMIC: A low-cost, very-high-speed, local communications architecture. In *Proceedings of the 1993 Conference on Parallel Processing*, August 1993.

Cha93

A. Lyman Chapin. The billion node internet. In D. C. Lynch and M. T. Rose, editors, *Internet System Handbook*, chapter 17, pages 707--716. Addison--Wesley, Reading, MA, 1993.

CJRS89

David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, 27(6):23--29, June 1989.

Cla82

David D. Clark. Modularity and efficiency in protocol implementation. Request for Comments 817, MIT Laboratory for Computer Science, Computer Systems and Communications Group, July 1982.

Cla85

David D. Clark. The structuring of systems using upcalls. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 171--180, December 1985.

Cla92

D. Clark. Supporting real-time applications in an integrated services packet network: Architecture and mechanism. In *Proceedings of the SIGCOMM '92 Symposium*, pages 14--26, August 1992.

CLNZ89

S. K. Chen, E. D. Lazowska, D. Notkin, and J. Zahorjan. Performance implications of design alternatives for remote procedure call stubs. In *Proceedings of the Ninth International Conference on Distributed Computing Systems*, pages 36--41, June 1989.

CMRW93

J. Case, K. McCloghrie, M Rose, and S. Waldbusser. Structure of management information for version 2 of the simple network managment protocol (snmpv2). Request for Comments 1442, April 1993.

Com95

Douglas E. Comer. *Internetworking with TCP/IP. Volume I: Principles, Protocols, and Architecture*. Prentice--Hall, Inc., Englewood Cliffs, NJ, third edition, 1995.

Cou94

National Research Council. *Realizing the Information Future: The Internet and Beyond*. National Academy Press, Washington, DC, 1994.

CP89

Douglas E. Comer and Larry L. Peterson. Understanding naming in distributed systems. *Distributed Computing*, 3(2):51--60, May 1989.

CS93

Douglas E. Comer and David L. Stevens. *Internetworking with TCP/IP. Volume III: Client--Server Programming and Applications, BSD Socket Version*. Prentice--Hall, Inc., Englewood Cliffs, NJ, 1993.

CS94

Douglas E. Comer and David L. Stevens. *Internetworking with TCP/IP. Volume III: Client--Server Programming and Applications, AT&T TLI Version*. Prentice--Hall, Inc., Englewood Cliffs, NJ, 1994.

CZ85

David R. Cheriton and Willy Zwaenepoel. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems*, 3(2):77--107, May 1985.

Dav91

Bruce S. Davie. A host-network interface architecture for ATM. In *Proceedings of the SIGCOMM '91 Symposium*, pages 307--316, September 1991.

DP93

Peter Druschel and Larry L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 189--202, December 1993.

DPD94

Peter Druschel, Larry L. Peterson, and Bruce S. Davie. Experience with a high-speed network adaptor: A software perspective. In *Proceedings of the SIGCOMM '94 Symposium*, pages 2--13, August 1994.

DY75

R. L. Drysdale and F. H. Young. Improved divide/sort/merge sorting networks. *SIAM Journal on Computing*, 4(3):264--270, September 1975.

EWL+94

A. Edwards, G. Watson, J. Lumley, D. Banks, C. Calamvokis, and C. Dalton. User-space protocols deliver high performance to applications on a low-cost Gb/s LAN. In *Proceedings of the SIGCOMM '94 Symposium*, pages 14--23, October 1994.

Fer90

D. Ferrari. Client requirements for real-time communications services. *IEEE Communications*, 28(11), November 1990.

Fin88

Raphael A. Finkel. *An Operating Systems Vade Mecum*. Printice--Hall Inc., Englewood Cliffs, NJ, 1988.

FLYV93

V. Fuller, T. Li, J. Yu, and K. Varadhan. Classless inter-domain routing (cidr): an address assignment and aggregation strategy. Request for Comments 1519, September 1993.

FV93

D. Ferrari and D. Verma. Distributed delay jitter control in packet-switching internetworks. *Journal of Internetworking: Research and Experience*, 4:1--20, 1993.

FV90

D. Ferrari and D. Verma. A scheme for real-time channel establishment in wide-area networks. *IEEE Journal of Selected Areas in Communication (JSAC)*, 8(3):368--379, April 1990.

GG94

I. Gopal and R. Guerin. Network transparency: The plaNET approach. *IEEE/ACM Transactions on Networking*, 2(3):226--239, June 1994.

Gia94

Edoardo Giagioni. A structured tcp in standard ml. In *Proceedings of the SIGCOMM '94 Symposium*, pages 36--45, August 1994.

Hed88

C. Hedrick. Routing information protocol. Request for Comments 1058, June 1988.

Hei93

J. Heinanen. Multiprotocol encapsulation over atm adaptation layer 5. Request for Comments 1483, July 1993.

HMPT89

N. Hutchinson, S. Mishra, L. Peterson, and V. Thomas. Tools for implementing network protocols. *Software---Practice and Experience*, 19(9):895--916, September 1989.

Hol91

Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice--Hall, Englewood Cliffs, NJ, 1991.

HP95

Gerard J. Holzmann and Bjorn Pehrson. *The Early History of Data Networks*. IEEE Computer Society Press, Los Alamitos, California, 1995.

Huf52

D. A. Huffman. A method for the construction of minimal-redundancy codes. *Proceedings of the IRE*, 40(9):1098--1101, September 1952.

IL94

H. Ishida and L. Landweber. Issue on internet technology. *Communications of the ACM*, 37(8), August 1994.

Jac88

Van Jacobson. Congestion avoidance and control. In *Proceedings of the SIGCOMM '88 Symposium*, pages 314--329, August 1988.

Jaf81

Jeffrey M. Jaffe. Flow control power is nondecentralizable. *IEEE Transaction on Communications*, COM-29(9):1301--1306, September 1981.

Jai89

Raj Jain. A delay-based approach for congestion avoidance in interconnected heterogeneous computer networks. *ACM Computer Communication Review*, 19(5):56--71, October 1989.

Jai91

Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modeling*. John Wiley and Sons, Inc., New York, NY, 1991.

Jai94

Raj Jain. *FDDI Handbook: High-Speed Networking Using Fiber and Other Media*. Addison-Wesley, Reading, MA, 1994.

JBB92

V. Jacobson, R. Braden, and D. Borman. TCP extensions for high performance. Request for Comments 1323, May 1992.

KC88

Hemant Kanakia and David R. Cheriton. The VMP network adapter board (NAB): High-performance network communication for multiprocessors. In *Proceedings of the SIGCOMM '88 Symposium*, pages 175--187, August 1988.

Kes91

S. Keshav. A control-theoretic approach to flow control. In *Proceedings of the SIGCOMM '91 Symposium*, pages 3--15, September 1991.

Kle75

L. Kleinrock. *Queueing Systems. Volume 1: Theory*. Wiley, 1975.

Kle79

L. Kleinrock. Power and deterministic rules of thumb for probabilistic problems in computer communications. In *Proceedings of the International Conference on Communications*, June 1979.

KP91

Phil Karn and Craig Partridge. Improving round-trip time estimates in reliable transport protocols. *ACM Transactions on Computer Systems*, 9(4):364--373, November 1991.

KPS95

Charlie Kaufman, Radia Perlman, and Mike Speciner. *Network Security: Private Communication in a Public World*. Prentice-Hall, Englewood Cliffs, NJ, 1995.

Lau94

M. Laubach. Classical ip and arp over atm. Request for Comments 1577, January 1994.

Lin93

Huai-An (Paul) Lin. Estimation of the optimal performance of ASN.1/BER transfer syntax. *Computer Communications Review*, 23(3):45--58, July 1993.

LMKQ89

Samuel J. Leffler, Marshall K. McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley Publishing Company, 1989.

LTWW94

W. Leland, M. Taqqu, W. Willinger, and D. Wilson. On the self-similar nature of ethernet traffic. *IEEE/ACM Transactions on Networking*, 2:1--15, February 1994.

Mal93

G. Malkin. Rip version 2 carrying additional information. Request for Comments 1388, January 1993.

MB93

Chris Maeda and Brian Bershad. Protocol service decomposition for high-performance networking. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, December 1993.

MD93

P. E. McKenney and K. F. Dove. Efficient demultiplexing of incoming TCP packets. In *Proceedings of the SIGCOMM '92 Symposium*, pages 269--280, August 1993.

Met93

R. Metcalf. Computer/network interface design lessons from arpanet and ethernet. *IEEE Journal of Selected Areas in Communication (JSAC)*, 11(2):173--180, February 1993.

Min93

Daniel Minoli. *Enterprise Networking: Fractional T1 to SONET, Frame Relay to BISDN*. Artech House, Norwood, MA, 1993.

Moy94

J. Moy. Ospf version 2. Request for Comments 1583, March 1994.

MP85

J. Mogul and J. Postel. Internet standard subnetting procedure. Request for Comments 950, August 1985.

MR91

K. McCloghrie and M Rose. Management information base for network managment of tcp/ip-based internets: Mib-ii. Request for Comments 1213, March 1991.

Mul90

Sape Mullender. Amoeba: A distributed operating system for the 1990s. *IEEE Computer Magazine*, 23(5):44--53, May 1990.

Nel92

Mark Nelson. *The Data Compression Book*. M&T Books, San Mateo, CA, 1992.

NYKT94

Erich M. Nahum, David J. Yates, James F. Kurose, and Don Towsley. Performance issues in parallelized network protocols. In *Proceedings of the First USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 125--137, November 1994.

OCD+88

John K. Ousterhout, Andrew R. Cherenson, Federick Dougkis, Michael N. Nelson, and Brent B. Welch. The Sprite network operating system. *IEEE Computer Magazine*, pages 23--36, February 1988.

OP91

S. O'Malley and L. Peterson. Tcp extensions considered harmful. Request for Comments 1263, October 1991.

Ope94

Open Software Foundation. *OSF DCE Application Environment Specification*. Prentice--Hall Inc., Englewood Cliffs, NJ, 1994.

OPM94

Sean W. O'Malley, Todd A. Proebsting, and Allen B. Montz. Universal stub compiler. In *Proceedings of the SIGCOMM '94 Symposium*, pages 295--306, August 1994.

Pad85

M. A. Padlipsky. *The Elements of Networking Style and Other Essays and Animadversions on the Art of Intercomputer Networking*. Prentice--Hall, Inc., Englewood Cliffs, NJ, 1985.

PAM94

Joseph Pasquale, Eric Anderson, and P. Keith Muller. Container shipping: Operating system

support for I/O-intensive applications. *IEEE Computer Magazine*, 27(3):84--93, March 1994.

Par94

Craig Partridge. *Gigabit Networking*. Addison--Wesley, Reading, MA, 1994.

PB61

W. W. Peterson and D. T. Brown. Cyclic codes for error detection. In *Proc. IRE*, volume 49, pages 228--235, January 1961.

Per92

Radia Perlman. *Interconnections: Bridges and Routers*. Addison--Wesley, Reading, MA, 1992.

Per93

Radia Perlman. Routing protocols. In D. C. Lynch and M. T. Rose, editors, *Internet System Handbook*, chapter 5, pages 157--182. Addison-Wesley, Reading, MA, 1993.

Pet88

Larry L. Peterson. The Profile naming service. *ACM Transactions on Computer Systems*, 6(4):341--364, November 1988.

PF94

V. Paxson and S. Floyd. Wide-area traffic: The failure of poisson modeling. In *sigcomm94*, pages 257--268, London, UK, August 1994.

PH90

David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, 1990.

Pry91

Martin De Prycker. *Asynchronous transfer mode: solution for broadband ISDN*. Ellis Horwood, Chichester, England, 1991.

Ram93

K. K. Ramakrishnan. Performance considerations in designing network interfaces. *IEEE Journal on Selected Areas in Communications*, 11(2):203--219, February 1993.

RF89

T. R. N. Rao and E. Fujiwara. *Error-Control Coding for Computer Systems*. Prentice--Hall, Englewood Cliffs, NJ, 1989.

RF94

A. Romanow and S. Floyd. Dynamics of TCP traffic over ATM networks. In *Proceedings of the*

SIGCOMM '94 Symposium, pages 79--88, October 1994.

RL95

Y. Rekhter and T. Li. A border gateway protocol 4 (bgp-4). Request for Comments 1771, March 1995.

Ros86

F. E. Ross. FDDI - a tutorial. *IEEE Communications Magazine*, 24(5):10--17, May 1986.

Ros94

Marshall Rose. *The Simple Book: An Introduction to Internet Management*. Prentice-Hall, 2nd edition, 1994.

Sal78

J. Saltzer. Naming and binding of objects. In *Lect Notes Comput Sci*, volume 60, pages 99--208. Springer, New York, NY, 1978.

SB89

Michael D. Schroeder and Michael Burrows. Performance of Firefly RPC. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 83--90, December 1989.

Sch94

Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., New York, NY, 1994.

Sha48

C. Shannon. A mathematical theory of communication. *Bell Systems Technical Journal*, 27:379--423, 623--656, 1948.

Sho78

J. Shoch. Inter-network naming, addressing, and routing. In *17th IEEE Comput Soc Int Conf (COMPCON)*, pages 72--79, September 1978.

SHP91

John Spragins, Joseph Hammond, and Krzysztof Pawlikowski. *Telecommunications: Protocols and Design*. Addison--Wesley, Reading, MA, 1991.

Sit92

Richard L. Sites. *Alpha Architecture Reference Manual*. Digital Press, 1992.

Sri95a

R. Srinivasan. RPC: Remote procedure call protocol specification version 2. Request for

Comments 1831, August 1995.

Sri95b

R. Srinivasan. XDR: External data representation standard. Request for Comments 1832, August 1995.

Sta90

William Stallings. *Local Networks*. Macmillan Publishing Company, New York, NY, third edition, 1990.

Sta91

William Stallings. *Data and Computer Communications*. Macmillan Publishing Company, New York, NY, third edition, 1991.

Ste94a

Peter A. Steenkiste. A systematic approach to host interface design for high speed networks. *IEEE Computer Magazine*, 27(3):47--57, March 1994.

Ste94b

W. Richard Stevens. *TCP/IP Illustrated. Volume 1: The Protocols*. Addison--Wesley, Reading, MA, 1994.

SW95

W. Richard Stevens and Gary R. Wright. *TCP/IP Illustrated. Volume 2: The Implementation*. Addison--Wesley, Reading, MA, 1995.

Swe95

J. L. Swerdlow. Information revolution. *National Geographic*, 188(4):5--37, October 1995.

T. 93

In T. G. Robertazzi, editor, *Performance Evaluation of High Speed Switching Fabrics and Networks: ATM, Broadband ISDN and MAN Technology*. IEEE Press, Piscataway, NJ, 1993.

Tan88

Andrew S. Tanenbaum. *Computer Networks*. Prentice--Hall, Inc., Englewood Cliffs, NJ, second edition, 1988.

Tan92

Andrew S. Tanenbaum. *Modern Operating Systems*. Printice--Hall Inc., Englewood Cliffs, NJ, 1992.

Ter86

D. Terry. Structure-free name management for evolving distributed environments. In *6th Int Conf on Distributed Computing Systems*, pages 502--508, May 1986.

TL93

Chandramohan A. Thekkath and Henry M. Levy. Limits to low-latency communication on high-speed networks. *ACM Transactions on Computer Systems*, 11(2):179--203, May 1993.

TNML93

C. A. Thekkath, T. D. Nguyen, E. Moy, and E. D. Lazowska. Implementing network protocols at user level. *IEEE/ACM Transactions on Networking*, 1(5):554--565, October 1993.

TS93

C. B. S. Traw and J. M. Smith. Hardware/software organization of a high-performance ATM host interface. *IEEE Journal on Selected Areas in Communications*, 11(2):240--253, February 1993.

UI81

USC-ISI. Transmission Control Protocol. Request for Comments 793, September 1981.

VL87

G. Varghese and T. Lauck. Hashed and hierarchical timing wheels: Data structures for the efficient implementation of a timer facility. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 25--38, November 1987.

Wat81

R. Watson. Identifiers (naming) in distributed systems. In B. Lampson, M. Paul, and H. Siebert, editors, *Distributed System -- Architecture and Implementation*, pages 191--210. Springer, New York, NY, 1981.

WC91

Zheng Wang and Jon Crowcroft. A new congestion control scheme: Slow start and search (Tri-S). *ACM Computer Communication Review*, 21(1):32--43, January 1991.

WC92

Zheng Wang and Jon Crowcroft. Eliminating periodic packet losses in 4.3-Tahoe BSD TCP congestion control algorithm. *ACM Computer Communication Review*, 22(2):9--16, April 1992.

Wel84

Terry Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8--19, June 1984.

WM87

Richard W. Watson and Sandy A. Mamrak. Gaining efficiency in transport services by

appropriate design and implementation choices. *ACM Transactions on Computer Systems*, 5 (2):97--120, May 1987.

WMC94

I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes*. Van Nostrand Reinhold, New York, NY, 1994.

X.292a

Open systems interconnection: Specification of abstract syntax notation one (asn.1). CCIT Recommendation X.208, 1992.

X.292b

Open systems interconnection: Specification of basic encoding rules for abstract syntax notation one (asn.1). CCIT Recommendation X.209, 1992.

YHA87

Y-S. Yeh, M. B. Hluchyj, and A.S. Acampora. The knockout switch: A simple, modular architecture for high-performance packet switching. *IEEE Journal of Selected Areas in Communication (JSAC)*, 5(8):1274--1283, October 1987.

ZL77

J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337--343, May 1977.

ZL78

J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530--536, September 1978.

About this document ...

Computer Networks: A Systems Approach

This document was generated using the [LaTeX2HTML](#) translator Version 95.1 (Fri Jan 20 1995)
Copyright © 1993, 1994, [Nikos Drakos](#), Computer Based Learning Unit, University of Leeds.

The command line arguments were:

```
latex2html -split 2 -address Copyright 1996, Morgan Kaufmann Publishers  
book.tex.
```

The translation was initiated by Larry Peterson on Wed Dec 20 09:59:41 MST 1995


Copyright 1996, Morgan Kaufmann Publishers

...Mosaic.


Those rare individuals who have not used a computer network will find some help getting started in that direction later in this Section.

...(ITU),
A subcommittee of the ITU on telecommunications (ITU-T) replaces an earlier subcommittee of the ITU, which was known by its French name Comité Consultatif International de Télégraphique et Téléphonique (CCITT).

...product.

Easy, that is, if we know the delay and the bandwidth. Sometimes we do not, and estimating them well is a challenge to protocol designers. More on this in Chapter .

...format.

We will study GIF, JPEG, and MPEG in Section , in the context of compression. In effect, however, these standards combine elements of presentation formatting and compression.

...way"

This is a definition of ``marshalling" taken from Webster's New Collegiate Dictionary.

...participants

We use the term *participant* for the parties involved in a secure communication since that is the term we have been using throughout the book to identify the two end-points of a channel. In the

security world, they are typically called *principals*.

Copyright 1996, Morgan Kaufmann Publishers