

Trend Micro, Inc.
EX1007-1

Sponsoring Editor Jennifer Mann
Production Manager Yonie Overton
Production Editor Julie Pabst
Editorial Assistant Jane Elliott
Cover and Text Design Ross Carron Design
Illustration ST Associates, Inc.
Copyeditor Jeff Van Bueren
Proofreader Ken DellaPenta
Composition Ed Sznyter, Babel Press
Indexer Steve Rath
Printer Courier Corporation

Morgan Kaufmann Publishers, Inc.
Editorial and Sales Office
340 Pine Street, Sixth Floor
San Francisco, CA 94104-3205 USA
Telephone 415/392-2665
Facsimile 415/982-2665
Internet mkp@mkp.com
Order toll free 800/745-7323

© 1996 by Morgan Kaufmann Publishers, Inc.
All rights reserved
Printed in the United States of America

00 99 98 97 96 5 4 3 2 1

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means—electronic, mechanical, photocopying, recording, or otherwise—without the prior written permission of the publisher.

Library of Congress Cataloging-in-Publication Data
Peterson, Larry L.
Computer networks : a systems approach / Larry L. Peterson and Bruce S. Davie.
p. cm.
Includes bibliographical references and index.
ISBN 1-55860-368-9
1. Computer networks. I. Davie, Bruce S. II. Title.
TK5105.5.P479 1996
004.6'5—dc20

96-1590

Trend Micro, Inc.
EX1007-2

PROBLEM
Building a Network

I must Create a System, or be enslav'd by another Man's; I will not Reason and Compare: my business is to Create.

—Blake

Suppose you want to build a computer network, one that has the potential to grow to global proportions and to 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 then to 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, the term *network* meant the set of serial lines used to attach dumb terminals to mainframe computers. To some, the term implies the voice telephone network. To 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, or 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, how do we proceed? Fortunately, we will not be building the first network. Others, most notably the community of researchers respon-

Trend Micro, Inc.
EX1007-3

sible 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 toward 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.

Foundation

Trend Micro, Inc.
EX1007-4

1.1 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.[1] 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 about the technology that underlies computer networks. This book explains computer networking technology from a systems perspective, which is to say it explains networking in a way that will teach you how networks are implemented—what sort of hardware is used, how it works, and how to implement 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.

1.1.1 Explosive Growth

While catchy phrases like "information superhighway" 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 1.1 plots 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 World 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 multiparty teleconferencing to take place over the Internet. Our goal in this book is to teach enough about the implementation of networks for you to 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 kind of information such as digital voice samples, digitized images, and so on, this potential was not particularly interesting 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

[1] Those rare individuals who have not used a computer network will find some help in getting started in that direction later in this section.

Trend Micro, Inc.
EX1007-5

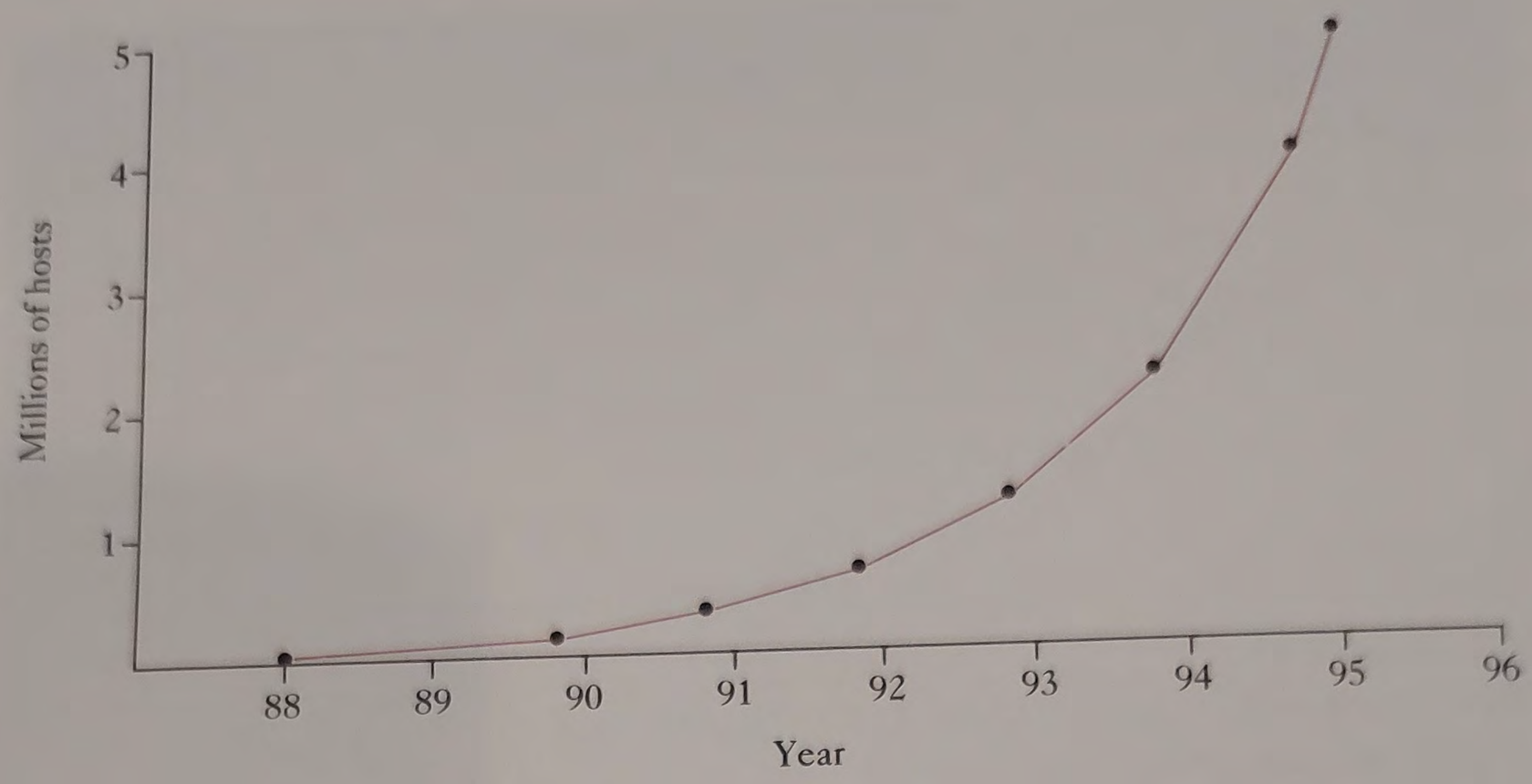


Figure 1.1 Growth in the number of computers connected to the Internet since 1988.

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.

1.1.2 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 subsections describe some of the most common applications that motivate our discussion of networking functionality.

FTP: File Transfer Protocol

The acronym FTP is a little confusing because it can interpreted to stand for either File Transfer *Program*, which is an application, or for 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:

Trend Micro, Inc.
EX1007-6

```
% 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. (All text typed by the user is shown in color in the example.) 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 could 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 retrieved 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; our computer connected to the FTP server on ds.internic.net. The client then issued a series of requests to the server, which responded. You 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

Trend Micro, Inc.
EX1007-7

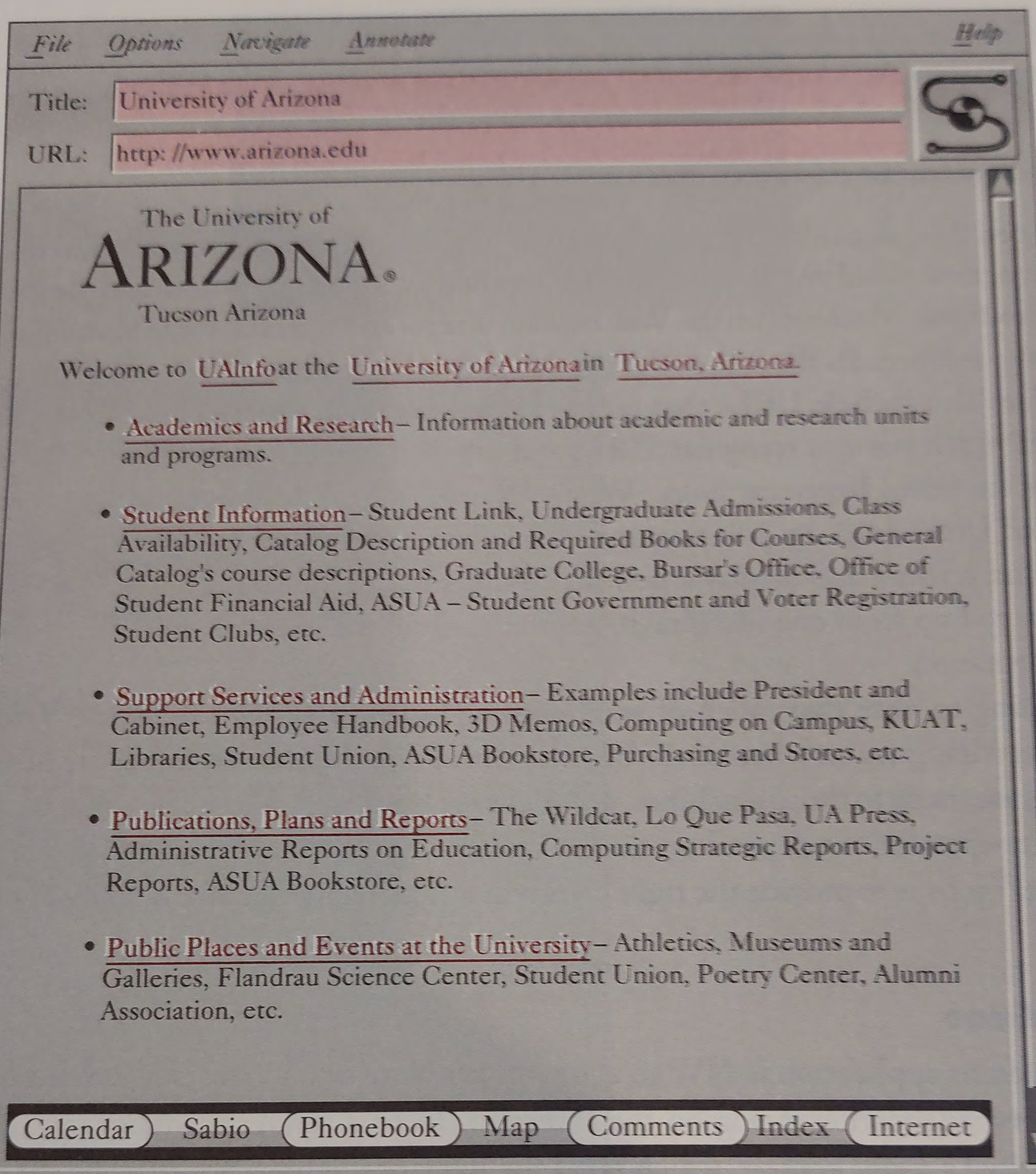


Figure 1.2 The Mosaic Web browser.

~~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 1.2 shows the Mosaic browser in use, displaying a page of information from the University of Arizona.

Trend Micro, Inc.
EX1007-8

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 would 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, an acronym for Network Video. NV is used primarily for multiparty videoconferencing on the Internet. To be a sender of video, you need 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 the size of each frame would be $(352 \times 240 \times 24)/8 = 247.5$ kilobytes (KB).

The rate at which images are taken is called the frame rate and needs to be about 25–30 frames per second to provide video quality that is as good as television. Frame rates of 15 frames per second are still tolerable, while update rates of greater than 30 frames per 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, 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

Trend Micro, Inc.
EX1007-9

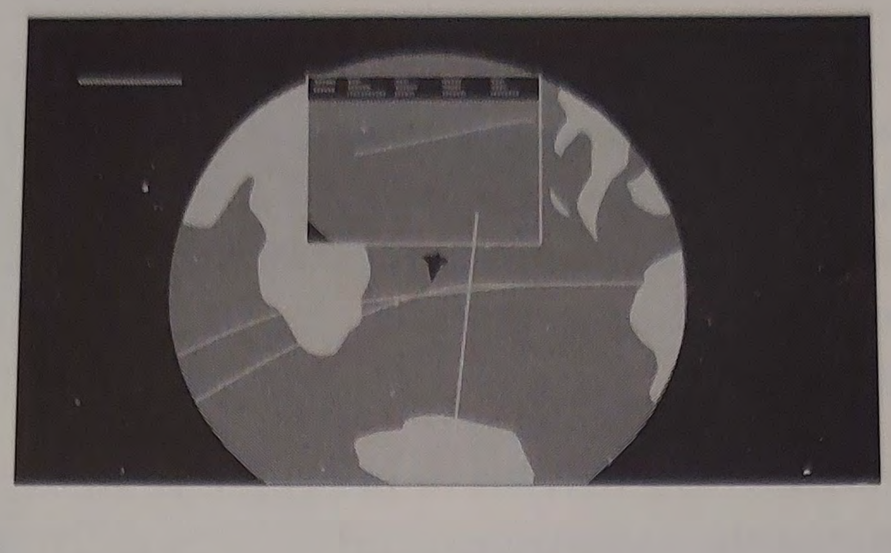


Figure 1.3 The NV video application.

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 1.3 shows the control panel for an NV session along with a frame of the video feed from NASA during a space shuttle mission.

Trend Micro, Inc.
EX1007-10

1.2 Requirements

Before trying to understand how a network that supports applications like FTP, WWW, and NV is designed and implemented, it is important 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 the application 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; and
- a *network provider* would list the characteristics of a system that is easy to administer and manage, for example, in which faults can be easily isolated and where 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.

1.2.1 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 (of which the Internet is the prime example), are designed to grow in a way that allows them the potential 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 1.4, 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

Trend Micro, Inc.
EX1007-11

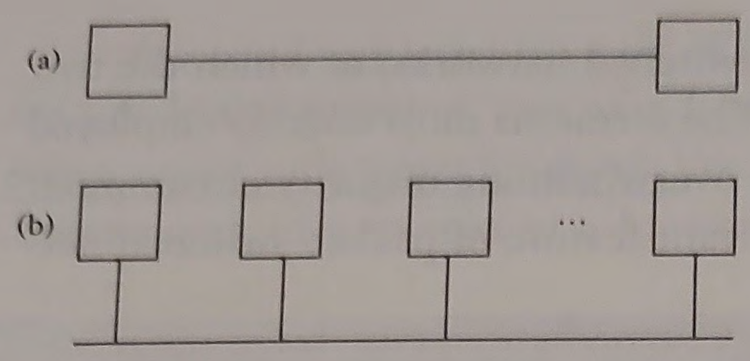


Figure 1.4 Direct links: (a) point-to-point; (b) multiple-access.

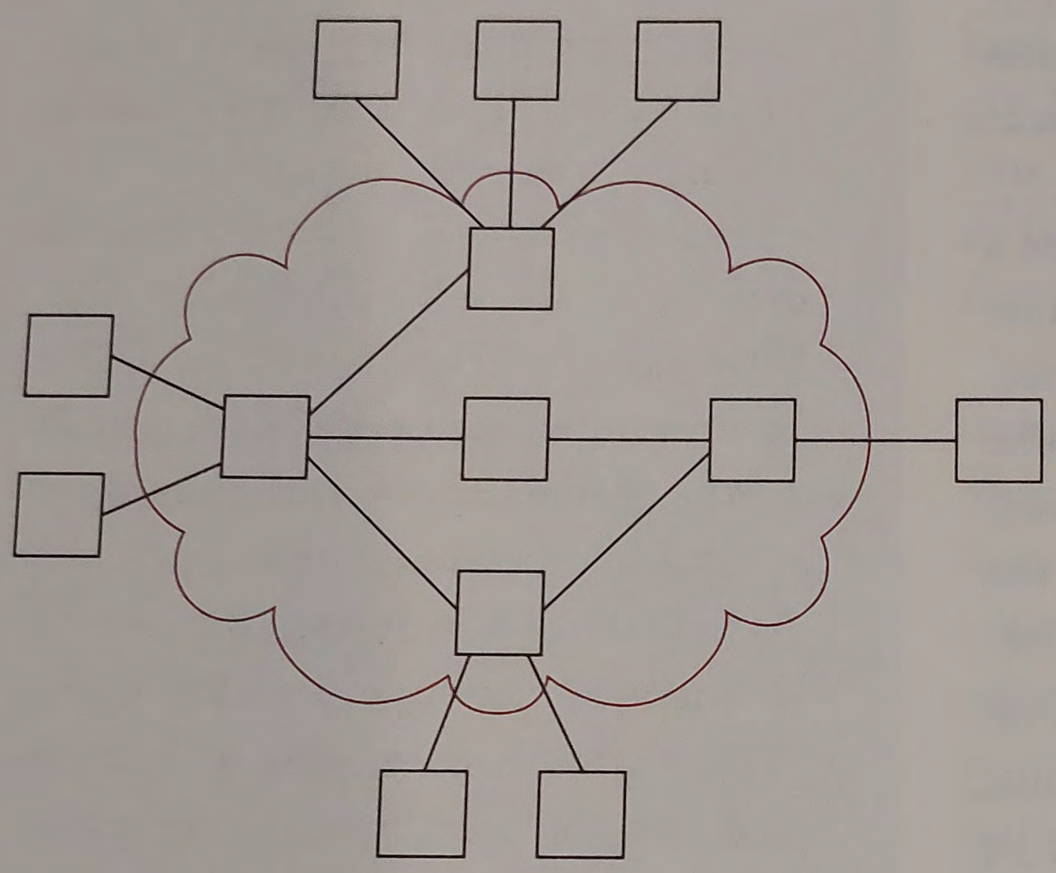


Figure 1.5 Switched network.

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.

If computer networks were limited to situations in which 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 could connect 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 1.5 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 organized in a systematic way, these forwarding nodes

Trend Micro, Inc.
EX1007-12

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 a 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 1.2.2.

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 1.5 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 1.5 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.

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 WANs (wide area networks)—the former typically extend less than 1 kilometer, while the latter can be worldwide. Other networks are classified as MANs (metropolitan area networks), 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 for 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; we discuss this issue more 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 in

Trend Micro, Inc.
EX1007-13

A second way in which a set of computers can be indirectly connected is shown in Figure 1.6. 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 lowercase i internet, and the currently operational TCP/IP Internet as the capital 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 that 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.

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; i.e., 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 use this address to decide how to forward the message toward the destination. The process of determining systematically how to forward messages toward 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 (*unicast*). 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. Or a source node might want to send a message to some subset of

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 is gaining attention today is the DAN (desk area network). 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, MANs, and WANs. Establishing this interconnection provides uniform access to all the resources that might be required by a network application.

Trend Micro, Inc.
EX1007-14

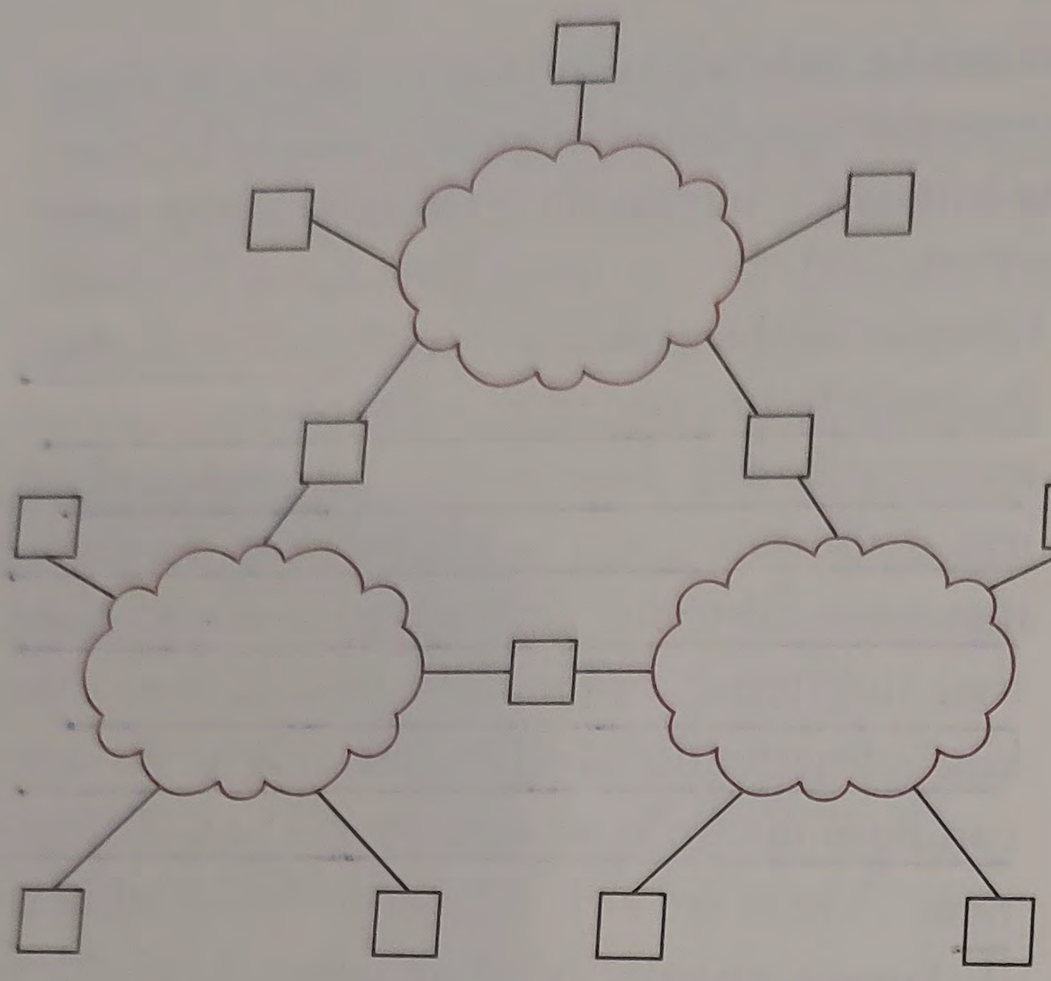


Figure 1.6 Interconnection of networks.

the other nodes, but not all of them, a situation called *multicast*. Thus, in addition to node-specific 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 as 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 level, the 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 connectivity), and to be able to use this address to route messages toward the appropriate destination node(s).

1.2.2 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? And, 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?

Trend Micro, Inc.
EX1007-15

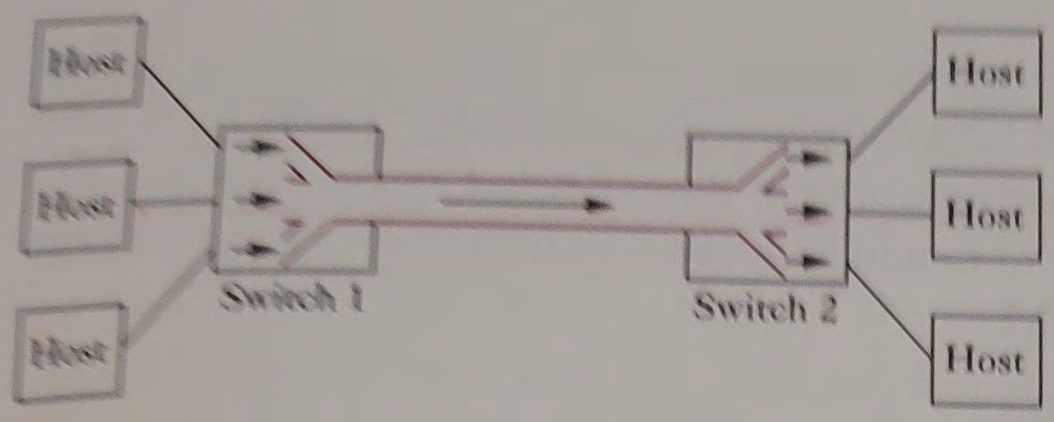


Figure 1.7 Multiplexing multiple logical flows over a single physical link.

To understand how hosts share a network, we need to introduce a fundamental concept, *multiplexing*, which means 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.

To see how this might work, consider the simple network illustrated in Figure 1.7, where the three hosts on the left side of the network are sending data to the three hosts on the right by sharing a switched network that contains only one physical link. (For simplicity, assume that 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 intentionally vague about exactly what a "flow of data" corresponds to. For the purposes of this discussion, assume that 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 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, 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

Trend Micro, Inc.
EX1007-16

large—for example, consider the amount of time you spend reading a Web page (leaving the link idle) compared to the time you spend fetching the page. Second, both STDM and FDM are limited to situations in which the maximum number of flows is fixed and known ahead of time. It is not practical to resize the quantum or to 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 for understanding the concept, statistical multiplexing is really quite simple; it involves two key ideas. First, it is like STDM in that the physical link is shared over 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 predetermined 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.

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, statistical multiplexing defines an upper bound on the size of the block of data that each flow is permitted to transmit at a given time. This limited-size block of data is typically referred to as a *packet*, 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 complete 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 1.8 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 such as the one in Figure 1.7, 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 may 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 be designed to service the different flows in a round-robin manner, just as in STDM. However, statistical multiplexing does not

Trend Micro, Inc.
EX1007-17

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 generally not introduced in a single physical link, but it can happen when packets experience different queuing delays in a multihop 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 that 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 33 ms. 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. The reason for this is that if the receiver knows the upper and lower bounds on the latency that a packet can experience, 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 9.

1.3 Network Architecture

In case you hadn't noticed, the previous section established a pretty substantial set of requirements for 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—usually 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.

1.3.1 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 can capture 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 that can be efficiently implemented in the underlying system. This is exactly what we were doing when

Trend Micro, Inc.
EX1007-18

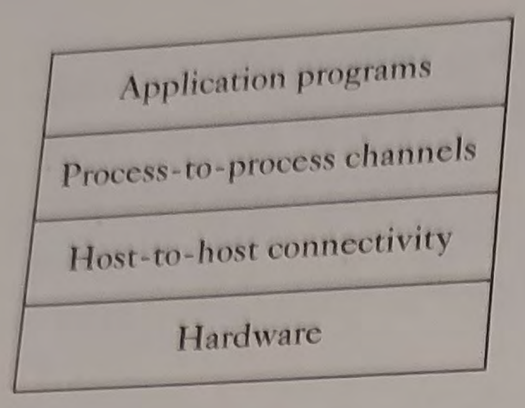


Figure 1.13 Example of a layered network system.

we introduced the idea of a channel in the previous section: we were providing an abstraction for applications that hides the complexity of the network from application writers.

Abstractions naturally lead to layering, especially in network systems. The general idea is that you start with the services offered by the underlying hardware, and then add 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 in 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 1.13. 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.

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, reusing the functions provided at all the other layers.

Thinking of a system as a linear sequence of layers is an oversimplification, 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 1.2.3: 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 multilevel networking system, as illustrated in Figure 1.14.

Using this discussion of layering as a foundation, we are now ready 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 pro-

Trend Micro, Inc.
EX1007-19

Application programs	
Request/reply channel	Message stream channel
Host-to-host connectivity	
Hardware	

Figure 1.14 Layered system with alternative abstractions available at a given layer.

Figure 1.15 Service and peer interfaces.

tocols) 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.

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, a 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 1.15.

Trend Micro, Inc.
EX1007-20

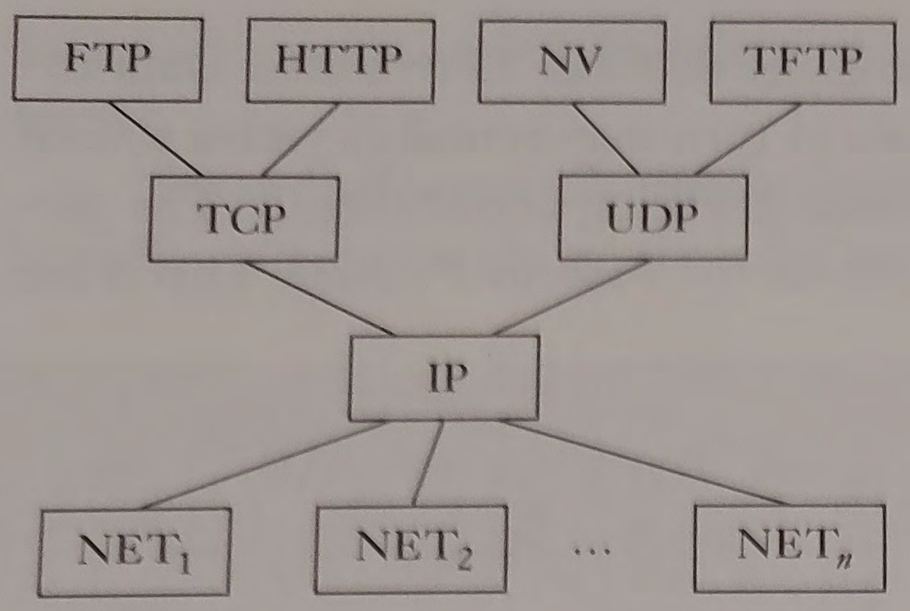


Figure 1.19 Internet protocol graph.

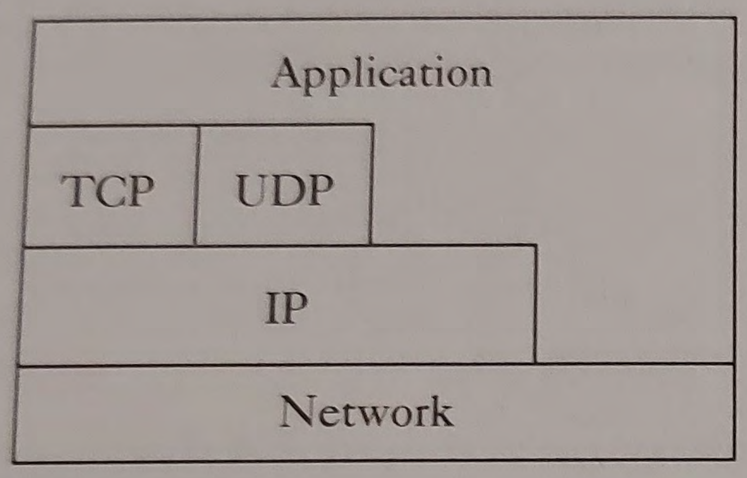


Figure 1.20 Alternative view of the Internet architecture.

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.

1.3.3 Internet Architecture

The Internet architecture, which is also sometimes called the TCP/IP architecture after its two main protocols, is depicted in Figure 1.19. An alternative representation is given in Figure 1.20. The Internet architecture evolved out of 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 the experience gained from building them was a major influence on the OSI reference model.

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 net-

Trend Micro, Inc.
EX1007-21

work device driver). For example, one might find Ethernet or Fiber Distributed Data Interface (FDDI) protocols at this layer. (These protocols in turn may actually involve several sublayers, 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 the language of the Internet, 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, such as FTP, TFTP (Trivial File Transport Protocol), Telnet (remote login), and SMTP (Simple Mail Transfer Protocol, or electronic mail), that enable the interoperation of popular applications. 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 (HyperText Transport Protocol). 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 1.20, the Internet architecture does not imply strict layering. The application is free to bypass the defined transport layers and to directly use IP or one of the underlying networks. In fact, programmers are free

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—on 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 for the success of one and the failure of the other 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

Trend Micro, Inc.
EX1007-22

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 in Figure 1.19, 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 is 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 with this requirement, the implementations of the protocols in the Internet architecture have evolved quite significantly over time as researchers and practitioners have gained experience using them.

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 to acquire, and the widespread 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)

1.4 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

Trend Micro, Inc.
EX1007-23

PROBLEM

Protocols Have to Be Implemented

There are three things to be looked to in a building: that it stand on the right spot; that it be securely founded; that it be successfully executed.

—Goethe

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 implementers can concentrate on the algorithms that make up network software. The good news is that because the *x*-kernel

Trend Micro, Inc.
EX1007-24

XID is a unique transaction id, much like CHAN's MID field. The reason that SunRPC cannot guarantee at-most-once semantics is that on the server side, SunRPC does not remember that it has already seen a particular XID once it has successfully completed the transaction. This is only a problem if the client retransmits a request message as a result of a timeout and that request message is in transit at exactly the same time as the reply to the original request is on its way from the server back to the client. When the retransmitted request arrives at the server, it looks like a new transaction, since the server thinks it has already completed the transaction with this XID. Clearly, if the reply arrives at the client before the timeout, then the request will not be retransmitted. Likewise, if the retransmitted request arrives at the server before the reply has been generated, then the server will recognize that transaction XID is already in progress, and it will discard the duplicate request message. So it is really quite unlikely that this erroneous behavior will occur. Note that the server's short-term memory about XIDs also means that it cannot protect itself against messages that have been delayed for a long time in the network. This has not been a serious problem with SunRPC, however, because it was originally designed for use on a LAN.

Returning to the SunRPC header format, the request message contains variable-length Credentials and Verifier fields, both of which are used by the client to authenticate itself to the server, that is, to give evidence that the client has the right to invoke the server. How a client authenticates itself to a server is a general issue that must be addressed by any protocol that wants to provide a reasonable level of security. This topic is discussed in more detail in the next chapter.

6.4 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.

Although each operating system is free to define its own protocol API (and most have), over time certain of these APIs have become widely supported, that is, they have been ported to operating systems other than their native system. The reason this happens is easy to understand: if a common API is supported on a wide range of systems, then it should be easy to port application programs written to this API to all of those systems. A word of warning is called for, however, since "easy" is a relative term. A network application that uses other OS facilities, such as processes, files, and memory objects, is still tightly bound to that OS.

What is not easy to understand is why a particular API becomes more popular than another. In the realm of networking, the most popular API is the *socket interface* originally developed in the Berkeley implementation of Unix. Because of its popularity, the socket in-

Trend Micro, Inc.
EX1007-25

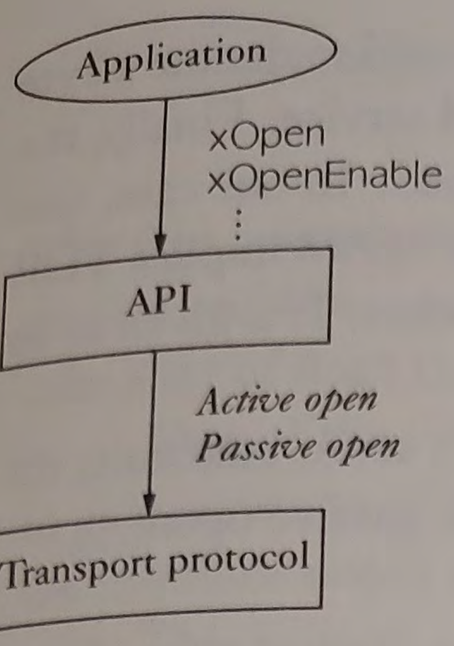


Figure 6.22 Relationship between the API and the protocol.

terface has been ported to nearly all versions of Unix, and a version is now even available on Microsoft Windows. This section gives a brief introduction to the socket interface.

Before describing the socket interface, it is important to keep two concerns separate in your mind. (Since we have not yet described the socket interface, we use the *x*-kernel interface to illustrate the point.) On the one hand, the protocol defines the *semantics* of the communication. For example, protocols define abstractions like "ports" and "connections," as well as abstract operations like "active open" and "passive open." The state-transition diagram for TCP given in Figure 6.6 identifies some of the abstract operations that TCP supports. On the other hand, the API defines the *syntax* for the communication. For example, it defines its own set of objects (sessions and participants) and it defines its own set of operations (xOpen and xOpenEnable). The implementation is then responsible for mapping the tangible set of operations and objects defined by the API onto the abstract set of operations and objects defined by the protocol, for example, sessions onto ports and xOpen onto "active open." This situation is illustrated in Figure 6.22.

We are now ready to describe the socket interface, which we cover in only enough detail to illustrate the relationship between the API and the transport protocols. In particular, we limit ourselves to showing how the socket interface is used in conjunction with TCP. The main abstraction of the socket interface, not surprisingly, is the *socket*. A good way to think of a socket is as the point where a local application process attaches to the network. The interface defines operations for creating a socket, attaching the socket to the network, sending/receiving messages through the socket, and closing the socket.

The first step is to create a socket, which is done with the following operation:

```
int socket(int domain, int type, int protocol)
```

The reason that this operation takes three arguments is that the socket interface was designed to be general enough to support any underlying protocol suite. Specifically, the domain argument specifies the protocol *family* that is going to be used. PF_INET is used to denote the Internet family; PF_UNIX is an alternative that denotes the Unix pipe facility. The type argu-

Trend Micro, Inc.
EX1007-26

ment indicates the semantics of the communication. SOCK_STREAM is used to denote a byte stream; SOCK_DGRAM is an alternative that denotes a message-oriented service. Finally, the protocol argument identifies the specific protocol that is going to be used. In our case, this argument is UNSPEC because the combination of PF_INET and SOCK_STREAM implies TCP. Finally, the return value from socket is a handle for the newly created socket. It is given as an argument to other socket operations.

The next step depends on whether you are a client or a server. On a server machine, the application process invokes the following three instructions to effect the passive open:

```
int bind(int socket, struct sockaddr *address, int addr_len)

int listen(int socket, int backlog)

int accept(int socket, struct sockaddr *address, int *addr_len)
```

The bind operation, as its name suggests, binds the newly created socket to the specified address. This is the address of the *local* participant, i.e., the server. Note that, when used with the Internet protocols, address is a data structure that includes both the IP address of the server and a UDP or TCP port number. The listen operation then defines how many connections can be pending on the specified socket. Finally, the accept operation carries out the passive open. It is a blocking operation that does not return until a remote participant has established a connection, and when it does complete, it returns a *new* socket that corresponds to this just-established connection, and the address argument contains the *remote* participant's address. It is possible to specify a "don't care" value, to enable any client to connect to this server.

Note that when accept returns, the original socket that was created and given as an argument to it still exists and still corresponds to the passive open; it is used in future invocations of accept. Also note that even though the server does not block until it invokes the accept operation, it is by invoking the listen operation that the application process causes TCP to make the transition to the LISTEN state.

On the client machine, the application process invokes a single operation to effect the active open:

```
int connect(int socket, struct sockaddr *address, int addr_len)
```

This operation does not return until the three-way handshake algorithm has successfully completed, meaning that the application can then begin sending data. In this case, address contains both the local and the remote participant's addresses. In practice, the client usually specifies only the remote participant's address and lets the system fill in the local information. Unlike a server that depends on a well-known port, a client typically does not care which port it uses for itself; the OS simply selects an unused one.

Once a connection is established, the application processes invoke the following two operations to send and receive data:

```
int write(int socket, char *message, int msg_len, int flags)
```

Trend Micro, Inc.
EX1007-27

```
int read(int socket, char *buffer, int buf_len, int flags)
```

The first operation sends the given message over the specified socket, while the second operation receives a message from the specified socket into the given buffer. Both operations take a set of flags that control certain details of the operation.

One of the interesting properties of an API is whether or not it allows the application to gain access to all the features of the underlying protocol. In the case of TCP, for example, the interface needs to give a program the ability to indicate that the data it is sending is urgent. This is done in the socket interface by setting the out-of-band flag (MSG_OOB) in the call to write. Similarly, on the receiving host, the interface needs to provide a mechanism by which TCP informs the application that urgent data has arrived. Delivering such protocol-to-application signals is typically the trickiest part of interface design. In the socket interface, for example, Unix delivers a SIGURG signal to the application process. In some cases, the API does not support a feature defined by the protocol. For example, the socket interface does not support a process's ability to send data while in the LISTEN state.

6.5 Performance

Recall that Chapter 1 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 that includes alternative transport protocols, we can discuss how to meaningfully measure its performance; we can hold a sort of protocol track meet. The importance of such measurements is that they represent the performance seen by application programs.

6.5.1 Experimental Method

We begin, as any report of experimental results should, by describing our experimental method. This includes the apparatus used in the experiments.

We ran our experiments on a pair of DEC 3000/600 workstations (with an Alpha 21064 processor running at 175 MHz) connected by an isolated 10-Mbps Ethernet. Each workstation was configured with a stand-alone version of the *x*-kernel, configured with the protocol graph illustrated in Figure 6.23. Each of the protocols at the top of the protocol graph (TSTTCP, TSTUDP, TSTRPC) ran a series of independent experiments (TST stands for "test"). Each experiment involved running five identical versions of the same test. Each test, in turn, involved sending a message of some specified size back and forth between the two machines 10,000 times. The system's clock was read at the beginning and end of each test, and the difference between these two times was divided by 10,000 to determine the time taken for each round trip. The average of these five times (the five runs of the test) are reported for each experiment below.

Trend Micro, Inc.
EX1007-28