# Chapter 13 Fundamentals of Networking and Network Protocols
# (Revision number 21)

Imagine that your computer has no connectivity to the Internet. Would we consider such a computer fully functional? Probably not. While we take the Internet and network connectivity for granted, it is a revelation to review how we got to this point in the first place. We will do such a historical review at the end of this chapter (see Section 13.18). First, we will understand the basic elements of networking that allow computers to talk to one another, be they in the same building or halfway across the globe from each other.

## 13.1 Preliminaries

As we mentioned in Chapter 10, peripheral devices interface with the rest of the computer system in one of two ways: *programmed I/O*; or *Direct Memory Access (DMA)*. The former is appropriate for slow speed devices and the latter is appropriate for high-speed devices. The network, a high-speed device, uses DMA for interfacing with the system.

Up until now, everything we have learned in the previous chapters regarding the computer is in our control, be it the hardware inside the box or the operating system abstractions for dealing with the hardware. While connecting our computer to the network requires a simple piece of DMA hardware, the implications of such a connection is profound. As opposed to a peripheral device such as a disk, which is still part of our "box", connecting our computer to the network opens our computer up to the whole world. On the one hand, connecting the computer to the network gives us the ability to browse the web, electronically chat with our friends anywhere in the world, and act as both a user and a contributor to information available via the network. On the other hand, since we cannot control everything that happens on the network, and it is unpredictable how our computer may be affected by such vagaries of the network. It is like a roller-coaster ride, you want the adrenaline rush but with an assurance of safety.

The number of topics worthy of discussion in the context of networks is plenty and includes network protocols, network security, network management, network services, and so on. Suffice it to say each of these topics deserves their own textbooks, and we list several good ones in the references.

The intent in this chapter is to stick to the theme of this textbook, namely, hardware and operating system issues as it pertains to your "box". With this goal in mind, this chapter takes you through a specific journey into computer networking focusing on the fundamentals of networking from both the hardware and operating systems points of view. The aim is not to delve into the depths but give you enough of an exposure to networking that it will perk your interest to pursue more advanced courses on this topic. We do not cover topics relating to network security or network management in this chapter.

We take a top-down approach to understanding the fascinating area of computer networking. There are three touch points for the operating system to facilitate the box to hook up to the network. Just as the *pthreads* library provides an API for developing multithreaded programs, the **`socket`** library provides an API for developing network applications. Defining and implementing this API is the first touch point for the operating system to facilitate network programming. Messages generated by an application program have to reach the destination. This involves a myriad of issues as we will see shortly. The abstraction that addresses all these issues is referred to as the **`protocol stack`** and is the second touch point for the operating system to facilitate network programming. The box itself connects to the network via a *Network Interface Card (NIC)*. The **`network device driver`** that interacts with the NIC is the third touch point for the operating system to facilitate network programming.

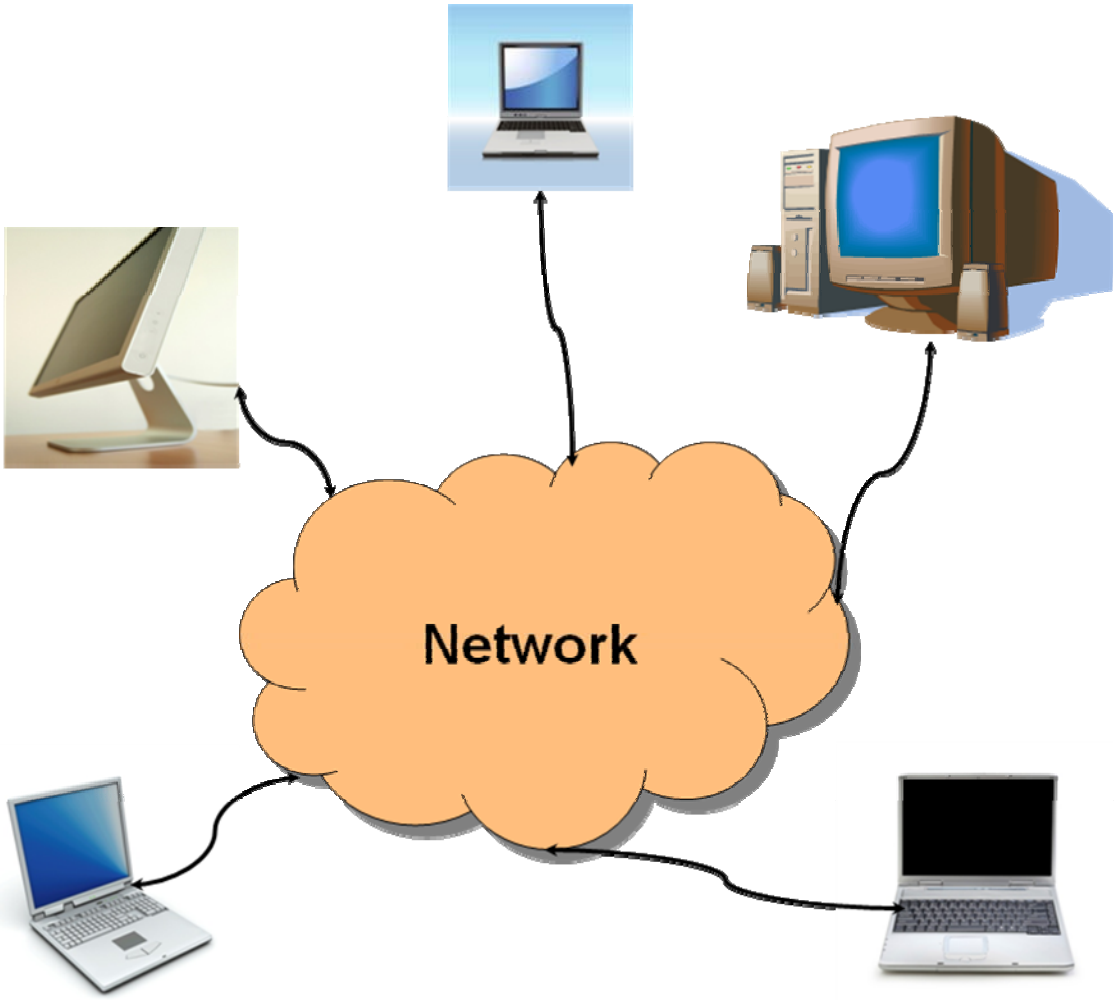



**Figure 13.1 Hosts connected to the network**

## 13.2 Basic Terminologies

We will start our journey with some basic terminologies. In network parlance, our computer that hooks on to the network is called a *host*. To hook up our computer to the

network we need a peripheral controller. This is called a *Network Interface Card* or *NIC* for short. Figure 13.1 shows a number of hosts connected to the network.

In Figure 13.1, we have shown the network as one big cloud. However, what is "*the network*"? It really is a composite of several networks, as we will see shortly. This collection of networks is given the name *Internet*.

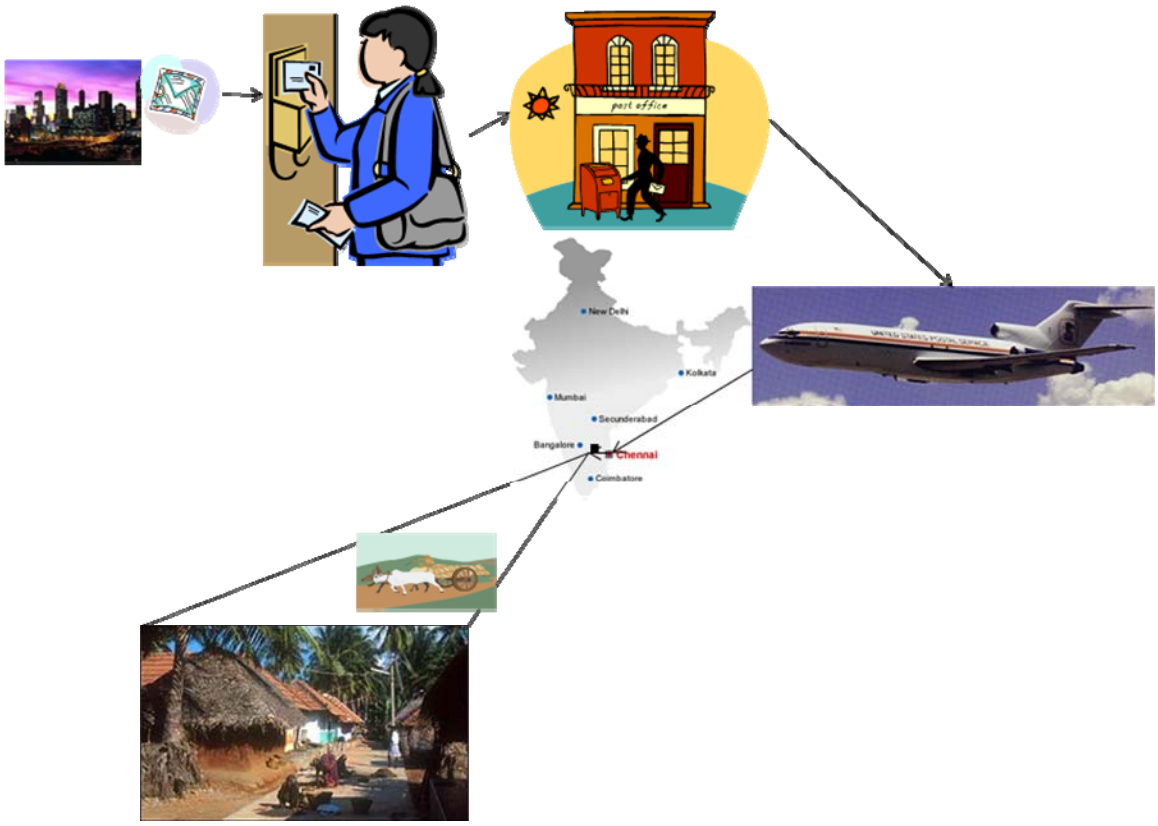



**Figure 13.2: Postal delivery from Atlanta, GA, USA to Mailpatti, Tamilnadu, India**

What is the Internet? You open any book on computer networks. Each will try to answer this question in a unique way. In this book, let us try to do this from *your* perspective. Consider the postal service as an analogy. Vasanthi wants to send a letter to her grandmother in Mailpatti, Tamilnadu, India from Atlanta, GA. She writes the address of her grandma on the envelope and places the letter in her mailbox (see Figure 13.2). The mailwoman picks up the letter. Of course, she does not know how to get the letter to Vasanthi's Grandma; all she knows is her route and the houses to which she has to deliver mail and houses from which she has to pick up mail to be sent out. But, she for sure knows that if the mail is handed to the regional post office they will take care of the rest of the journey for that piece of mail. Let us carry this story forward some more since it will serve as a good parallel to how the Internet works. In the post office, the letters get sorted and the letters meant for the state of Tamilnadu, India get placed in a bin.

Eventually, this bin gets a plane ride to Chennai, India. From the head post office in Chennai the letter gets routed to the post office in Mailpatti. Since there are no roads to reach grandma's house, the last mile is covered by a bullock cart from the regional post office in Mailpatti. Grandma is happy

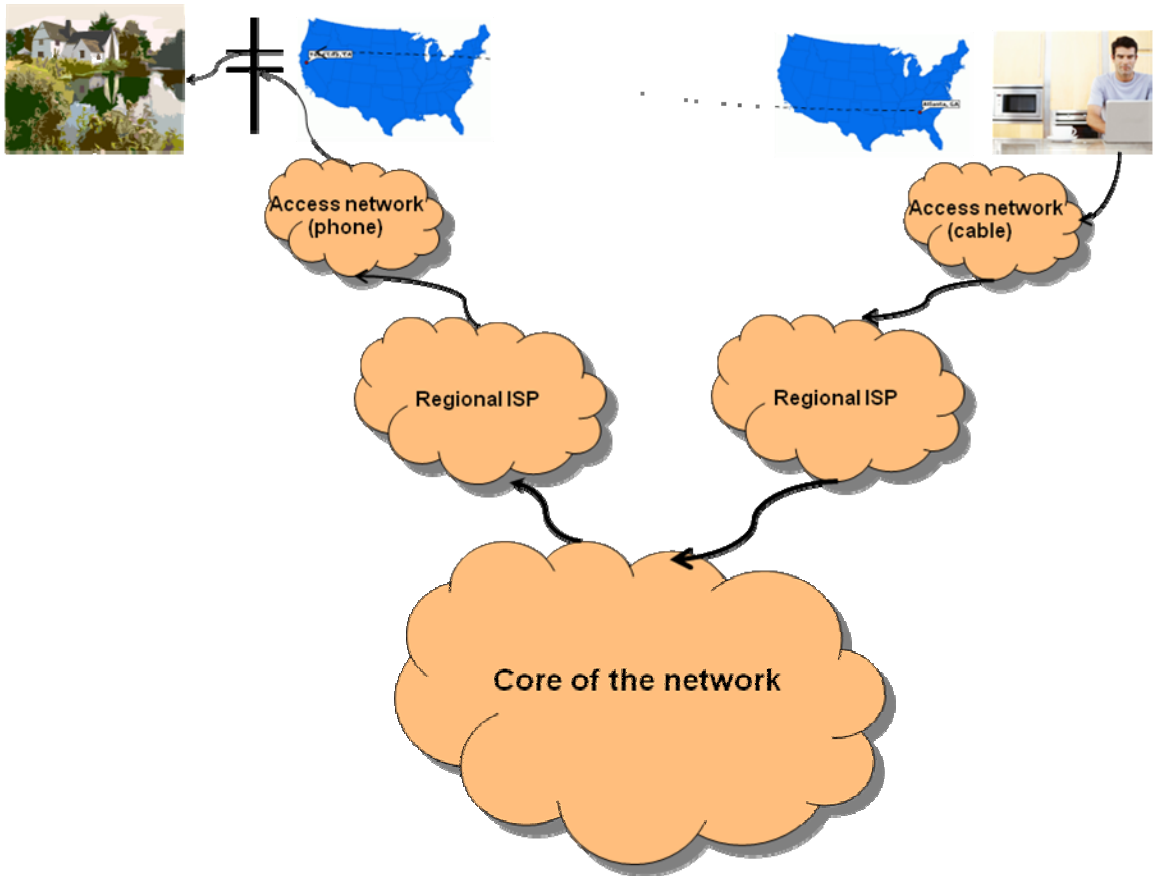



**Figure 13.3: Passage of Charlie's e-mail from Atlanta, GA to Yuba City, CA**

There is a striking resemblance between the journey undertaken by Vasanthi's letter and the passage of an e-mail from Charlie to his mom in Yuba City, California. Charlie lives in Atlanta and gets his Internet connectivity via cable. His computer plugs into a *cable modem*. The cable modem connects him to an *Internet Service Provider (ISP)*, which in this case is the local cable company. Since this ISP serves as Charlie's access point into the Internet, it represents an *access network*. One's access to the Internet could be by any means of course, cable, phone line, satellite, and so on. One would choose an appropriate ISP depending on one's preference. The access network is like the mailwoman. It knows how to get stuff to a given computer and stuff from a given computer but does not know how to get Charlie's e-mail all the way to his mom in California. However, the access network knows how to hand it over to a regional ISP, quite similar in functionality to the regional post office. The regional ISP knows how to talk to other regional ISPs across the country and across the globe. Charlie's e-mail from the Atlanta regional ISP is sent to the Bay area ISP in California. There are a lot more messages flying between Atlanta and the Bay area than just Charlie's e-mail. Therefore, the *core* of the network is capable

of handling much larger traffic, which is akin to sending all the US mail destined for Tamilnadu, India, in a U.S. Postal Service airplane. From the regional ISP In the Bay area, the message reaches the access network in Yuba City. Charlie's mom is not much of a computer user. She uses her phone line as a dial-up connection to connect to the Internet via a modem. The local phone company in Yuba City is the access network for Charlie's mom. The slow speed dial-up connection that delivers the e-mail to Charlie's mom's computer is akin to the bullock cart that finally delivers Vasanthi's letter to her grandma.

Each of the ISP cloud in Figure 13.3 represents computer systems of the respective ISPs. These ISP systems are not directly connected to one another. There are entities called *routers* that sit in the core of the network and know how to route messages between the various ISPs across the globe. The regional ISPs know one another and route messages destined for one another through the core of the network. This might seem like magic at first glance but it is not. How does the regional post office in Lilburn, GA know that Vasanthi's letter to her grandma has to be put on a plane to Chennai, India? The postal service has universally adopted *zipcode* as a way of uniquely identifying any particular postal region in the entire world. In the same manner, Internet has adopted a universal addressing system for every device that may be connected to the Internet. This is the familiar *Internet Protocol Address* or *IP address* for short. Charlie's computer has a globally unique IP address, and so does his mom's computer. The collection of access networks, ISPs, and the core of the network together form what we commonly refer to as the Internet. In short, Internet is a network of networks.

The key point to understand about the packet flow on the Internet is that a packet traverses through a series of *queues* that reside at the input of the routers *en route* from source to destination. In the absence of any contention, the queues are empty and Charlie's e-mail will sail through these routers to his mom. However, in the presence of other traffic (i.e., *network congestion* – see Section 13.6.3.1), a packet may experience *queuing delays* as it makes its way through the routers towards the destination. Due to its finite-size, if a queue associated with a router is full then packets may be dropped leading to *packet loss*. Thus, queuing delays and packet loss are inherent to the nature of packet traffic on the Internet. We will discuss these aspects in more detail in later sections (see Sections 13.3 and 13.11.1).

At an abstract level, the postal service delivers letters from person A to person B. Let us look at the supporting infrastructure for the global postal service. It includes human mail carriers on foot, bikes, boats, and bullock carts. It further includes postal trucks, freight carriers, and even airplanes. Similarly, a whole host of gadgetry supports the Internet. Let us take a bird's eye view of this networking landscape before delving into the details in the rest of the chapter.

We will expand the earlier definition of a *host* to include computers that we come into contact directly in everyday life (our laptops, PDAs, servers such as web server and mail server). These computers represent the *edge* of the network. Quite often, we hear the terms *client* and *server* machines. Basically, these terminologies refer to the role played

by a given host. For example, when we do a Google search, our machine is a client; the machine that takes on the role of the search engine at the other end is a server. The edge of the network is distinguished from the *core* of the network. The machines that sit in the bowels of the network routing packets from source to destination constitute the core of the network.

Today, you connect to your computer to a *local area network (LAN)* if you are in school or office or in your dorm room. Even at home, especially in the western world, many may have a LAN connecting all the machines in the home. As we mentioned already, you connect to an *Internet Service Provider (ISP)* if you are at home, which may be a cable company or a phone company depending on your preference. The ISPs have a way of communicating with one another so that independent of the ISP used by the edges of the network, Charlie from Atlanta, Georgia is always able to send an e-mail to his mom in Yuba City, California. There are a number of other gadgetry that complete the nuts and bolts of the Internet infrastructure. These include the physical medium itself for ferrying the bits, as well as the electronic circuitry for connecting the hosts to the physical medium. Such circuitry include *hubs*/*repeaters, bridges, switches, and routers*. We will discuss these hardware elements and their functionalities in a later section (see Section 13.9) after we discuss the networking software.

**13.3 Networking Software**

The networking software is a crucial part of any modern operating system. We usually refer to this part of the operating system as the *protocol stack*. Let us understand what is meant by network protocol. It is a *language* that defines the *syntax* and *semantics* of messages for computers to talk to one another. As the reader may have already guessed from the use of the term *protocol* in this chapter, the term refers to any agreed upon convention for two entities to interact with each other. For example, even inside the "box", the processor and the memory follow a protocol for interaction on the memory bus. The software convention for register save/restore that we discussed in Chapter 2 (Section 2.14) is also a protocol between the caller and the callee.

However, the protocol for computers to communicate with one another gets complicated due to a variety of reasons. For example, we will shortly review Ethernet, a protocol used on LANs (Section 13.8.1). Ethernet uses a maximum packet size of 1518 bytes, referred to as a *frame* that contains data, the destination address, and other necessary information to assure the integrity of data transmission. The maximum packet size limit for a given network technology arises primarily from the details of the protocol design constraints. An orthogonal issue is the maximum transmission rate on the medium for a given network technology. Just as the processor clock speed is a function of the delay properties of the logic used in the implementation, the maximum transmission rate limit for any network technology also arises due to the signaling properties of the physical layer and the delay properties of the logic used to drive the medium.

Consider usage of the network. You may be sending an image you took in class over the network from your dorm room to your family. The image may be several megabytes in size. You can immediately see the problem. Your image is not going to fit in one

network packet. Thus, your notion of a message (arbitrary in size) has to be *broken into several smaller sized packets* commensurate with the physical limitations of the network medium. Breaking a message into packets immediately introduces another problem. Referring to Figure 13.3, the packets originating in Charlie's computer may have to go over several different networks before it gets to his mom's home computer. There is other traffic flows on the network as well in addition to Charlie's e-mail. Consequently, there is no predicting how much *queuing delays* his e-mail will experience along the way both in the presence of and in the absence of other competing network flows. There is no way to guarantee that the packets of the message will arrive *in order* at the destination. That is, consider that a message consists of three packets: 0, 1, and 2. They are sent in that order at the sender. However, since there may be multiple paths between the sender and the receiver, the network is free to route the packets along different paths. Thus, the receiver may get the packet in the order 0, 2, and 1. Still the receiver has to be able to assemble these packets correctly to form the original message. Thus, *out of order delivery* of packets is a second issue in data transport. A third problem arises due to the very nature of the network architecture, namely, a fully decentralized network with local autonomy over how packets are buffered, forwarded, or ignored. Consequently, a packet may get *lost* on the network. Such packet loss happens due to insufficient resources (for example, buffer capacity in an intermediate router) as a packet moves through the network. A fourth problem involves transient failures along the way that may *mangle* the contents of a packet. That is, there can be *bit errors* in the transmission of the packet through the network.

To summarize, the **set of problems** introduced because a message has to traverse through a network from source to destination:

1. **Arbitrary message size** and physical limitations of network packets
2. **Out of order delivery** of packets
3. **Packet loss** in the network
4. **Bit errors** in transmission
5. **Queuing delays** en route to the destination

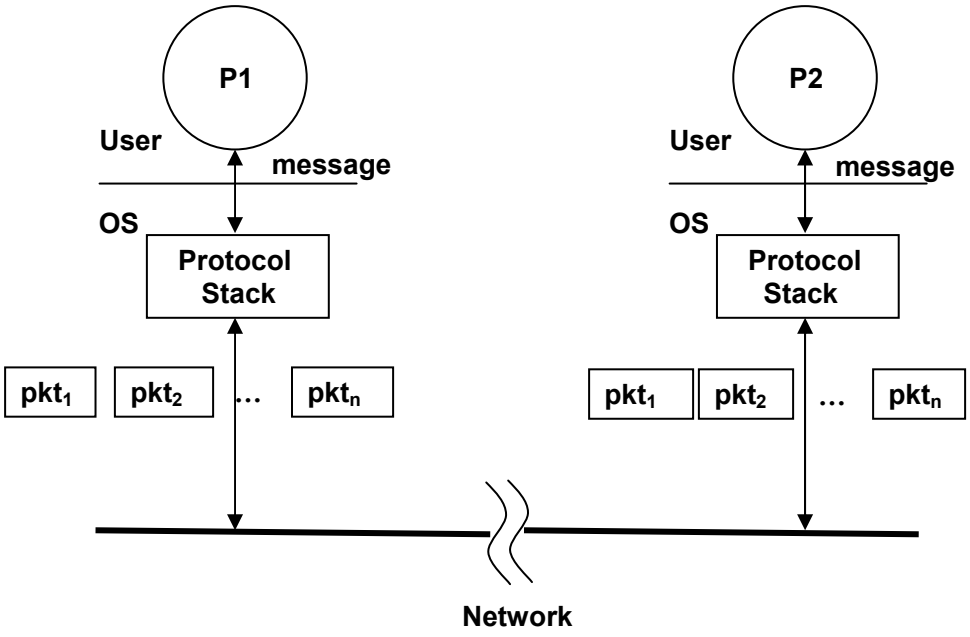



**Figure 13.4: Message Exchange Between Two Network Connected Nodes**

Of course, we can let the application program take care of all the above problems. However, since any network application will need to worry about these problems, it makes sense to take care of these problems as part of the operating system. The portion of the operating system that addresses these problems is the *protocol stack*. In a bit it will become clear why this piece of software is called a protocol *stack*.

Just as the system software (compiler and operating system) takes an arbitrary application data structure and houses it into rigidly structured physical memory, the protocol stack takes an arbitrary sized application message, transports it across the network, and reconstructs the message in its entirety at the destination. Figure 13.4 shows this message exchange between two processes P1 and P2 on two different hosts.

### 13.4 Protocol Stack

Let us investigate how we should structure the protocol stack. One possibility is to implement the protocol stack as one big monolith subsuming all the functions that we discussed so far to ensure the reliable transport of the message from source to destination. The downside to that approach is that it unnecessarily bundles in the details of the physical network into this stack. For example, if the physical network changes then it affects this entire stack.

Protocol layering addresses this problem precisely. Rather than bundle in all the functionalities into one stack, protocol layering uses the power of abstraction to separate the concerns. It is for this reason that we call this piece of the operating system a protocol *stack*. Outgoing messages are *pushed* down the layers of the stack from the application down to the physical medium, and similarly incoming messages are *popped* up the layers of the stack from the wire to the application.

### 13.4.1 Internet Protocol Stack

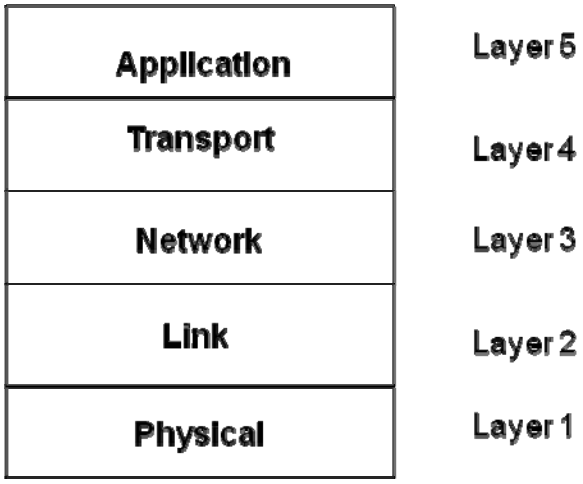



**Figure 13.5: Internet Protocol Stack**

Next, we need to understand the concerns that the protocol stack has to address. This has been the focus of networking research that had its origins in the late 60's. Networking luminaries such as Vinton Cerf and Robert Kahn envisioned that such network of computers might have to be networked together and coined the term "Internetting." Of course, today Internet is a household term. By the end of the 70's, much of the protocols that are considered ubiquitous today such as TCP, UDP, and IP were conceptually in place in the Internet protocol architecture.

Figure 13.5 shows the 5-layer Internet protocol stack. We will quickly summarize the role performed by each layer of the Internet protocol stack. In later sections of this chapter, we will go into much more depth on each of the transport, network, and link layers.

**Application Layer**

As the name suggests, this layer is responsible for supporting network-based applications such as an instant messenger (IM), multi-player video games, P2P music/video sharing, web browser, electronic mail, and file transfer. A variety of protocols may be employed in this layer including HTTP for web applications, SMTP for electronic mail, and FTP for file transfer. Essentially, the role of such protocols is to provide a common language for application level entities (clients, servers, and peers) to use for communicating with each other.

**Transport Layer**

This layer is responsible for taking an application layer message and ferrying it between the ends of communication. Naturally, this layer has to worry about all the vagaries of the network we discussed earlier (such as breaking a message into packets and out of order delivery of packets). TCP and UDP are the two dominant transport protocols in use on the Internet today. TCP (which stands for *Transmission Control Protocol*)

provides a *reliable* and *in-order delivery* of application data in the form of a *byte-stream* between two endpoints. It is a connection oriented protocol, i.e. , there is a connection established[1] between the two endpoints much like a telephone call before the actual data transmission takes place. Once the conversation is over, the connection is closed, usually referred to as connection *teardown* in network parlance. On the other hand, UDP (which stands for *User Datagram Protocol*) is analogous to sending a postcard through U.S. mail. It deals with *messages* that have *strict boundaries*, i.e., there is no relationship at the protocol level between successive messages sent using UDP. Succinctly put, TCP gives stream semantics for data transport while UDP gives datagram semantics. UDP does not involve any connection establishment before sending the message, or any teardown after sending the message. With UDP, the messages may arrive out of order since the protocol makes no guarantees regarding ordered delivery. In short, the salient difference between these two dominant transport protocols on the Internet is that TCP gives end-to-end reliability for in-order data delivery, while UDP does not.

**Network Layer**

The transport layer has no knowledge as to how to *route* a packet from source to destination. Routing is the responsibility of the network layer of the protocol stack. The role of the network layer is simple: on the sending side, given a packet from the transport layer, the network layer finds a way to get the packet to the intended destination address. At the receiving end, the network layer passes the packet up to the transport layer, which is responsible for collating this packet into the message to be delivered to the application layer. In this sense, the role of the network layer is much like the postal service. Drop your letter in the postbox and hope that it gets to the destination. Of course, in the case of the postal service, a human is reading the address you have scribbled on the envelope and determining how best to route the letter. In the case of the network packet, which is processed by the network layer, we need a precise format of the information in the packet (address, data, etc.). In Internet parlance, the protocol for this layer has the generic name *Internet Protocol* (IP for short), and subsumes both the formatting of the packet as well as determining the route that a packet has to take towards the destination.

**Link Layer**

Consider the postal service analogy. Vasanthi's letter went by airplane from Atlanta, GA to Chennai, India. However, to cover the last mile, a bullock cart carried the letter from the post office in Mailpatti to grandma's house. The airplane and bullock cart serve as different conduits for the postal service for transporting Vasanthi's letter between different hops of the postal system. The link layer performs a similar role for ferrying the IP packets between nodes on the Internet through which a packet has to be routed from source to destination. Ethernet, Token Ring, and IEEE 802.11 are all examples of link layer protocols. The network layer hands the IP packet to the appropriate link layer depending on the next hop to be taken by the packet (if necessary breaking up the IP packet into *fragments* at the network layer commensurate with the characteristics of the

[1] Telephone infrastructure establishes a real connection called circuit switching between the two endpoints by pre-allocating physical resources. On the other hand, TCP is "connection-oriented" since there is no real connection between the two endpoints in terms of physical resources.

link layer). The link layer delivers the fragments to the next hop, where they are passed up to the network layer. This process is repeated until the packet (possibly fragmented) reaches the destination. As should be evident, a given IP packet may be handled by diverse link layer protocols in its journey from source to destination. At the destination, the network layer *reassembles* the fragments to reconstruct the original IP packet.

**Physical Layer**

This layer is responsible for physically (electrically, optically, etc.) moving the *bits* of the packet from one node to the next. In this sense, this layer is intimately tied to the link layer. A given link layer protocol may use multiple physical media to ferry the bits. And for each such physical medium there may be a distinct physical layer protocol. For example, Ethernet may use a different physical layer protocol for a twisted pair of copper wires, a coaxial cable, an optical fiber, etc.

Figure 13.6 shows the passage of a message from source to destination through the layers of the protocol stack and through multiple network hops.

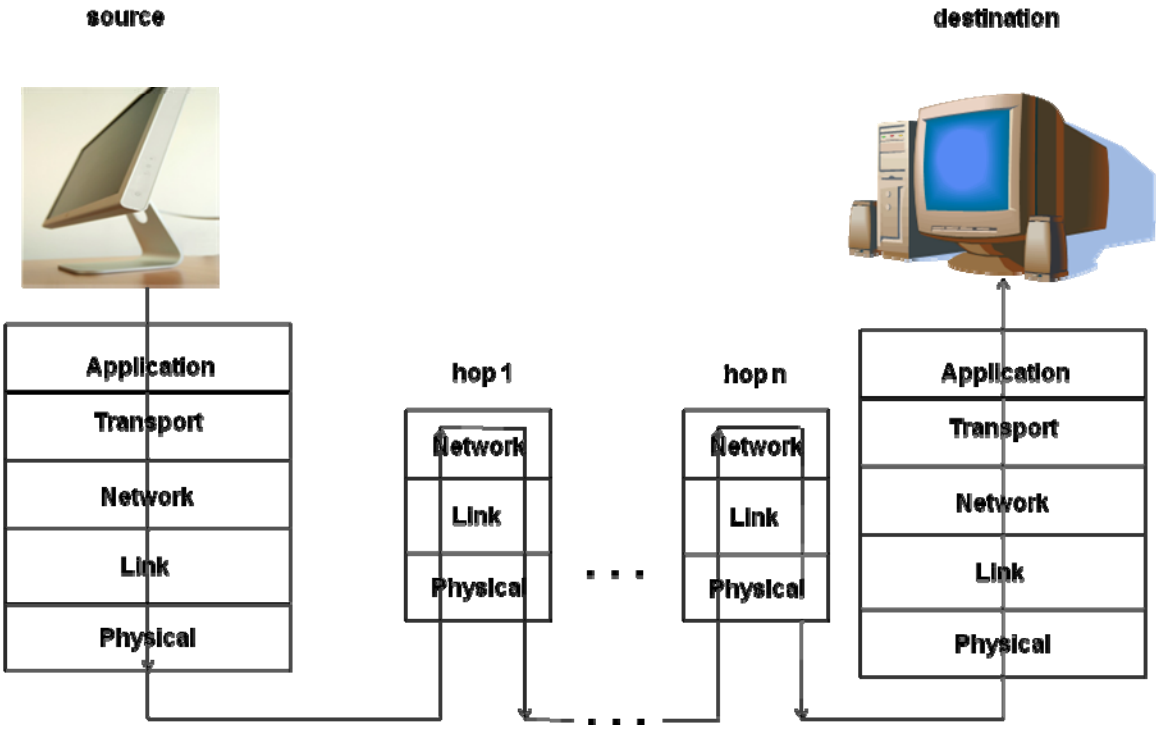



**Figure 13.6: Passage of a packet through the network**

There are well-defined interfaces between any two layers of the protocol stack. Layering provides the modularity for each layer to make its own implementation decisions independent of the other layers.

Layering is a structuring tool for combating the complexity of the protocol stack. It allows partitioning the total responsibility for message transmission and reception among

various layers. The modularity allows integration of a new module at a particular layer with minimal changes to the other layers. For example, appearance of a new physical layer in the stack affects the link layer, and in principle should not affect the network and transport layers. At first glance, it would appear that a potential downside to layering might be a performance penalty, as the message has to traverse several layers. However, judicious definition of interfaces between the layers would avoid such inefficiencies.

There is no hard and fast rule as to the number of layers that a protocol stack should have. Really, it depends on the functionality provided by the protocol stack.

### 13.4.2 OSI Model

The International Standards Organization (ISO) came up with a 7-layer model of the protocol stack, the *Open Systems Interconnection (OSI)* suite. The OSI 7-layer model is an abstract reference model for describing the functionalities that are part of the protocol stack and how to apportion them among the various layers. Figure 13.7 shows this reference model.

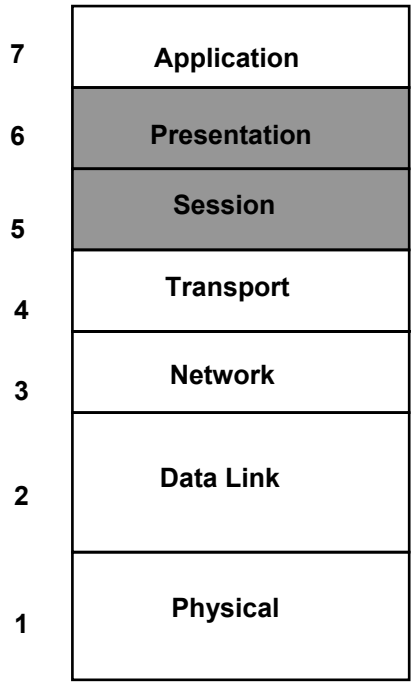



**Figure 13.7: OSI reference model**

Comparing Figures 13.5 and 13.7, we can see that the latter has two more layers than the former: *presentation, and session.* The *session* layer, as the name suggests, is the manager of a *specific* communication session between two end points. For example, imagine you are having several simultaneous *instant messaging (IM)* sessions with friends across the network. The session layer maintains process level information specific to each such pairs of communication sessions. The transport layer worries about getting the message reliably across *any* pair of end points. The session layer abstracts out the details of the transport from the application, and presents a higher-level interface to

the application (for example the Unix *socket* abstraction). The *presentation* layer subsumes some functionality that may be common across several applications. For example, formatting the textual output on your display window is independent of the specific client program (AOL, MSN, etc.) you may be using for IM. Thus, presentation functions (such as locally echoing text characters, formatting, and character conversion) that are not dependent on the internal details of the application itself belong in the presentation layer.

### 13.4.3 Practical issues with layering

The OSI model serves as a useful reference checklist to ensure that an implementation covers all the necessary functionalities of the protocol stack. However, real implementations of the stack seldom stick to the strict layering discipline prescribed by the model. As a practical matter, the Internet evolved simultaneous with the definition of the OSI model in the 80's. As we have mentioned already, TCP/IP protocol is the *de facto* standard for the transport and network layers for communication on the Internet. The main observation to take away is that with the evolution of the Internet, standard protocols such as TCP and IP have led to the collapsing of the layers embodied in the OSI model. For example, protocols such as HTTP, FTP, and SMTP subsume layers 7-5 of the OSI model. TCP and UDP are at the level of layer 4 of the OSI model. IP subsumes layer 3 of the OSI model functionality. Network interfaces (such as an Ethernet card on your computer) assume the role of layer 2 of the OSI model.

We have given a high-level description of the five-layer Internet protocol stack. There are several excellent textbooks devoted to covering the details of each of these layers. As we have said at the outset, the intent in this chapter is to expose the reader to network as an important I/O device, both from the point of view of the system architecture and the operating system.

We will take a top-down approach in exploring the layers of the protocol stack and the design considerations therein. We start with a discussion of the application layer to set the context for the rest of the chapter. We place particular emphasis on the transport layer since that is the strongest touch point to the operating system. We then work our way down the stack to cover the networking layer, which is also part of the operating system. Finally, we will review the link layer, which represents the strongest touch point to the system architecture.

A note on the organization of the rest of the chapter. We explore the transport, the network, and the link layers in some depth in the next three subsections. This level of detail may not be necessary depending on the reader's perspective. Since we have already given a high-level overview of the functionalities of these layers, it is perfectly fine for the reader to skip ahead to Section 13.10 where we discuss the relationship between these layers and then go on to exploring operating system issues in implementing the protocol stack.

## 13.5 Application Layer

As we know, Internet applications abound in number with the ubiquity of the network-connected gadgets ranging from iPhones to high-performance computing clusters. In this context, it is important to differentiate between *applications* and *application-layer protocols*.

Generally speaking any network application has two parts:

- **Client**: This is the part that sits on end devices such as handhelds, cellphones, laptops, and desktops.
- **Server**: This is the part that provides the expected functionality of some network service (e.g., a search engine).

Examples of network applications include the World Wide Web (WWW), electronic mail, and network file systems. For instance, the client side of the WWW is a web browser such as FireFox and Internet Explorer. The server side of WWW is referred to as a Web Server that includes generic ones such as Apache, and portals for a host of specialized services such as Google and Yahoo!. As another instance, the client side of electronic mail are programs such Microsoft Outlook and Unix Pine. The server side is a mail server such as Microsoft Exchange.

A network application is much more than the application-layer protocol. For example, the web browser has to maintain a history of URLs accessed, a cache of web pages downloaded, etc. These details make a particular application different from another. On the other hand, since the message exchanges between the client and the server of a network application are well defined, they are embodied in application layer protocols.

Application-layer protocols may be tailored for different classes of network applications. For example, web applications use **HTTP** (which stands for Hyper-Text Transfer Protocol) for specifying the interactions between the web clients and the servers. Similarly, electronic mail uses **SMTP** (which stands for Simple Mail Transfer Protocol) for specifying the interactions between the mail clients and the servers.

Many network applications such WWW and e-mail often transcend hardware architecture of the box and the operating systems. This is the reason why you can read your e-mail or access a popular website such as **CNN** or **BBC** from your cellphone and/or public terminals at airports and Internet Cafés. Thus, application-layer protocols such as HTTP and SMTP are *standards* independent of the *platform* (defined as the combination of the hardware architecture and the operating system) that hosts either the client or the server for such applications.

On the other hand, operating systems provide their own unique network services. For example, you may be accessing a file from a departmental Unix file system. Alternatively, you may be printing from a Unix terminal to a network printer. These are also examples of client-server applications that enhance the functionality of an operating system. To enable the development of such applications, operating systems provide network communication libraries. These libraries provide APIs for client/server

interaction across the network much like how pthreads provide APIs for the interactions of threads within an address space. Such a communication library represents an application layer protocol as well. For example, Unix operating system provides the **socket** library as the API for building network applications. Other popular operating systems such as Microsoft Vista and Apple MacOS also provide a similar API for enabling the development of network applications on their respective platforms. Similar to implementing a thread library (see Chapter 12), there are specific operating system issues in implementing the socket library API. Such details are outside the scope of this textbook. The interested reader is referred to other textbooks that deal with these issues[2,3]. We return to discussing the basics of network programming using socket API later in this textbook (see Section 13.15).

### 13.6 Transport Layer

Let us assume that the transport layer provides a set of calls, *Application Program Interface (API)*, so that the application layer can send and receive data on the network.

- **send (destination-address, data)**
- **receive (source-address, data)**

Let us enumerate the expected functionality of the transport layer of the protocol stack:

1. Support arbitrary data size at the application level
2. Support in-order delivery of data
3. Shield the application from loss of data
4. Shield the application from bit errors in transmission.

The transport layer may view the data from the application layer either as a *byte stream* or as a *message*. Correspondingly, the transport may be either *stream* or *connection-oriented* (e.g., *TCP – Transmission Control Protocol*) in which case the application data is considered a continuous stream of bytes. The transport layer chunks the data into pre-defined units called *segments* and sends these segments to its peer at the destination. Alternatively, the transport layer may be *message* or *datagram oriented* (e.g., *UDP – User Datagram Protocol*) in which case the application data is treated similar to a postcard sent through the postal mail system. To keep the discussion simple, we will simply refer to the unit of transfer at the transport level as a *message*.

The transport layer at the source has to break up the data into *packets* commensurate with the hardware limitations of the network. Correspondingly, the peer transport layer at the destination has to assemble the packets into the original message for delivery to the recipient of the message. We refer to this combined functionality as *scatter/gather*[4]. Recall that the packets may arrive out of order at the destination. Therefore, the transport layer at the source gives a unique *sequence number* for each packet of a message. The sequence number helps in the reconstruction of the original message at the receiving end

[2] TCP/IP Illustrated, Volume 2: The Implementation (Addison-Wesley Professional Computing Series) by Gary R. Wright, W. Richard Stevens

[3] The Design and Implementation of the FreeBSD Operating System, by Marshall Kirk McKusick, George V. Neville-Neil

[4] Referred to as *segmenting* the stream of data into packets in TCP parlance.

despite the order of arrival of the individual packets. Thus, attaching a unique sequence number to each packet takes care of the first two problems with network communication, namely, arbitrary message size and out of order delivery.

The source needs confirmation that the destination did receive the packets. Recall that there could be packet loss and packet corruption *en route*. In either case, the upshot as far as the transport layer is concerned is that the packet did not make it to the intended destination. There is nothing mysterious about packet loss or packet corruption. Surely, you have experienced sometimes that two hands are not enough to carry all the grocery bags from shopping. Occasionally, you may have even dropped an item or two without even realizing until a friendly shopper points that out to you. We are dealing with physical resources in the network, and as we will see when we discuss the network layer, we may run into capacity limits of these resources during peak load leading to packet loss. Similarly, electrical interference could result in bit errors during transmission. It should be mentioned that such bit errors do not always make the packet completely unusable. As we will see towards the end of this section, it is possible to do error correction on packets, usually referred to as *Forward Error Correction (FEC)* when such bit-errors occur. However, when the errors are beyond the fixing capabilities of FEC algorithms, the packet is as good as lost. The protocol should have a facility for recognizing such packet loss. One possibility is to use *positive acknowledgement* as shown in Figure 13.8. I am sure many of you may have used on occasion the postal service's feature of "acknowledgement due" while sending mail. The postal service returns to the sender some proof that her mail was successfully delivered to the recipient. To disambiguate the acknowledgements from one another, the postal service gives a *unique id* to each such mail for tracking purposes. Positive acknowledgement in the transport layer is akin to this service. In this context, it is important to mention an important parameter of the transport layer, namely, *round trip time* (*RTT* for short). We define *round trip time (RTT)*, as the time taken for a small (say, zero byte) message to be sent from the sender to the receiver and back again to the sender (see Figure 13.8). RTT serves an estimation of the cumulative time delay experienced at the sender for sending a packet and receiving an acknowledgment, and is used in selecting timeout values for retransmission as we will see shortly (see Section 13.6.1). RTT depends on a number of factors including the distance between the sender and receiver, queuing delays in the network en route, and message processing overheads at the sender and receiver.

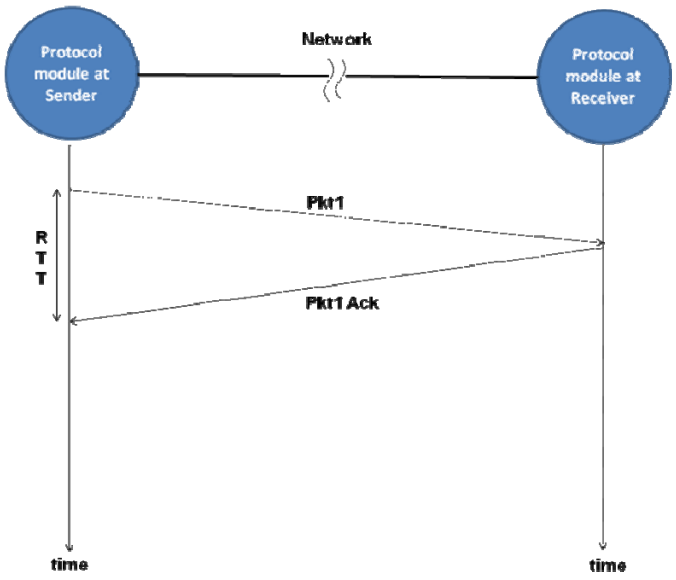



**Figure 13.8: Packet-level positive acknowledgement.**

However, there are some important differences. First, the post office delivers the acknowledgement to the end user. The analogous entity to the post office is the transport layer, and the acknowledgement stops there and does not go up to the application. Second, in the case of the postal service, the message is delivered in its entirety and there is just one acknowledgement for the entire message. In the case of a network message, the transport layer breaks up the message into several packets. There are a number of choices on how the transport layer deals with acknowledgements, which gives rise to a plethora of transport protocols.

### 13.6.1 Stop and wait protocols

A simple approach is to do the following:

1. The sender sends a packet and waits for a positive acknowledgement, commonly referred to as ACK.
2. As soon as a packet is received, the recipient generates and sends an ACK for that packet. The ACK should contain the information for the sender to discern unambiguously the packet being acknowledged. Sequence number is the unique signature of each packet. Thus, all that needs to be in the ACK packet is the sequence number of the received packet.
3. The sender waits for a period of time called *timeout*. If within this period, it does not hear an ACK, it *re-transmits* the packet as shown in Figure 13.9. Similarly, the destination may re-transmit the ACK, if it receives the same packet again (an indication to the receiver that his ACK was lost en route, see Figure 13.10).

We refer to such a protocol as **stop-and-wait**, since the sender *stops* transmission after sending one packet and *waits* for an ACK before proceeding to send the next packet.

**Example 1:**
How many packets need to be buffered in the protocol stack of the sender for reliable transport in a stop-and-wait protocol regime?

**Answer:**
**The answer is 1.** The sender side protocol stack sends one packet at a time and waits for an ACK.

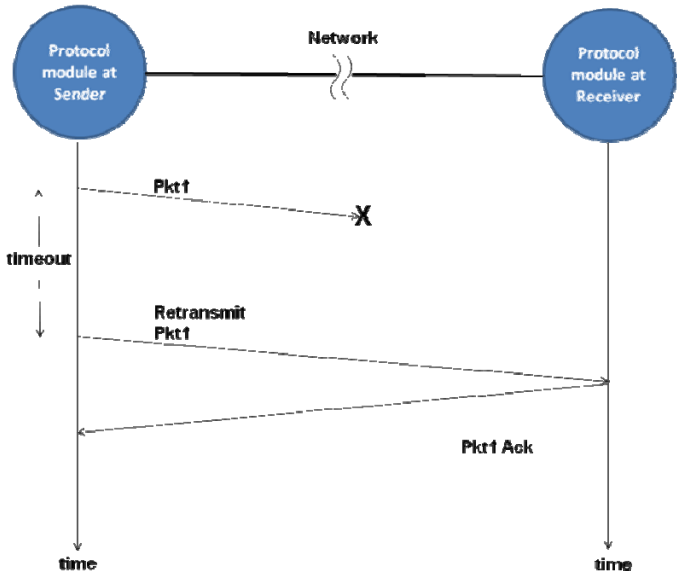



**Figure 13.9: Source retransmits if acknowledgement is not received within a timeout period.**

Let us understand why need sequence numbers in the first place. After all, the sender will not proceed to send the next packet until it hears an ACK for the current packet that it has already sent. The reason is quite simple and intuitive. Packet loss can happen for the data and ACK packets. Therefore, both the sender and the receiver have the mechanics in the protocol for retransmitting a data or an ACK packet after a certain timeout period. How will the sender know if a received ACK is for the current packet or a duplicate ACK for the previous packet? This is the role of the sequence number. By giving a monotonically increasing sequence number, the sender can determine if an incoming ACK is for the current packet or a duplicate ACK.

Let us see if we can simplify the sequence number associated with each packet for this protocol. At any point of time, there is exactly *one* packet that is in transit from the source to the destination. Given this protocol property, do we really need a monotonically increasing sequence number for the packets? Not really, since the packets by design of the protocol arrive in order from source to destination. The purpose of the sequence number is simply to eliminate duplicates. Therefore, it is sufficient if we represent the sequence number by a *single bit*. The protocol sends a packet with sequence number 0 and waits for an ACK with sequence number 0. Upon receiving an

ACK with sequence number 0, it will proceed to send the next packet with sequence number 1 and wait for an ACK with a sequence number 1. For this reason, the stop-and-wait protocol is also often referred to as an *alternating bit protocol*.

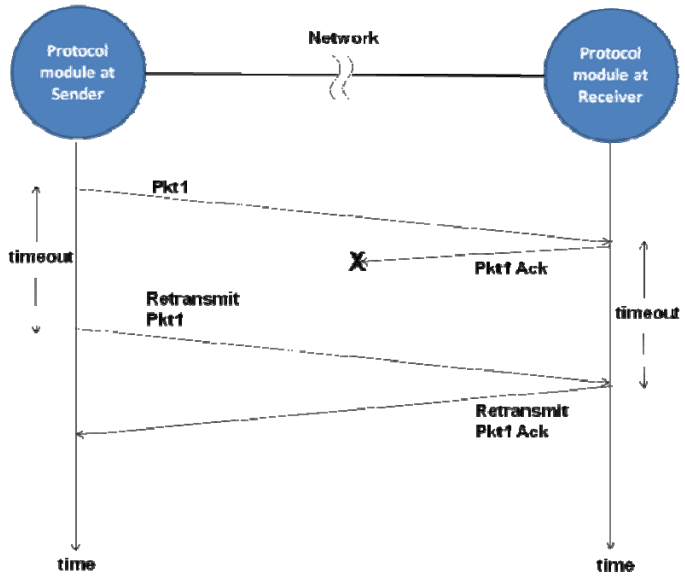



**Figure 13.10: Destination retransmits the acknowledgement for previous packet if it receives the same packet.**

Let us see how we can choose the timeout parameter. From our commuting experience, we know that the time it takes to come to school may be different from the time it takes to go home from school. We may take a different route each way; the traffic conditions may be different, and so on. The same is true of message transmission. The paths taken by a message from the sender to the receiver may be different from that taken in the opposite direction. Further, the queuing delays experienced in the network may be different in the two directions as well. All of this may lead to asymmetry in the measured times for message traversal in the forward and backward directions between two hosts on the network. This is the reason RTT is useful than one-way message transmission time. We will discuss message transmission time in more detail later on (see Section 13.11.1). At this point, we just want to point out that the timeout parameter has to have a value larger than the expected RTT.

Figure 13.11 shows the flow of data and ACK packets between the sender and the receiver for the stop-and-wait protocol. In this figure, RTT is the round-trip time for a message. The sender has to wait for RTT units of time after transmitting a packer to receive an ACK. Then it is ready to send the next packet and so on.

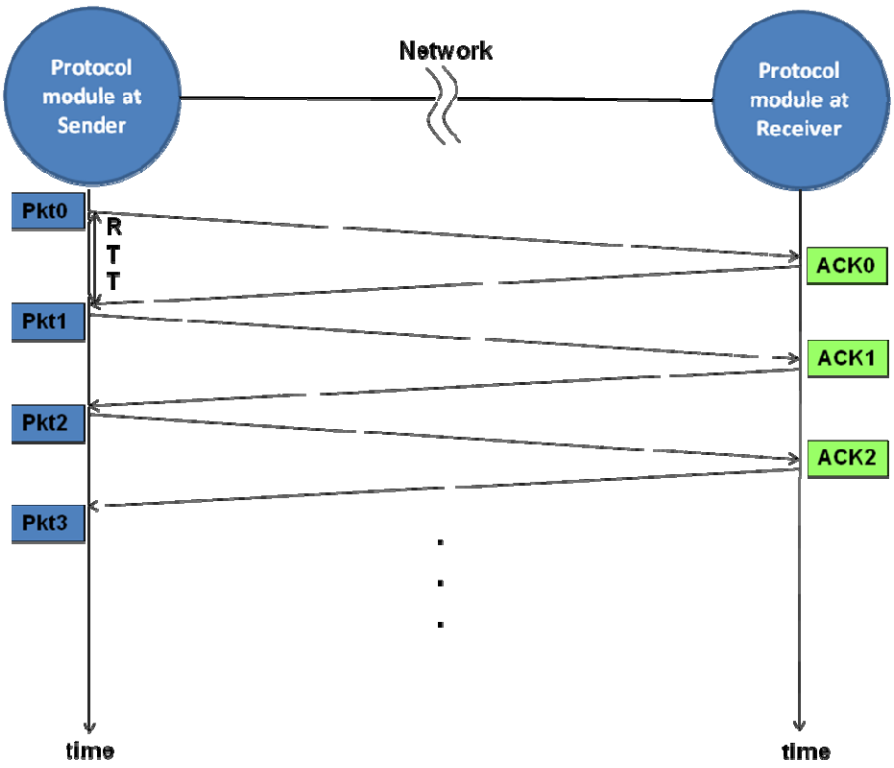



**Figure 13.11: Timeline of packet sending in Stop-and-wait protocol**

---

**Example 2:**

A message has 10 packets, and the RTT for a packet is 2 msec. Assuming that the time to send/receive the packet and the ACK are negligible compared to the propagation time on the medium, and no packet loss, how much time is required to complete the transmission with the stop-and-wait protocol?

**Answer:**

In this example, **RTT** = 2 msec.
Therefore, the total time for the message transmission = 10 * RTT
= 10 * 2 msec
= **20 msec**

---

### 13.6.2 Pipelined protocols

The virtue of the stop-and-wait protocol is its simplicity. However, you will not be happy with this protocol if you are downloading a movie from your buddy in India on the Internet. Look at Figure 13.11. There is so much *dead time* on the network while the sender is waiting for the ACK to arrive. Dead time is defined as the time when there is no activity on the network. Suppose we have Gigabit connectivity to our computer, we could be sending 1 Gigabit of data every second while the network is idle. The flaw in the above protocol is the presumption that packet loss is the norm as opposed to an exception. For example, in the above example, the RTT is 2 ms. In other words, the transport layer sends 1 packet every 2 ms, yielding a transport level throughput of 500

packets/sec. Suppose each packet has a data payload of 1000 bytes, the transport level throughput is 4 Mbits/sec. In other words, the Gigabit connectivity is heavily under-utilized, since we are using only 0.4% of the available network bandwidth. If the network is reliable (i.e., there is no packet loss), we will blast all the packets of the message one after another without waiting for any ACKs.

For example, if the network is reliable (i.e., there is no packet loss), then we can *pipeline* the packet transmission without waiting for ACKs as shown in Figure 13.12.


**Figure 13.12: Pipelining packet transmission with no ACKs**

It is important to make a distinction between bandwidth and propagation time at this point. The bandwidth determines the amount of time it would take the host to place a packet on the wire. The propagation time pertains to the end-to-end delay for a packet to reach the destination, which is a cumulative function of the propagation delays and queuing delays en route to the destination. We will re-visit and have a more precise definition of these terms later in this chapter (see Section 13.12).

---

**Example 3:**

A message has 10 packets, and the time to send a packet from source to destination is 1 msec. Assuming that the time to send/receive the packet is negligible compared to the propagation time on the medium, and no packet loss, how much time is required to complete the transmission with the pipelining with no-acks protocols?

**Answer:**

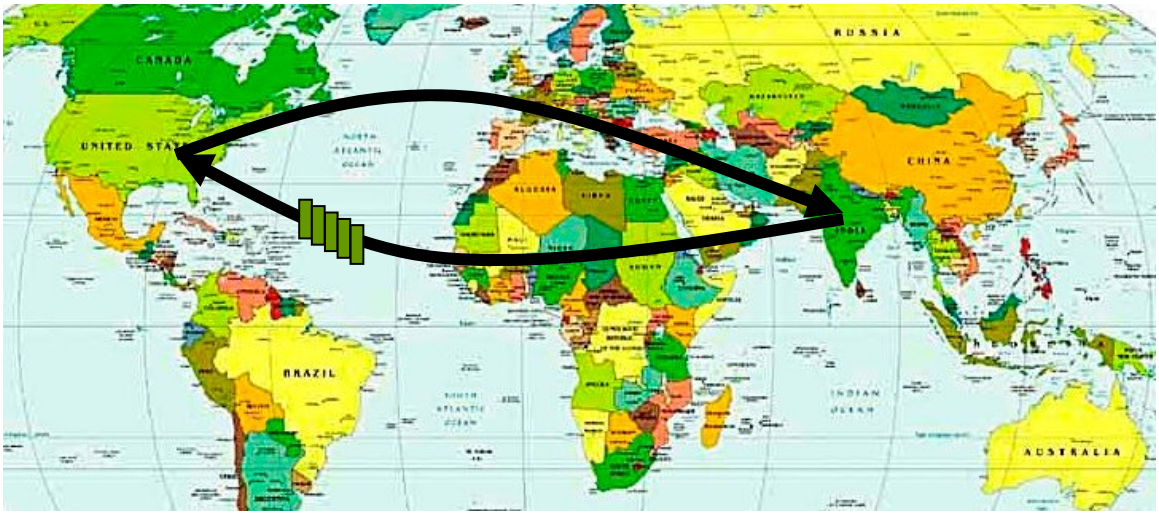
This is similar to the earlier example with the difference that the packets are pipelined as shown in Figure 13.12. Time to form the packets and place them on the wire is negligible, so the time for all the packets to reach the destination is just the source to destination end-to-end delay.

Total time for transmission = **1 msec**

---

This extreme example, though unrealistic shows the importance of pipelining the packets, especially when there is a huge latency from source to destination. Figure 13.13 shows pictorially, the difference between stop-and-wait and pipelined protocols. In Figure 13.13-(a), only one data packet is in transit at any point of time, while in Figure 13.13-(b) multiple data packets are in transit.


(a)




(b)

**Figure 13.13: Difference between stop-and-wait and pipelined transmission[5]**

Figure 13.12 begs the question do we even need ACKs? The answer to this question really depends on the service guarantee needed by the application from the transport layer. For example, some applications may not need reliable delivery. As we will see shortly (Section 13.6.5), UDP is a transport protocol that does not use ACKs and will be suitable for such applications. However, as we mentioned before, some applications may need a reliable transport just as one would use "acknowledgement due" with normal postal service. For such applications, we cannot assume that the network is reliable to the extent of doing away with the ACKs altogether.

### 13.6.3 Reliable Pipelined Protocol

An intermediate approach between the two extremes of *stop-and-wait* and *pipelining with no-acks* is to pipeline the sending of the packets and the reception of the ACKs. The source sends *a set of packets* (called a *window*) before expecting an acknowledgment.

[5] World map courtesy of http://www.hawaii.edu/powerkills/WF1.WORLD.JPG

The destination acknowledges each packet **individually** as before but the good news is that the sender does not have to wait for acknowledgements for all the outstanding data packets before starting to send again. Figure 13.14 shows this situation pictorially.

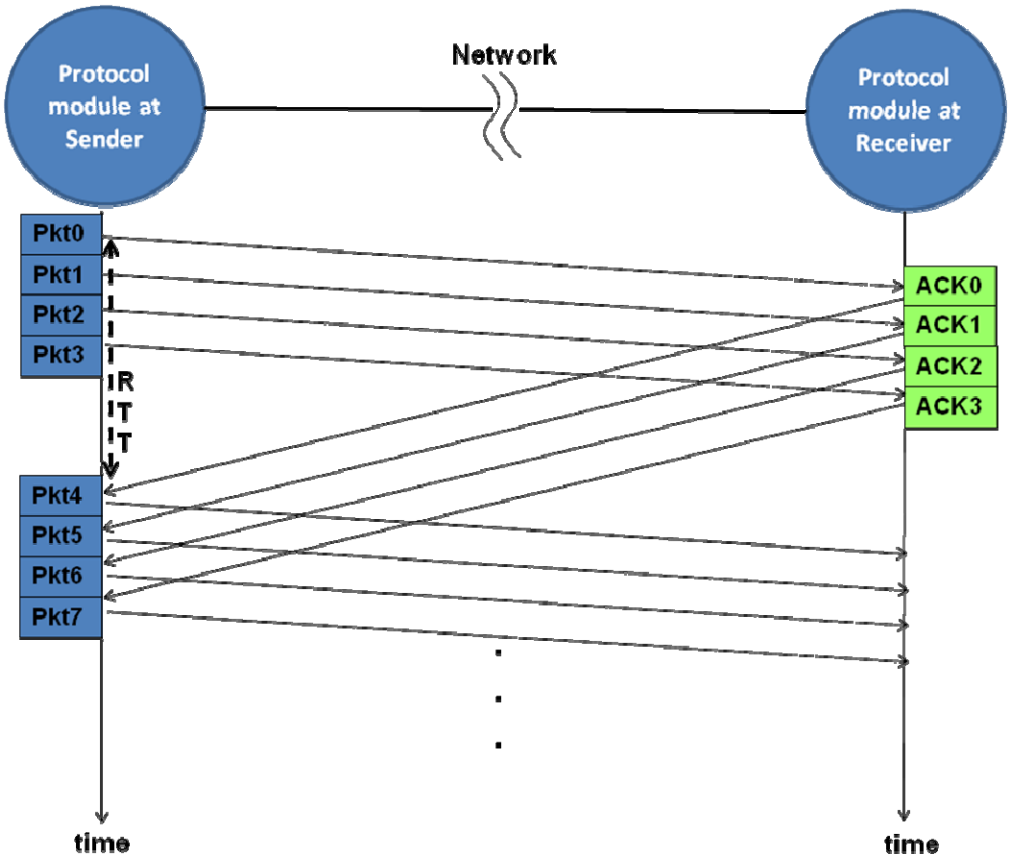



**Figure 13.14: Reliable Pipelined Transmission with Acks (window size = 4)**

As seen in Figure 13.14, the source (with a window size of 4) sends 4 packets and waits for ACKs; as soon ACK0 is received, it is able to send Pkt4. In an ideal situation, where there is no loss of packets, the source repeats the above cycle. That is, the source sends Pkt4-7 and then waits; upon receiving ACK4, it sends the next four packets Pkt8-11, and so on until the message transmission is complete.

A packet transmission is complete once the source receives the corresponding ACK for that packet. The size of the window may be a mutually agreed upon parameter of the protocol between the sender and the receiver, or dynamically adjusted by the sender based on the observed *network congestion*.

### 13.6.3.1 Network Congestion

Let us understand what exactly network congestion is and why it happens.
As a simple analogy, consider why the highway starts backing up at certain times of the day, for example at rush hour. Basically, there are several feeder roads or surface streets that are putting traffic on the highway. At the other end of the commute, folks are getting off the highway into surface streets again. Added to this, highways from different

directions may merge as they come to the heart of the city. One of several reasons lead to the highway backing up. Cumulatively, the surface streets are trying to put more cars on the highway than can be sustained. At the other end of the commute, the cars that want to get off the highway are not able to due to the limited capacity of the surface streets (number of lanes, speed limit, etc.) are smaller than that of the highway. When two highways merge into one, essentially the number of lanes of traffic shrinks relative to the density of cars.

Network congestion happens for almost exactly the same reasons. Consider the following figure:


There are four 1 Gbps network flows coming into the pipe that can support up to 10 Gbps. Even if all the four network flows are coming in at full throttle the 10 Gbps fat pipe can sustain their requirements. However, if there are 20 such flows coming into the fat pipe, you can quickly see that the network cannot sustain that rate and so the individual network flows are going to start backing up. This is the reason and the manifestation of network congestion. The upshot of network congestion is a buildup of the *packet queues* in the routers. Further, the packets no longer have a straight pass through the routers since they sit in the queues of the routers *en route* to the destination. This results in unpredictable *queuing delays* exactly similar to what we may experience on the highways during peak traffic. Further, if the hardware queues in the routers fill up it will lead to routers simply dropping the packets due to lack of buffer space leading to *packet losses*.

You may be wondering why one would design the network in this manner that could result in such congestion. The answer is pretty simple. As we have already seen, the Internet is a network of networks. Returning to our example of sending a message from Atlanta to Bangalore (Figure 13.13), the message crosses so many network links that are part of different networks from source to destination. At the sending end, you may have a Gigabit connectivity, the core of the network may be able to sustain several such Gigabit links, but ultimate destination in Bangalore may have only a slow dial-up connection to the Internet. This is analogous to the slow surface streets and the fast highways. Unless we have all the links having the same bandwidth end-to-end, we cannot avoid congestion in the network.

Each of us may deal with traffic congestion that we may encounter on the highway in our own unique way. Take an exit; get a cup of coffee; stretch our legs for a little, and so on, before we resume our travel. A transport protocol may do something similar to combat network congestion. For example, it may self-regulate the amount of data it hands to the network layer depending on the observed congestion in the network. The ubiquitous Internet transport protocol, TCP, does precisely that, and in this sense, works for the

common good, a socialistic protocol! The basic idea is that if everybody is equally well behaved, then every network flow will get a fair share of the available network bandwidth commensurate with the available bandwidth and current traffic conditions. In other words, a well-behaved transport protocol throttles its own contribution to the network load to ensure that it is using its own fair share among all the competing network flows.

Naturally, there is a downside to this socialistic approach. A protocol that incorporates congestion control, cannot guarantee how soon the data will get to the intended recipient. For example, if other transport protocols do not follow such self-regulation then a well-behaved protocol, being the nice guy, will end up a loser! You have seen smart aleck drivers on the highway, who sensing a lane closing up ahead forge ahead on the lane that is ending all the way to the closing point, and jump into the working lane in the last minute! In other words, there is no *upper bound for delay* that may be experienced when using such a well-behaved transport protocol. Another way of saying the same thing is that there is not even a *minimum guaranteed transmission* rate when using such a protocol. You may have experienced widely varying wait times when trying to access information on the web, since TCP, which is the underlying transport protocol for web-based applications, is such a well-behaved protocol that incorporates congestion control. For this reason, network applications (such as video and audio) that need to provide real-time guarantees may choose to use UDP and provide their own reliability on top of it.

### 13.6.3.2 Sliding Window

The window size serves as the mechanism for self-regulation in a transport protocol that incorporates congestion control. The window size restricts the sender's rate and in turn prevents the buildup of queues in the routers thus mitigating network congestion.

As we know, the source breaks up the message into a set of packets each with its own unique sequence number. Thus, for a given window size we can define an *active window* of sequence numbers that corresponds to the set of packets (with those sequence numbers) that the source can send without waiting for ACKs. This is shown pictorially in Figure 13.15[6]. Looking at this figure, one might wonder what decides the *width* of each packet. The width represents the time it takes at the sender to push a packet out of the computer on to the network. As a first order of approximation, we can say that it is simply the ratio of packet size to the bandwidth of the network interface. For example, if you have a Gigabit/sec full-duplex network interface, and if the packet size is 1000 bytes, the width of each packet is 8 microseconds. The width is important as it tells us how many packets we can fit within a RTT. In other words, the width of the packet gives us an upper bound for the maximum window size that is reasonable to use for a given RTT. For example, if the RTT is 2 milliseconds, then the maximum window size can be 250 packets (each packet is 1000 bytes), assuming that the ACK packets have negligible width. The actual window size may be chosen to be smaller than this upper bound for several reasons. These include network congestion (see Section 13.6.3.1), buffer space for packets at the sender and receiver, and the size of the field used to denote the packet

[6] This figure is inspired by a similar figure that appears in the book by Kurose and Ross, "Computer Networking: A top down approach featuring the Internet," Addison-Wesley.

sequence number in the header. We re-visit message transmission time in much more detail in a later section (See Section 13.11.1).

As soon as an ACK for the first red packet in the active window is received, the active window moves one-step to the right (the first white packet becomes a blue packet). Since the active window slides over the space of sequence numbers (from left to right) as time progresses, we refer to this as a *sliding window protocol*. While this discussion presents sequence numbers as monotonically increasing, as a practical matter, the space of sequence numbers is circular and wraps down to 0 in any real implementation.

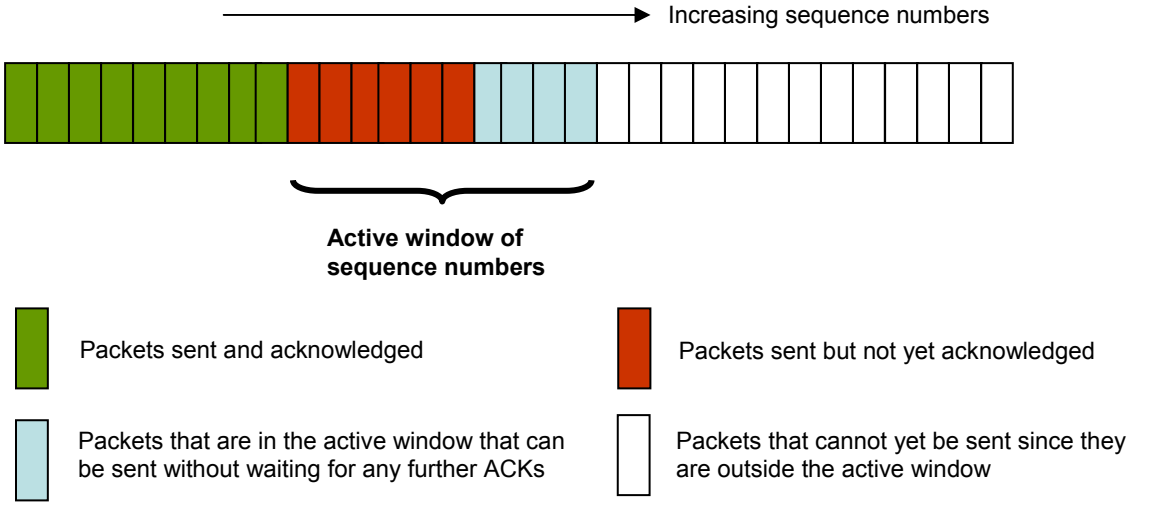



**Figure 13.15: Active window (window size = 10) of a sliding window protocol**

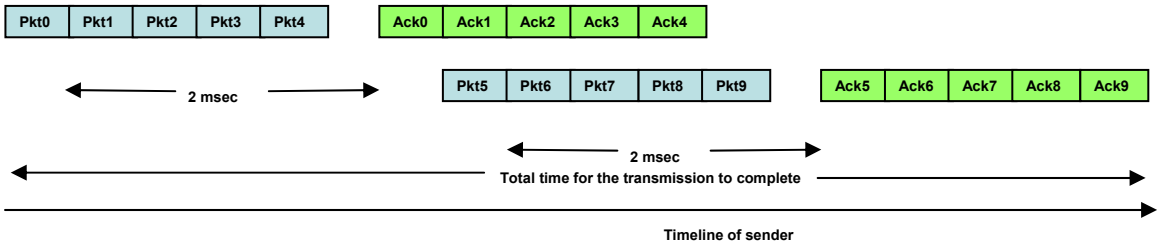
---

### Example 4:

A message has 10 packets, and the RTT is 2 msec. Assuming that the time to send/receive the packet and the ACK are negligible compared to the propagation time on the medium, and no packet loss, how much time is required to complete the transmission with the sliding window protocol with a window size of 5?

### Answer:

The following figure shows the timeline to complete the transmission:




In this example, **RTT** = 2 msec. The source sends a window of 5 packets and then waits for the ACK. The first ACK is received 2 msec after the first packet is sent. Therefore, in one cycle of 2 msec (RTT), the source has successfully completed transmission of 5

packets (since we are ignoring all other times except the propagation time on the medium). Two such cycles are needed to complete the transmission.

The total time for the message transmission = 2 * RTT = 2 * 2 msec = **4 msec**

We can see that the chosen window size specifies the **maximum** number of packets that can be outstanding at any point of time. For example, if instead of 10 (as in the above example) we need to send 12 packets then 3 cycles of message transmission will be needed. In the third cycle, the last two packets will be sent out to complete the message transmission (see a variation on the above example in the exercises).

An optimization that is used quite often in transport protocols to reduce the number of ACK packets is to aggregate the acknowledgments and send a *cumulative ACK*. The idea is simple and intuitive. The sender sends a window of packets before waiting for an ACK from the receiver. Let us say the receiver, receives *n* packets with consecutive sequence numbers. Instead of sending an ACK for each of the *n* packets, a single ACK for the $n^{th}$ packet is sent. The semantics of the protocol allows this optimization since the reception of an ACK for the $n^{th}$ packet implies the successful reception of the *n-1* packets preceding the $n^{th}$ packet. TCP uses such cumulative ACKs to reduce the overhead of network transmission.

As we mentioned earlier, packets (data or ACK) may be lost. In this case, both the source and destination have to be ready to retransmit the lost packets and ACKs, respectively. The source and destination use a *timeout* mechanism to discover packet losses. The basic idea is for each side to set a timer upon sending a packet. For example, if the source does not receive an ACK for the packet within the timeout period, it will retransmit the packet. Naturally, the source has to *buffer* the packets (red ones in Figure 13.15) for which acknowledgements have not yet been received.

There are a number of details to be taken care of in designing a sliding window protocol. We already mentioned buffering and timeout-triggered retransmissions. Additional details of the protocol include:

- Choosing appropriate values for the timeouts
- Choosing the right window size
- Dealing with packets that arrive out of order
- Deciding when to acknowledge packets, including cumulative acknowledgement for a set of packets, and moving the active window (see Exercise 19 at the end of the Chapter)

Such details are outside the scope of our discussion and left to more advanced courses in computer networking[7].

[7] For an elaborate discussion on these topics, please see Kurose and Ross, "Computer Networking: A top down approach featuring the Internet," Addison-Wesley.

**Example 5:**

Assume that the network loses 1 in 5 packets on an average. For a message that consists of 125 packets, determine the total number of packets sent by the sender to successfully complete the message transmission.

**Answer:**

We would expect to lose 20% (1/5) of the 125 packets, i.e., 25 packets. When we resend those packets, we will expect to lose 5 of those, etc.

| Packets Sent | Lose | Successful |
|---|---|---|
| 125 | 25 | 100 |
| 25 | 5 | 20 |
| 5 | 1 | 4 |
| 1 | 0 | 1 |
| 156 | | 125 |

Total number of packets sent to successfully complete the transmission = **156**

### 13.6.4 Dealing with transmission errors

First, the destination has to be able to tell by inspecting the packet that there has been an error in the transmission. To enable the destination to identify "good" from "bad" packets, the source computes a *checksum*, a value that is based on the actual contents of the packet and appends it to the end of the packet. The computation of the checksum can be very simple to very sophisticated. For example, the checksum in Internet is often computed simply as the sum of the data bytes (viewed as 16-bit integers). The destination does the same computation and compares the result to the checksum in the received packet to catch erroneous packets. It is also conceivable to use *error correcting codes (ECC)* to not only detect errors in the packet but repair the packet errors. Such discussions are once again deferred to more advanced courses in computer networking[5]. However, if a packet is corrupted beyond repair it is as good as lost in the network. It is the responsibility of the transport layer to recognize and take corrective action when this happens. In other words, *Forward Error Correction* (FEC) gives an ability to repair packets but it is not a guarantee. Therefore, transport protocols necessarily have to resort to timeouts and retransmissions to overcome transmission errors.

It is interesting to note that in the above description, the size of a packet is the *only* network specific information used. In other words, the above description *abstracts* out the details of the network itself. Thus, the transport layer of the protocol stack provides all the functions described in this subsection that lie *above* the network level.

### 13.6.5 Transport protocols on the Internet

Transport protocols in use on the Internet may be grouped into two broad categories: *connection-oriented* and *connection-less*. TCP (Transmission Control Protocol) is an example of the former and UDP (User Datagram Protocol) of the latter.

**TCP**

TCP involves first setting up a connection between the two endpoints of communication. Once such a connection is set up, the actual data flow between the two endpoints is a *stream* of bytes, i.e., TCP does not deal with messages but streams. For example, let us say you request a web page from CNN using your web browser. The web browser application creates a TCP connection with the web server at CNN (or its proxy). Once the connection is made, the client sends a series of requests first, and the server responds by sending a series of objects in return that form the web page. The requests and responses between the client and the server appear as a stream of bytes at the TCP transport level. Once the entire page has been sent, the two sides may decide to tear down the connection.

TCP connection is a *full duplex* connection, i.e., both sides can simultaneously send and receive data once the connection is set up. Even though one or the other endpoint may initiate the connection set up, the connection is symmetric once established. TCP provides the following major functionalities to facilitate information flow between the two endpoints:

- **Connection set up**: During this phase, the two sides negotiate the initial sequence number for transmission using a three-way handshake:
    - Client sends the server a connection request message (which has a special field to indicate that it is a connection request), with information about the initial sequence-number it plans to use for its data packets.
    - The server sends an acknowledgement message to the connection request (once again, with the special field to indicate it is part of the connection establishment 3-way handshake), with information about the initial sequence-number the server plans to use for its data packets.
    - The client allocates resources (packet buffers for windowing, timers for retransmission, etc.) and sends an acknowledgement (which is the final leg of the three-way handshake). Upon receiving this acknowledgement, the server allocates resources for this connection (packet buffers for windowing, timers for retransmission, etc.).

  At this point, the client and server side of the newly established TCP connection are ready to exchange data.
- **Reliable data transport**: During this phase, the two endpoints can send and receive data. The protocol guarantees that the data handed to it from the upper layers will be faithfully delivered in order to the receiving end without any loss or corruption of the data.
- **Congestion control**: During the data transport phase, the sender also self-regulates its flow by observing the network congestion, and dynamically adjusting its window size to avoid the buildup of queues in the routers and thus reducing the network congestion as detailed in 13.6.3.1. For this reason, TCP flows could experience unbounded delays in the presence of network congestion. This spells problems for real-time traffic that needs deterministic guarantees. Despite this inherent problem, due both to its ubiquity and reliable nature, TCP is used for many real-time traffic flows.

- **Connection teardown**: During this phase, the two endpoints agree to tear down the connection as follows:
    - Client sends a connection teardown request (with a special field to distinguish it from normal data) to the server. The server sends an ACK.
    - The server sends its own teardown connection request (with a special field to distinguish it from normal data) to the client. The client sends an ACK. The client de-allocates client-side resources associated with this connection. Upon receiving the ACK, the server de-allocates the server-side resources associated with this connection. The connection is officially closed.

Although this discussion assumes that the client initiated the teardown, either the client or the server may initiate the connection teardown. When connection teardown is in progress, no new data will be accepted for transport on this connection. However, all data already accepted for transport previously on this connection will be reliably delivered before the connection is closed.

**UDP**

UDP sits atop IP and provides an unreliable datagram service for applications. As we just saw, TCP is stream-oriented and entails an elaborate handshake to open a connection between the two endpoints before actual communication of information can begin. Similarly, once the communication is complete, there is elaborate handshake to close the connection. Moreover, TCP has a number of advanced features (such as acknowledgements, windowing, and congestion control) to ensure reliable transport over wide area networks, adhering to principles of fairness in the presence of sharing the available bandwidth with other users. Such advanced features translate to overhead for communication. Applications that can function quite adequately with lesser guarantees (*a la* sending a postcard) use UDP since it is faster owing to its simplicity. Further, for applications such as Voice over IP (VoIP) latency of getting the packets to the destination is more important than losing a few packets (since there is no time to recover lost or corrupted packets due to real time constraints). So, the use of UDP has been growing and the current estimate is that 20% of all Internet traffic is UDP.

The cons of UDP of course are several: messages may out of order; messages may be lost, and there is no self-control so UDP flows may be the source of increased congestion in the network. Similar to TCP, UDP does not provide any guarantees (such as upper bound on delays, or lower bound on transmission rate). Table 13.1 summarizes the pros and cons of TCP vs. UDP for networked applications

| Transport protocol | Features | Pros | Cons |
|---|---|---|---|
| **TCP** | Connection-oriented; self-regulating; data flow as stream; supports windowing and ACKs | Reliable; messages arrive in order; well-behaved due to self-policing and reducing network congestion | Complexity in connection setup and tear-down; at a disadvantage when mixed with unregulated flows; no guarantees on delay or transmission rate |
| **UDP** | Connection-less; unregulated; message as datagram; no ACKs or windowing | Simplicity; no frills; especially suited for environments with low chance of packet loss and applications tolerant to packet loss | Unreliable; message may arrive out of order; may contribute to network congestion; no guarantees on delay or transmission rate |

**Table 13.1: A comparison of TCP and UDP**

The above discussion makes it sound as though networked applications that require some real-time guarantees can use neither UDP nor TCP. Actually, this is not entirely true. The application developers of such real-time applications take the features (or lack thereof) of these protocols into consideration so that the user experience is not negatively impacted. For example, applications that serve audio or video, will buffer several minutes of playtime before starting the viewer or the music. Further, they may dynamically increase the buffer capacity to account for network congestion, etc. Table 13.2 shows a snapshot of networked applications and the transport protocol they use.

| Application | Key requirement | Transport protocol |
|---|---|---|
| **Web browser** | Reliable messaging; in order arrival of messages | TCP |
| **Instant messaging** | Reliable messaging; in order arrival of messages | TCP |
| **Voice over IP** | Low latency | Usually UDP |
| **Electronic Mail** | Reliable messaging | TCP |
| **Electronic file transfer** | Reliable messaging; in order delivery | TCP |
| **Video over Internet** | Low latency | Usually UDP; may be TCP |
| **File download on P2P networks** | Reliable messaging; in order arrival of messages | TCP |
| **Network file service on LAN** | Reliable messaging; in order arrival of messages | TCP; or reliable messaging on top of UDP |
| **Remote terminal access** | Reliable messaging; in order arrival of messages | TCP |

**Table 13.2: Networked applications and their transport protocols**

### 13.6.6 Transport Layer Summary

In general, operating systems support several protocol families to cater to the differing communication needs of applications. In the early 80's, the ISO standards body proposed a new transport protocol suite called *ISO-Transport Protocol (TP0 through TP4)* as a potential standard for Internet based communication. However, TP0-TP4 never really took off owing to the ubiquity of TCP. Today, though there are concerns expressed in the networking community as to the appropriateness of TCP for transport on the Internet, it is difficult to dislodge the stranglehold that TCP enjoys as the protocol of choice for many Internet applications.

Still one never knows what the morrow will bring. Projects such as GENI (which stands for Global Environment for Network Innovation) and global networking research infrastructures such as PlanetLab[8] may act as agents of change and bring new and better network protocols for different classes of applications on the Internet.

We have only given a brief overview of the fascinating field of transport protocols. You will learn a lot more about these protocols in a more advanced course on networking.

## 13.7 Network Layer

At first glance, the role of the network layer seems pretty simple and straightforward, namely, send a packet handed to it at the source from the transport layer to the destination; and pass up an incoming packet to the transport layer at the destination. This functionality could be bundled in with the transport layer. Let us understand why this may not be a good idea. First of all, consider how many different network connections you have on your laptop or your home computer. You may have a wired and wireless connection in the very least. In general, different destination hosts may be reachable on different network connections. Second, look at Figure 13.6. The source and destination may not be directly connected to one another but the packet may have to go through several intermediate hops to get to the destination. At these intermediate hops of the network, there is no need for the transport layer functionalities since these intermediate nodes simply *forward* the packet towards its final destination. Third, since we have no control over the evolution of the network, the actual path taken by packets from source to destination, referred to as the *route*, is not fixed. All of these reasons point to the fact that it is best to leave the decision as to how best to get a packet from a source to a destination host to a different layer of the protocol stack. We refer to the layer that implements this functionality as the *network layer*. Such a separation of responsibility allows the transport layer to be independent of any addition/deletion of network connections to a host. The *destination address* and the *packet size* parameterize the interface between the transport and network layers. The network layer is responsible for routing a packet given a destination address. For this purpose, it maintains a table (called *routing table*) that contains a route or a path from this source to any desired destination host[9]. On receiving a packet on the wire, the network layer either forwards the incoming packet on

[8] PlanetLab is an experimental network testbed with contributed network nodes from across the globe to facilitate controlled experiments to be conducted over a wide area network.

[9] Shortly (see Section 13.7.4) we will see that there is another table called the *forwarding table* in the network layer that contains the next hop a packet must take along its route to the destination address.

towards its ultimate destination, or passes it up to the transport layer if this node is the destination for the packet.

Let us understand the functionalities needed in the network layer:

- *Routing Algorithms*: The network layer should determine a route for packets to take from source to destination. Thus algorithms for determining routes, called *routing algorithms*, are the principal functionalities in the network layer. In this section, we will introduce the reader to some well-known routing algorithms widely used in the Internet.
- *Service Model*: The network layer should forward an incoming packet on an incoming link to the appropriate outgoing link based on the available routing information. This is also often referred to as the *switching* function of the networking layer. This functionality is very much tied to the *service model* provided by the network layer to the upper layers of the protocol stack. In this section, we will discuss well-known switching strategies and service models that are commonly employed in networks in general and the Internet in particular.

The device that performs the functionalities of the network layer is called a *router*. For the purposes of the routing algorithms to be discussed in this section, a router is no different from an end host. The only difference is that a router knows that it is not a terminating point for a packet. In this sense, the protocol stack at a router consists only of the physical, link, and network layers.

### 13.7.1 Routing Algorithms

The network is a collection of hosts and routers each with a distinct identity. In the Internet world, the identity is a unique IP address. If the network is fully connected then one could route a packet from any source to any destination in one hop, assuming that the *cost* of sending a packet between any two nodes is the same. However, in reality, (a) the network is not fully connected, and (b) the cost of sending a packet between any two nodes may not be the same. Let us understand what exactly "cost" means here. Ultimately, the goal is to move a packet with minimal latency from point A to point B. "Cost" can be thought of as some summary quantitative metric that is a composite of the latency (which depends on the bandwidth of the connection) for moving a packet between any two points and the network traffic between the two points. An analogy is commuting by car. The "cost" (in terms of travel time) for commuting a given distance depends on both the speed limit *en route* as well as the amount of traffic. The cost will be higher during rush hour. Sometimes it may be more cost-effective to take a seemingly longer route and reach the destination in lesser time.

**Dijkstra's Link State Routing Algorithm**

Figure 13.16 shows a graph representation of a network where the vertices are hosts, and the edges represent physical links incident at a given host. We define a node's *link state* as the cost associated with each physical link at any given node. For example, in Figure 13.16, the link state of node A is {B:2, C:1, D:4, E:5}. That is, it costs 2 unit to move a packet from A to B, 1 unit to move from A to C, 4 units to move from A to D, and 5 units to move from A to E. Similarly, the link state of B is {A:2, C:2, E:1}, and so on.

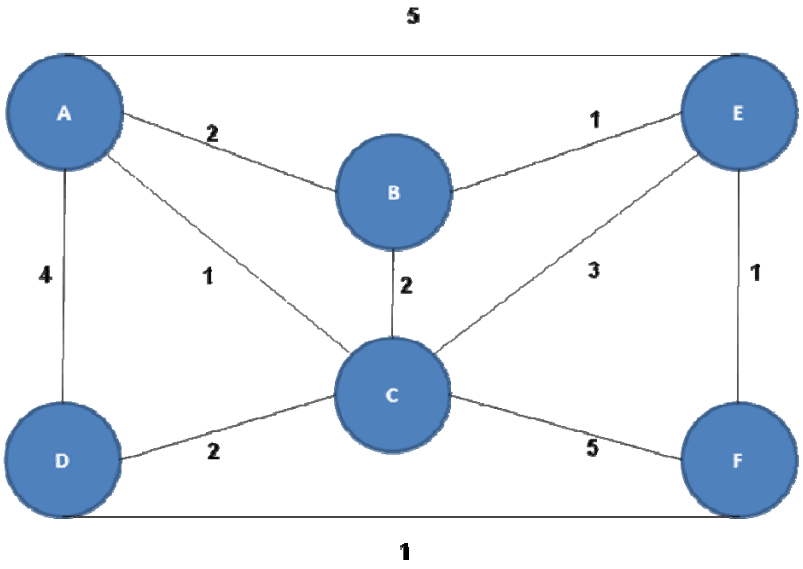



**Figure 13.16: An Example Network**

The Link State (LS) Routing algorithm (due to Dijkstra) is a *local* algorithm that uses *global* information. That is, all nodes in the network have *complete* information about the state of the network (connectivity and costs associated with the links). Given this information, each node can determine an optimal route to send a packet to any destination in the network by running this algorithm locally. Each node gets this information from its peers, since every node, periodically broadcasts its link state to all the other nodes in the network.

The algorithm allows a node to find out iteratively, the least cost distance to every other node in the network. In each iteration of the algorithm, the node discovers a least cost route to one more node in the network. Thus, if there are *n* nodes in the network, the algorithm needs *n-1* iterations to discover least cost routes to all the nodes.

The intuition behind this algorithm is quite simple. Initially, you know the cost from A to its immediate neighbors (B, C, D, and E). Out of these, A-C is the least-cost route (1 unit). The direct link A-D costs 4 units. However, if we go through the least-cost link A-C, then the cost from A to D is only 3units (A-C, then C-D).

The algorithm starts with the known costs of reaching nodes that are directly connected to A. In each iteration of the algorithm, we add a new least-cost route from A to some other node in the network. For example, in the first iteration, using A's link state, we determine one least-cost route, namely, A-C. In the second iteration, we update the cost of reaching other nodes in the network either directly from A or through the newly discovered least-cost route from the first iteration. The algorithm continues in this fashion until all the least-cost routes are discovered. Since the example has 6 nodes, the algorithm goes through 5 iteration steps.

| Iteration Count | New node to which least-cost route known | B Cost/route | C Cost/route | D Cost/route | E Cost/route | F Cost/route |
|---|---|---|---|---|---|---|
| **Init** | A | 2/AB | 1/AC | 4/AD | 5/AE | ∝ |
| **1** | AC | 2AB | **1/AC√** | **3/ACD** | **4/ACE** | **6/ACF** |
| **2** | ACB | **2/AB√** | √ | 3/ACD | **3/ABE** | 6/ACF |
| **3** | ACBD | √ | √ | **3/ACD√** | 3/ABE | **5/ADF** |
| **4** | ACBDE | √ | √ | √ | **3ABE√** | **4/ABEF** |
| **5** | ACBDEF | √ | √ | √ | √ | **4/ABEF√** |

**Table 13.3: Dijkstra's algorithm in action on the graph shown in Figure 13.16**

Table 13.3 summarizes the result of running the algorithm on the example graph shown in Figure 13.16. In each of the row of the table, the highlighted node in the second column is the one to which a least-cost route is known in that iteration. For example, in iteration 1, AC is the least-cost route. The routes that have been updated due to the newly identified least-cost route are shown in bold letters. For example, the routes to D, E, and F are updated in iteration 1. The route to B remains unchanged.

Shown below is the pseudo code for Dijktra's LS routing algorithm, which we informally described using the above example. In this algorithm:

- R denotes the set of nodes to which a least-cost route from A is known
- link-state(X:value) denotes the cost associated with a direct link from A to X
- cost(A->X) denotes the cost of the route from A->X
- route(A->X) denotes the route from A to X possibly through intermediate hops discovered during the running of the algorithm

```
Init:
R = {A}  // set of nodes for which route from A known
cost(A->X) = link-state(X:value) for all X adjacent to A
cost(A->X) = ∝ for all X not adjacent to A

// let n be the number of nodes in the network
for (i = 1 to n-1) {
     choose node X not in R whose cost(A->X) is a minimum;
     add X to R;
     set route(A->X) as the least-cost route from A to X;
     update routes for nodes adjacent to X:
          for each Y not in R and adjacent to X {
               cost(A->Y) = MIN(original cost(A->Y),
                                cost(A->X) + cost(X->Y));
               set route(A->Y); // only if new route
                                // through X is lower cost
          }
}
```

The link-state algorithm described above is also referred to as Dijkstra's *shortest-path* algorithm. Apart from the need for global information at each node, one problem with this algorithm is that there is a pre-supposition of a *synchronous* execution of this algorithm in *all* the nodes of the network so that each node will compute the same least cost paths. In practice, this requirement is hard to impose and the routers and hosts run this algorithm at different times, which could lead to some inconsistencies in the routing decisions. There are algorithmic extensions that ensure that there will not be instability in the network despite such temporary inconsistencies. Discussions of such extensions are beyond the scope of this textbook.

**Distance Vector Algorithm**

Another widely used routing algorithm in the Internet is the Distance Vector (DV) algorithm. By design, this algorithm is *asynchronous* and works with *partial knowledge* of the link-state of the network. Due to the evolutionary nature of the Internet, these two properties make the distance vector algorithm a very viable one for the Internet.

The intuition behind the algorithm is quite simple. Regardless of the destination, any node has to decide to send a packet to one of its immediate neighbors to which it has a physical link. The choice of a neighbor node to send the packet to is quite simple. The neighbor that will result in the least cost route to the desired destination is the one to choose. For example, referring to Figure 13.16, if E wants to send a packet to D, it will choose F as the next hop since among all the neighbors (A, B, C, and F). If we eyeball Figure 13.16, we can immediately see that F has the lowest cost route to reach D. Note that what E needs to make this determination is NOT the actual route to reach D; only the cost of reaching D from each of its immediate neighbors.

Each node maintains a routing table, called a *distance vector* table. This table has the lowest cost route of sending a packet to every destination in the network, through each of the immediate neighbors physically connected to this node. The name "distance vector" comes from the fact that every node has a cost vector of values for reaching a given destination through its immediate neighbors. Table 13.4 shows the DV table for node E for the example network shown in Figure 13.16. Each row shows the least cost route to a particular destination node through each of the immediate neighbors of E. The DV table will contain *only* the cost but for the sake of exposition, we have shown the actual route as well in parenthesis. It is important to understand that the actual route information is not needed for choosing the next hop for a given destination. The least cost route to a destination is shown in gray.

| | Cost through immediate neighbors | | | |
|---|---|---|---|---|
| **Destination** | A | B | C | F |
| **A** | 5(EA) | 3(BA) | 4(ECA) | 5(EFDCA) |
| **B** | 7(EAB) | 1(EB) | 5(ECB) | 6(EFDCB |
| **C** | 6(EAC) | 3(EBC) | 3(EC) | 4(EFDC) |
| **D** | 8(EACD) | 4(EBEFD) | 5(ECD) | 2(EFD) |
| **F** | 9(EABEF) | 2(EBEF) | 7(ECBEF) | 1(EF) |

**Table 13.4: DV table for Node E**

We will just give an informal description of the DV algorithm that is used by each node to construct this table, and leave it to the reader to formalize it with appropriate data structures. Each node sends its least cost route to a given destination to its immediate neighbors. Each node uses this information to update its DV table. A node recomputes its DV table entries if one of two conditions hold:

(1) The node observes a change in the link-state for any of the immediate neighbors (for example, due to congestion, let us say the link-state from E to B becomes 5 from 1).

(2) The node receives an update on the least-cost route from an immediate neighbor.

Upon such recomputation, if any of the table entries changes, then the node will communicate that change to its immediate neighbors. The core of the algorithm as stated above is pretty simple and straightforward. We will leave it as an exercise to the reader to formalize the algorithm with pseudo code.

The reader can immediately see the asynchronous nature of this algorithm compared to Dijkstra's link-state algorithm as well as the ability of the algorithm to work with partial knowledge. The state of the network is changing continually (traffic patterns of current network flows, new network flows, addition/deletion of nodes and routers, etc.). One might question the point of computing routes based on the *current state* given this continuous evolution of the network state. Fortunately, these algorithms have good converge properties that ensure that the route calculation is much quicker than the rate of change of the network state.

Over time, there have been numerous proposals for efficient routing on the Internet but LS and DV algorithms hold the sway as the exclusive algorithms for routing on the Internet.

**Hierarchical Routing**

Perhaps, the reader is wondering how the Internet can possibly operate with either the LS or the DV algorithm for routing given the scale of the Internet (millions of nodes) and the geographical reach. Both of these algorithms treat all the nodes on the Internet as "peers". In other words, the network is one monolithic entity wherein all nodes are equal. Such a flat structure will not scale to millions of nodes. In any organization (political, business, religion, charity, etc.), hierarchy is used as a way of reining in the chaos that

could result when the size of the organization grows beyond a threshold. Internet uses the same principle. Perhaps an equally compelling reason to bring some structure to the Internet, in addition to the scale of the Internet, is the need for administrative control. Especially in this day and age with the increase in SPAM, an organization may want to control what network traffic is allowed to come in and go out of a corporate network. Both the twin issues of scale and administrative control are dealt with nicely by organizing the routers on the Internet into regions called *Autonomous Systems (ASs)*. Routers within an AS may run one of LS or DV protocols for routing among the hosts that are within an AS. Naturally, one or more routers within an AS has to serve the need for communicating with destinations outside the AS. Such routers are called *gateway routers*. The gateway routers of different ASs communicate with one another using a different protocol called *Border Gateway Protocol* or *BGP* for short. Nodes within an AS do not care nor are they affected in any way by the evolution/attrition/expansion of nodes within other ASs.

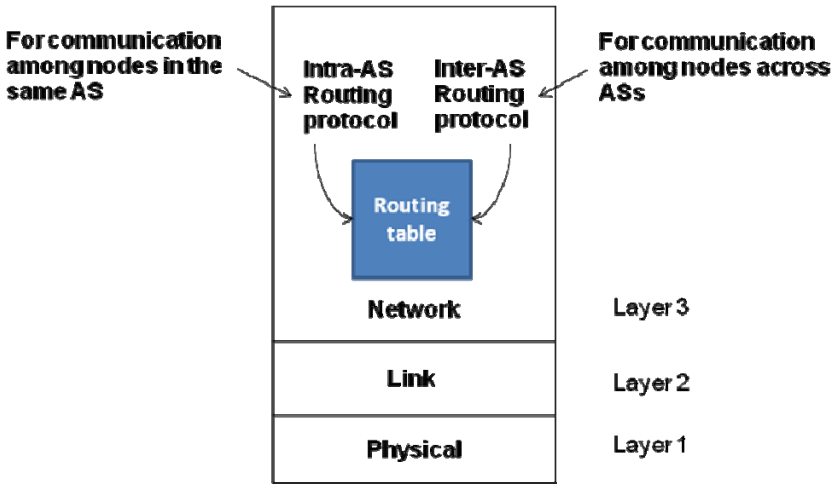



**Figure 13.17: Details of the network layer in a gateway node**

Thus, as shown in Figure 13.17, the network layer in a gateway node supports at least two protocols: one for inter-AS communication and one for Intra-As communication.

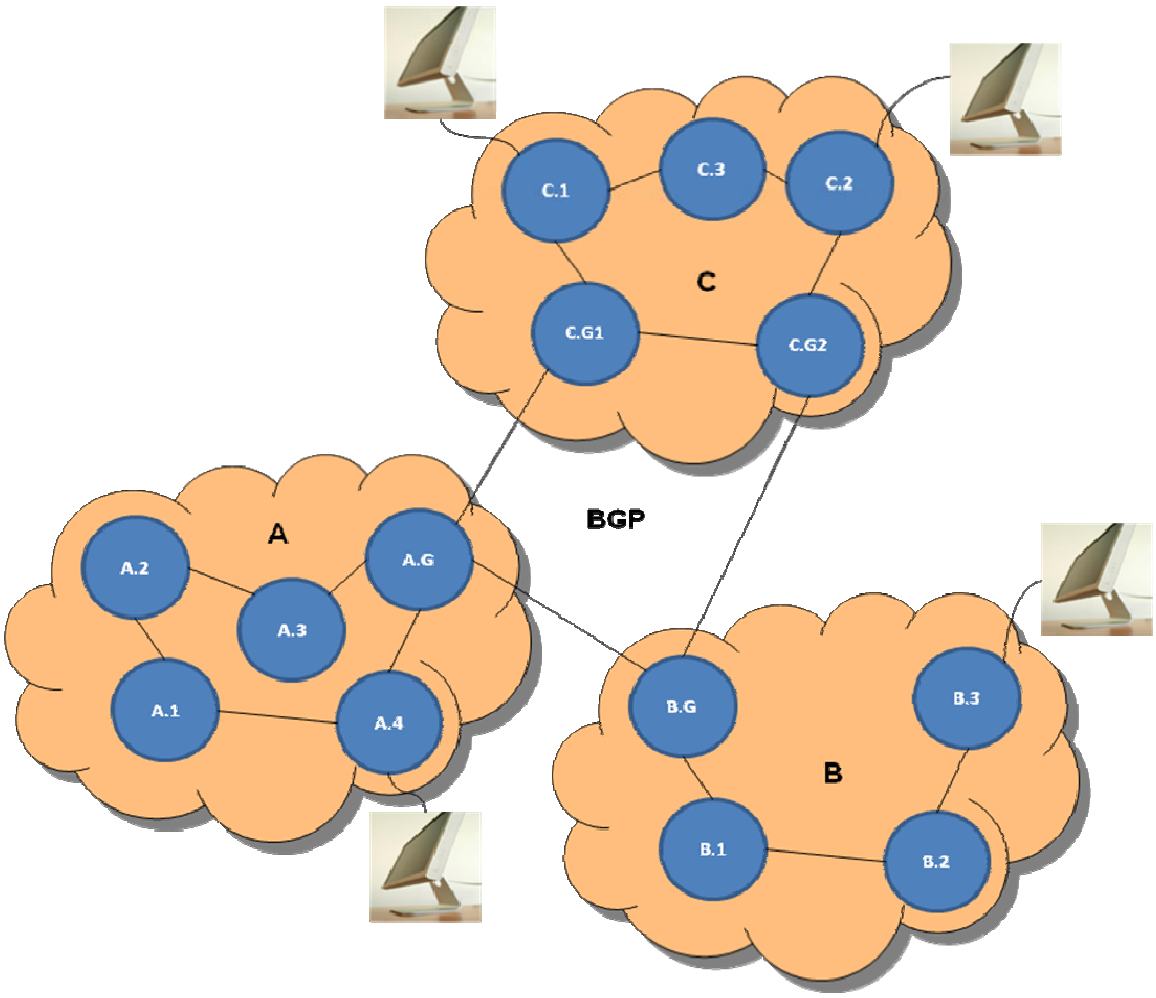



**Figure 13.18: Three different ASs coexisting with Intra- and Inter-AS protocols**

Consider the host connected to router C.1 needing to communicate with the host connected to router C.2. In this case, the communication is accomplished using whatever intra-AS protocol (LS, DV, or some other variant) is in use in the autonomous system C. On the other hand, consider that the host connected to router A.4 needs to send a packet to the host connected to router B.3. The routing table in A.4 knows that to send packets outside autonomous system A, they have to be sent to A.G. The routing table in A.G recognizes that to reach any destination in the autonomous system B, the packets have to be sent to B's gateway node B.G. A.G uses BGP to communicate with B.G, whence the packet is routed using the intra-AS protocol in effect within autonomous system B to node B.3. The routers A.G, B.G, C.G1, and C.G2 will each have a protocol stack as shown in Figure 13.17 that allows these routers to talk to nodes within their respective autonomous systems, as well as with the gateway nodes in other autonomous systems using BGP.

Note that the network layer in A.4 is completely oblivious of the internal organization of the autonomous system B. For that matter, even A's gateway node A.G is unaware of the specific intra-AS protocol that may be in effect within the autonomous system B.

Details of the BGP protocol itself are beyond the scope of this book[10].

### 13.7.2 Internet Addressing

So far, we have used the term "node" to signify either a host or a router. However, a host and a router are fundamentally quite different. As we mentioned earlier, a host is an end device at the edge of the network and typically has a single connection to the network through its Network Interface Card (NIC). On the other hand, a router (see Figure 13.18) allows several hosts to be connected to it. It serves as an intermediary to route an incoming message on one connection to an appropriate output connection towards the intended destination. Therefore, typically routers have a number of NICs, one for each of the connections that it supports. Further, a host being at the edge of the network, is both the producer and consumer of messages. Therefore, the protocol stack on the host contains all the five layers that we discussed in Section 13.4.1. On the other hand, the router contains only the bottom three layers of the protocol stack due to its intended functionality as a network level packet router.

Let us dig a little deeper into addressing on the Internet to understand how hosts get network addresses as well as the number of network addresses that each router handles. You have all heard of IP addresses. As we will see shortly, it is a fairly logical way of thinking about addressing any networked device that can be reached on the Internet. Consider phone numbers associated with landlines in the U.S. for a minute. Typically, all the landlines in a given city have the same area code (the top three digits of the phone number, e.g., Atlanta has area codes 404, 678, or 770 assigned to it). The next three digits may designate a particular exchange designated for a geographical area or an entity (e.g., Georgia Tech has the exchanges numbered 894 and 385 designated for all campus numbers). The bottom four digits are associated with the specific connection given to an end device.


**Figure 13.19: U.S. Phone number**

Thus, the phone number is a multi-part address. So are addresses on the Internet. However as we will see shortly, the IP addresses do not have a geographical meaning as do the phone numbers. The multipart nature of IP addresses is essentially a mechanism to support hierarchical addressing of networks that comprise the Internet. IP addresses (in IPv4[11]) are 32-bits. Every interface connected to the Internet has to have a globally unique IP address. Thus, if you have a laptop with a wireless card and a wired Internet connection, each of them will have a separate IP address. Similarly, a router that provides connectivity between several independent network segments will have a unique

[10] The interested reader is referred to, "BGP4: Interdomain routing in the Internet," by J. Stewart, Addison-Wesley, 1999.

[11] This discussion pertains to IPv4. IPv6 uses 64-bit addressing and has been proposed to overcome the 32-bit addressing limitation of IPv4. But due to the widespread adoption of IPv4, it is going to be some time before IPv6 can take its place, if at all.

IP address for each of its network interfaces. The 32-bit IP address is a 4-part address usually expressed in the *dotted decimal* notation: *p.q.r.s*, where each of p, q, r, and s, is an 8-bit quantity. Consider the IP address **128.61.23.216**. Each part is the decimal equivalent of the 8-bit pattern that represents that part. The 32-bit binary bit pattern for this IP address is

$$(10000000\ 00111101\ 00010111\ 11011000)_2$$
$$(\ 128\ \ 61\ \ 23\ \ 216\ )_{10}$$

This structure of the 32-bit IP address is a key feature that is used in Internet routing. Some number of the most significant bits of the IP-address constitutes what is termed as the "IP Network." For example, the top 24-bits of the address may be used to name a specific *IP network*, and the bottom 8-bits to uniquely identify the specific device connected to the network. It is customary to denote an IP network with the notation *x.y.z.0/n*, where *n* is the number of bits reserved for the network part of the IP address. In this case, the IP network is 128.61.23.0/24, since the top 24-bits of the IP address constitute the network part. Figure 13.20 shows a number of hosts on a LAN that are connected by a router to the Internet. How many IP networks are there in this figure?

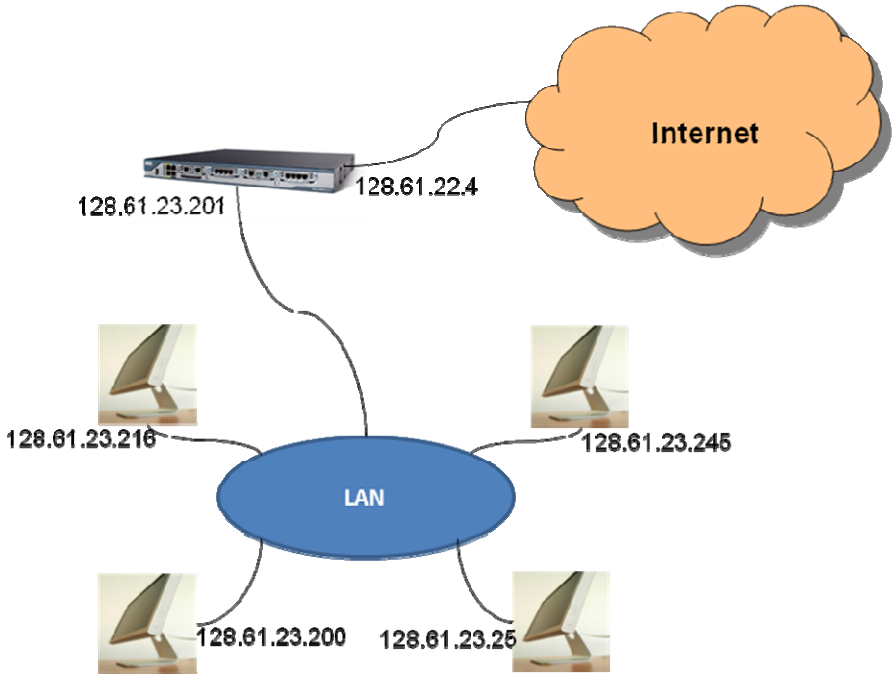



**Figure 13.20: IP Networks**

The answer is **2**. The hosts on the LAN and the interface in the router that also connects to the LAN are all part of the same IP network whose address is 128.61.23.0/24. On the other side, the router is connected through another interface that has an address 128.61.22.4 to the Internet, and is on a different IP network, namely, 128.61.22.0/24.

**Example 6:**
Consider Figure 13.21. How many IP networks are in this Figure? Assume that the top 24 bits of the 32-bit address name an IP network.

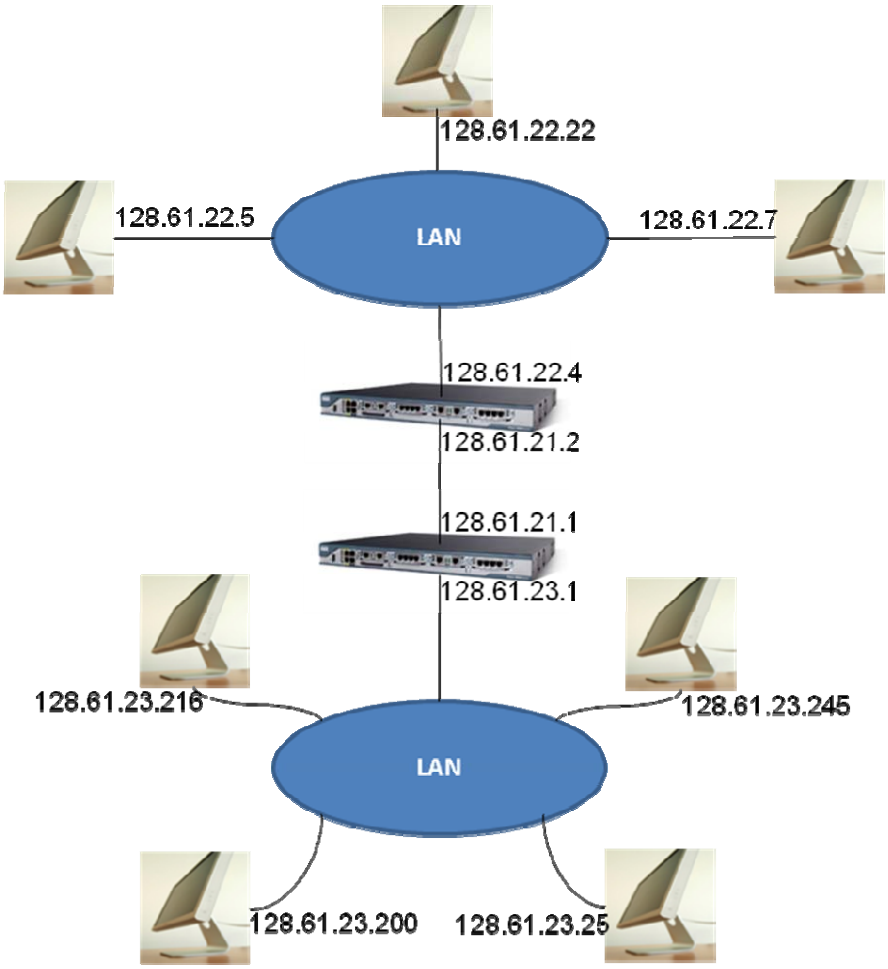



**Figure 13.21: Multiple IP Network Segments**

**Answer:**
There are **three** IP networks in this figure. One for the lower half of the figure connecting the 4 hosts on the lower LAN with the bottom router (network address: 128.61.23.0/24), second connecting the two routers together (network address: 128.61.21.0/24), and the third connecting the top 3 hosts on the upper LAN with the top router (network address: 128.61.22.0/24).

The Internet is composed of millions of such IP network segments. It is important to understand that it is not necessary that the network part of the address always be 24-bits. Let us say you are starting a company and need Internet connectivity to hook up 1000 computers to the Internet. You will request and get from an Internet Service Provider (ISP) a range of IP addresses for your organization that will have the top 22 bits fixed.

The bottom 10-bits allow you to connect up to 1024 ($2^{10}$) machines to the Internet. Such a network will have the dotted decimal form *x.y.z.0/22* indicating that the network part of the address is 22 bits. All the hosts in the organization will have the same top 22-bits for an IP address, and differ only in the bottom 10-bits. The network administrator may further subdivide the 10-bit bottom part of the address to create subnets, similar to the above example.

### 13.7.3 Network Service Model

As should be evident by now, in a large network, packets flow through several intermediate hops from the source before reaching its ultimate destination. For example, in Figure 13.21, consider a packet from the host on the bottom left (with interface address: 128.61.23.200) to the host at the top (with interface addresss: 128.61.22.22). This packet uses 3 network hops (128.61.23.0/24, 128.61.21.0/24, and 128.61.22.0/24) before reaching the destination.

#### 13.7.3.1 Circuit Switching

How should the network facilitate packet delivery among end hosts? This is the question answered by the network service model. Even before we discuss the network service model, we should understand some terminologies that are fundamental to networking. Let us start with telephone networks. We have come a long way from the days of Alexander Graham Bell (who is credited with the invention of the telephone). No longer is there a single physical wire between the two endpoints of a call. Instead, there is a whole bunch of switches between the two endpoints when a call is in progress. While the technology has changed dramatically from the early days of telephony, the principle is still the same: *logically* a *dedicated* circuit is established between the two endpoints for the duration of the telephone call. This is what is referred to as *circuit switching*, and to this day is the dominant technique used in telephony. Imagine Vasanthi is going to Mailpatti, TamilNadu, India from Atlanta, GA, USA to visit her grandmother. Being the paranoid person that she is, she makes reservations on every leg of the long journey, right from the airport shuttle bus in Atlanta, to the last bullock cart ride to her grandmother's house. If she does not show up for any of the legs of the long journey, then the seat goes unused on that leg of the trip. This is exactly the situation with circuit switching.

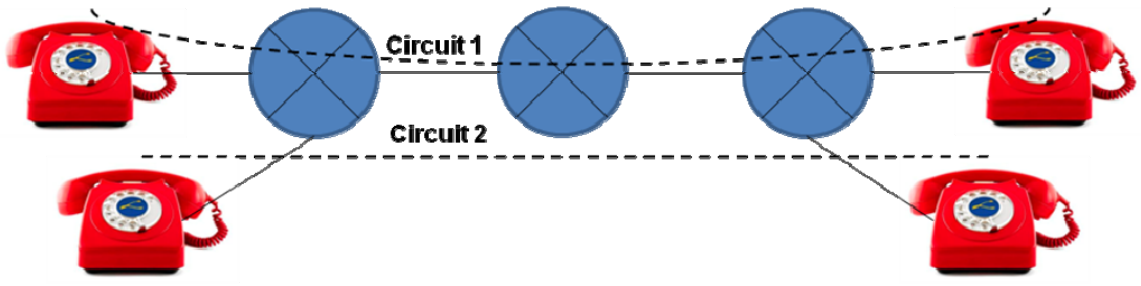



**Figure 13.22: Circuit switching**

There are a number of switches between the two end points of the telephone call. There are physical links connecting the switches. Figure 13.22 shows two distinct circuits (dashed lines) co-existing over the same set of physical links (solid lines) and switches. Network resources (in terms of bandwidth on these physical links) are reserved once the

call is established for the duration of the call. Thus circuit switching gives guaranteed quality of service once the call is established at the risk of under-utilizing the network resources (for e.g., *silent time* when no one is talking during a telephone call). The physical link that carries your telephone conversation between the switches may support several simultaneous *channels* or *circuits*. There are techniques such as *Frequency Division Multiplexing (FDM)* and *Time Division Multiplexing (TDM)* that allow sharing of the physical links for multiple concurrent connections. Given the total bandwidth available on a given link, these techniques allow creation of dedicated channels for supporting individual conversations. Naturally, if the maximum limit is reached, no new conversations can be entertained until some of the existing ones complete. This is the reason why, sometimes, you may hear the recording that says, "all circuits are busy, try your call later." Detailed discussion of these techniques is beyond the scope of this textbook, but suffice it to say that these techniques are analogous to having multiple lanes on a highway.

**13.7.3.2 Packet Switching**

An alternative to circuit switching is *packet switching*. Imagine Vasanthi gets adventurous on her second trip to her grandmother's village. Instead of reserving a seat on every leg of her journey, she just shows up at each leg of the journey. The ticketing agent determines the best choice for her next leg of the journey based on the current circumstance. It is possible that she will have to wait if there is no vacancy on any of the choices for her next leg. The basic idea of packet switching is exactly similar to this situation. The idea is not to reserve bandwidth on any of the physical links. When a packet arrives at a switch, the switch examines the destination for the packet, and sends it along on the appropriate outgoing link towards its destination. Figure 13.23 shows a conceptual picture of a packet switch (routers we introduced at the beginning of Section 13.7 is an example of a packet switch). It has input buffers and output buffers. Shortly, we will see the purpose served by these buffers.

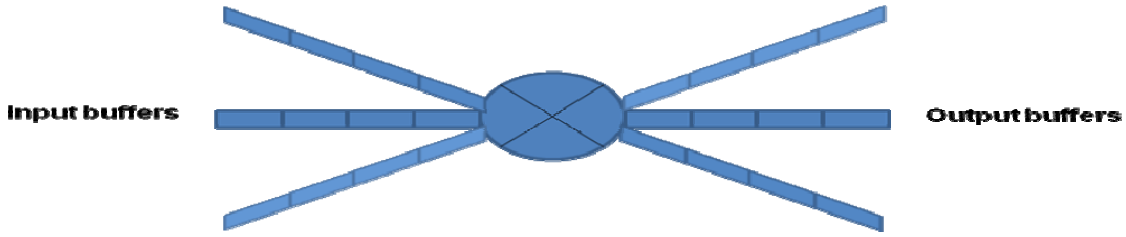



**Figure 13.23: Conceptual Picture of a Packet Switch**

Packet switched networks are referred to as *store and forward* networks. The switch cannot start sending the packet on an outgoing link until the entire packet has arrived at the switch. This is referred to as the *store and forward delay* in packet switched networks. Recall that a switch may have a number of physical links. Associated with each incoming link is an input buffer; the same is true of outgoing links. The input buffers are useful for receiving the bits of a packet as they arrive. Once the packet has arrived in its entirety, then it is ready to be sent on an outgoing link. The packet switch examines the destination address and based on the routing information contained in the

routing table, places the packet on the appropriate outgoing link. However, the outgoing link may be busy sending a previous packet. In this case, the switch will simply place the packet on the output buffer associated with that link. Thus, there may be some delay before the packet is actually sent on the outgoing link. This is referred to as the *queuing delay* in packet switched networks. As you can imagine, this delay is variable depending on network congestion. Further, since the amount of buffer space available on incoming and outgoing links are fixed, it is conceivable that the buffers are full (either on the input side or the output side) when a new packet arrives. In this case, the switch may have to simply drop a packet (either, one of the already queued ones, or the incoming packet depending on the network service model). This is the reason we mentioned *packet loss* in such networks right at the outset (see Section 13.3). Figure 13.24 shows the flow of packets of a given message in a packet-switched network. The figure should be reminiscent of the pipelined instruction execution the reader should be already familiar with from Chapter 5. Once the packets of the message have started filling up the pipeline, each switch is concurrently working on transmitting different packets of a given message.

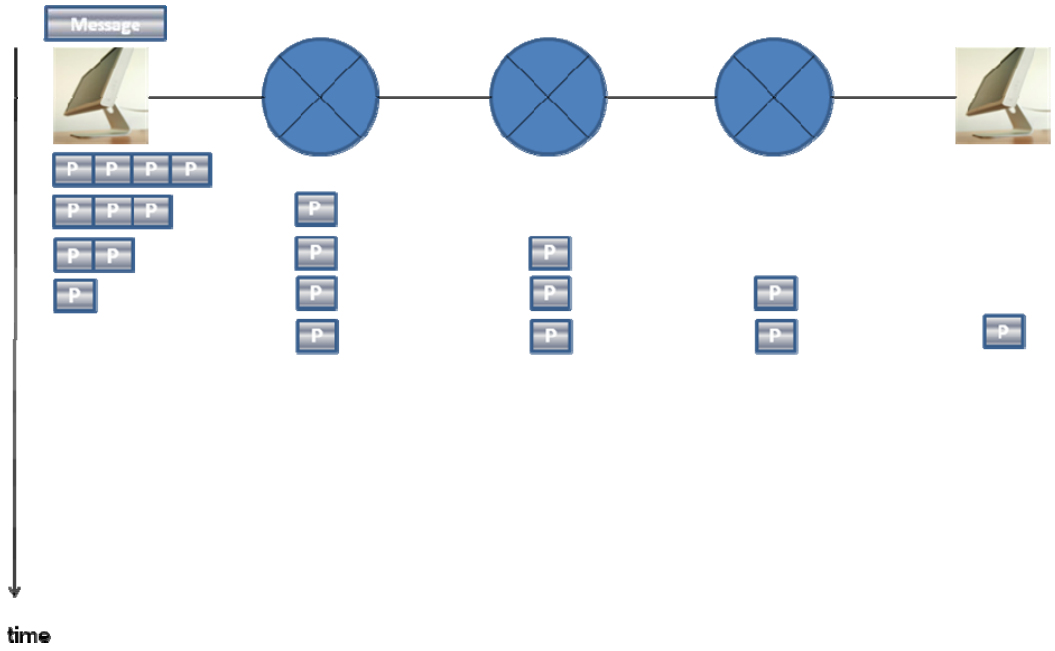



**Figure 13.24: A Packet-switched Network**

### 13.7.3.3 Message Switching

There is an assumption with packet switched networks that higher layers of the protocol stack (such as transport, see Section 13.5) deal with scatter/gather of message into packets, and out of order arrival of packets. It is conceivable that this functionality could be pushed into the network itself. Such a nework is called a *message switching* network. In this case, a switch stores and forwards at the granularity of an *entire message* as opposed to individual packets. This is illustrated in Figure 13.25. As should be evident, a packet switched network leads to better latency for individual messages (by pipelining the packets of a given message, compare Figures 13.23 and 13.24). Further, if there are bit-errors during transmission (which can occur due to a variety of electrical and

electromechanical reasons in the physical links carrying the data), the impact is confined to individual packets instead of the whole message. As the reader would have inferred already, packet switching will incur a higher header overhead compared to message switching, since packets have to be individually addressed, routed, and checked for integrity. It should be noted that message-switched networks also take advantage of pipelined communication; it is just that the pipelining is at the level of messages not packets within a single message.

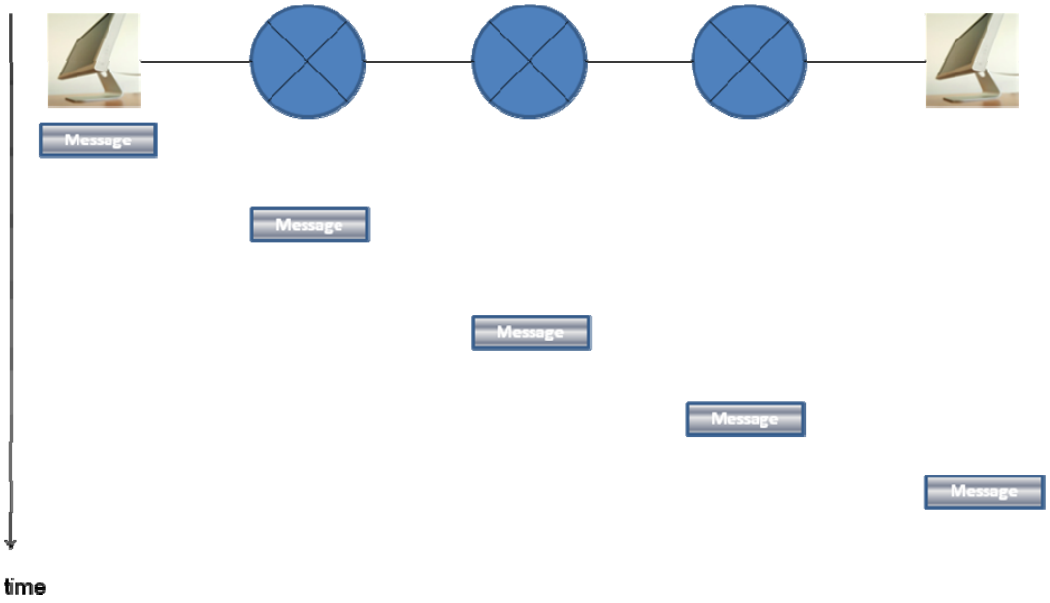



**Figure 13.25: A Message-switched Network**

### 13.7.3.4 Service Model with Packet Switched Networks

Packet switching offers several advantages over the competition, especially for computer networks:

- Since there is no advance reservation as opposed to circuit switching (using either FDM or TDM), the available bandwidth on a physical link can be fully utilized by a packet switched network. This is why even telephone networks are starting to adopt packet switching for expensive overseas links.
- The virtue of circuit switching is guaranteed transmission time between the two end points due to the reservation, which is important especially for voice traffic. However, this is less important for data traffic, which dominates the Internet today. With advances in network technology, even new applications (such as Voice over IP – VoIP that provides phone service via the Internet) that may need guaranteed quality of service, work fine over a packet switched network.
- Packet switched networks are simpler to design and operate compared to circuit switching.
- As we observed earlier, message switching may not use the network as efficiently as packet switching due to the coarseness of messages as opposed to packets.

Further, dealing with bit errors during transmissions at the packet level leads to better overall efficiency of the network.

Due to all these reasons cited thus far, Internet uses packet switching in the network layer and IP is the ubiquitous network layer protocol in the Internet.

With packet-switched networks,[12] there are two network service models: *datagram* and *virtual circuit*. Datagram service model is akin to postal service. Every packet is individually stamped with the destination address, and the routers *en route* look at the address to decide a plausible route to reach the destination. The routers use the routing tables to make this decision, and the routing algorithms that we discussed earlier are used to set up these routing tables and constantly update them as well. Internet by and large supports the datagram service model.

Virtual circuit is akin to making a telephone call. We understand circuit switching in which real physical resources (link bandwidth) are allocated in a dedicated fashion to each independent conversation. Virtual circuit is similar, except that physical resources are not reserved. The idea is to establish a route from source to destination during the *call set up* phase (called a *virtual circuit*). The source gets a *virtual circuit number*, which it can use for the duration of the call to send packets along the virtual circuit. The switches *en route* maintain a table (let us call it VC table) that contains information pertaining to each virtual circuit currently managed by the switch, namely, *incoming link, outgoing link*. Thus, the routing decision to be taken by the switch is very simple: examine the VC number in the incoming packet, consult the VC table, and place the packet in the output buffer of the corresponding outgoing link. Finally, once the message transmission is complete, the network layer at the source executes a *call teardown* that deletes the corresponding entry in each of the switches *en route* to the destination. This discussion of a virtual circuit is intentionally simplified so as not to confuse the reader. In reality, during the connection establishment phase, each switch is free to locally choose a VC number for the new connection (simplifies network management). So the VC table become a little bit more elaborate: VC number of incoming packet (chosen by the previous switch in the route), associated incoming link, VC number of outgoing packet (chosen by this switch), associated outgoing link. Examples of network layer protocols that support virtual circuits include ATM and X.25. Internet IP protocol supports only the datagram service model.

### 13.7.4 Network Routing Vs. Forwarding

It is important to distinguish between *routing* and *forwarding*. A simple analogy would help here. Consider driving to work every day. Routing is akin to the decision of which sequence of streets and highways to take from home to work, while forwarding is akin to the actual act of driving in that chosen route. We do the former only occasionally but the latter every day[13].

[12] We will not distinguish between message and packet switching any longer, since message switching may be considered a special case of packet switching, where the packet is the entire message.
[13] Thanks to my colleague Constantine Dovrolis for this simple yet illuminating analogy.

Routing and forwarding are both network layer functionalities. These are in addition to the service model provided by this layer. However, we don't want the reader to get the impression that the network layer has to compute a route every time it sees a packet from the transport layer. As we mentioned already (see Section 13.7), when the transport layer gives it a packet, the network layer determines the next hop the packet must take in its journey towards the ultimate destination. This is the *forwarding* function of the network layer. To aid in this function, the network layer has a *forwarding table* that contains the next hop IP address given a destination IP address. The forwarding table resides in the network layer of the Internet protocol stack.

The above discussion begs the question, where the routing table is and how the routes are computed. In Chapter 8, we discussed daemon processes in the operating systems that carry out certain needed functionalities in the background. For example, a paging daemon runs the page replacement algorithms to ensure that there is a pool of free page frames available for the virtual memory system upon encountering a page fault. Basically, the intent is to make sure that such book-keeping activities of the operating system are not in the critical path of program execution.

Computing network routes is yet another background activity of the operating system. In Unix operating system you may see a daemon process named *routed* (route daemon). This daemon process runs periodically in the background, computes the routes using algorithms similar to the ones discussed in this section, and creates *routing tables*. In other words, route calculation is not in the critical path of data transport through the network. The routing tables are disseminated to other nodes in the Internet to update the peer routing tables as network properties change. The routing daemon uses the newly discovered routes to update the forwarding table in the networking layer of the protocol stack.

### 13.7.5 Network Layer Summary

Table 13.5 summarizes all the key terminologies we have discussed thus far with respect to the network layer functionalities.

| Network Terminology | Definition/Use |
|---|---|
| **Circuit switching** | A network layer technology used in telephony. Reserves the network resources (link bandwidth in all the links from source to destination) for the duration of the call; no queuing or store-and-forward delays |
| **TDM** | Time division multiplexing, a technique for supporting multiple channels on a physical link used in telephony |
| **FDM** | Frequency division multiplexing, also a technique for supporting multiple channels on a physical link used in telephony |
| **Packet switching** | A network layer technology used in wide area Internet. It supports best effort delivery of packets from source to destination without reserving any network resources en route. |

| | |
|---|---|
| **Message switching** | Similar to packet switching but at the granularity of the whole message (at the transport level) instead of packets. |
| **Switch/Router** | A device that supports the network layer functionality. It may simply be a computer with a number of network interfaces and adequate memory to serve as input and output buffers. |
| **Input buffers** | These are buffers associated with each input link to a switch for assembling incoming packets. |
| **Output buffers** | These are buffers associated with each outgoing link from a switch if in case the link is busy. |
| **Routing table** | This is table that gives the next hop to be used by this switch for an incoming packet based on the destination address. The initial contents of the table as well as periodic updates are a result of routing algorithms in use by the network layer. |
| **Delays** | The delays experienced by packets in a packet-switched network |
| **Store and forward** | This delay is due to the waiting time for the packet to be fully formed in the input buffer before the switch can act on it. |
| **Queuing** | This delay accounts for the waiting time experienced by a packet on either the input or the output buffer before it is finally sent out on an outgoing link. |
| **Packet loss** | This is due to the switch having to drop a packet due to either the input or the output buffer being full and is indicative of traffic congestion on specific routes of the network. |
| **Service Model** | This is the contract between the network layer and the upper layers of the protocol stack. Both the datagram and virtual circuit models used in packet-switched networks provide best effort delivery of packets. |
| **Virtual Circuit (VC)** | This model sets up a virtual circuit between the source and destination so that individual packets may simply use this number instead of the destination address. This also helps to simplify the routing decision a switch has to make on an incoming packet. |
| **Datagram** | This model does not need any call setup or tear down. Each packet is independent of the others and the switch provides a best effort service model to deliver it to the ultimate destination using information in its routing table. |

**Table 13.5: A summary of key networking terminologies**

To conclude the discussion on the network layer, let us comment on its relationship to the transport layer. It is interesting to note that the intricacies involved with the network layer are completely hidden from the transport layer. This is the power of abstractions. The transport layer may itself be connection-less (such as UDP) or connection-oriented (such as TCP), independent of the network service model. However, in general, if the network layer supports virtual circuits, the transport layer will also be connection-oriented.

### 13.8 Link Layer and Local Area Networks

Now that we have covered the two layers of the protocol stack that has implications on the operating system, let us turn our attention to the link layer which has implications on the hardware. While the top-down approach that we have taken to present the protocol stack makes perfect sense, truth be told that it is innovation at the link layer dating back to the early days of networking that has made Internet a household name.

Essentially, the link layer is responsible for acquiring the physical medium for transmission, and sending the packet over the physical medium to the destination host. Just as a point of distinction, link layer deals with *frames* as opposed to packets. Depending on the details of the link layer technology, a network layer packet (such as that generated by the IP protocol) may require several frames at the link layer level to transmit to the destination.

Depending on the access mechanism used to gain control of the physical medium, link layer protocols may be grouped into two broad categories[14]: *random access* and *taking turns*. Ethernet is an example of the former, while token ring is an example of the latter. Today, Ethernet is the most popular link layer technology that forms the basis for connecting the end devices (hosts) to the local area network first, and from there to the wide area Internet.

The part of the link layer protocol that deals with gaining access to the physical medium is usually called *Media Access and Control* (*MAC* for short) layer.

#### 13.8.1 Ethernet

Electrically, Ethernet is a cable that connects computers together as shown in Figure 13.26. We use the term *node* to mean a computer or a host connected to a network.

[14] We borrow these terms from the textbook by Kurose and Ross, "Computer Networking: A top down approach featuring the Internet," Addison-Wesley.

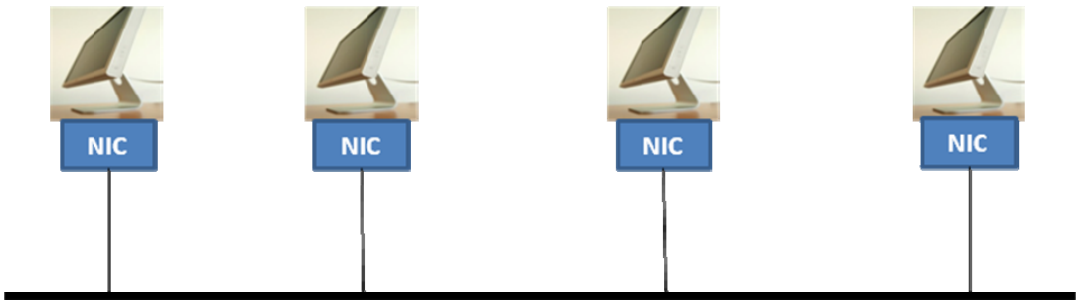



**Figure 13.26: Computers networked via Ethernet**

Let us understand the Ethernet protocol. Ethernet cable is akin to a bus[15]. However, this bus is not confined to a box as we have seen in earlier chapters, but perhaps runs through an entire building. In Chapter 10, we mentioned *bus arbitration*, a scheme for deciding who gets the bus among the competing components connected to it. In a bus, that connects the components within a box (processors, memory, I/O devices), *arbitration logic* decides who has control of the bus when multiple units require simultaneous access to the bus. Arbitration logic is the actual hardware that implements the bus arbitration scheme. Since there are a bounded and fixed number of units within a box, it is feasible to design such arbitration logic. Further, the signals travel a short distance (order of a few feet at most) within a box. On the other hand, Ethernet connects an *arbitrary* number of units together in an office setting over *several hundreds of meters*. Thus, the designers of Ethernet had to think of some other way of arbitrating among units competing to use the medium simultaneously to deal with the twin problems of large distances and arbitrary number of units.

### 13.8.2 CSMA/CD

We will describe a random access communication protocol called *carrier sense multiple access/collision detect (CSMA/CD)* that forms the basis for arbitration of a broadcast medium such as the Ethernet. We keep the discussion of the protocol simple and superficial without getting into too much detail on the communication theory behind the data transmission. The designers of Ethernet adopted this protocol to overcome the twin problems of distance and arbitrary number of devices being connected to the Ethernet cable. The basic intuition behind CSMA/CD comes from the way we would carry on a polite conversation among colleagues sitting around a conference table. We would look around to make sure no one is talking; then we will start to talk; if two or more people try to say something at the same time we will shut up and try again when no one else is talking.

CSMA/CD protocol is not much different from this human analogy. Figure 13.27 shows the state transitions of a computer sending a frame using CSMA/CD.

[15] As we will see later (Section 13.9), modern Ethernet using switches is point-to-point and switched. This view of Ethernet as a bus is useful to understand the nuances of the Ethernet protocol.

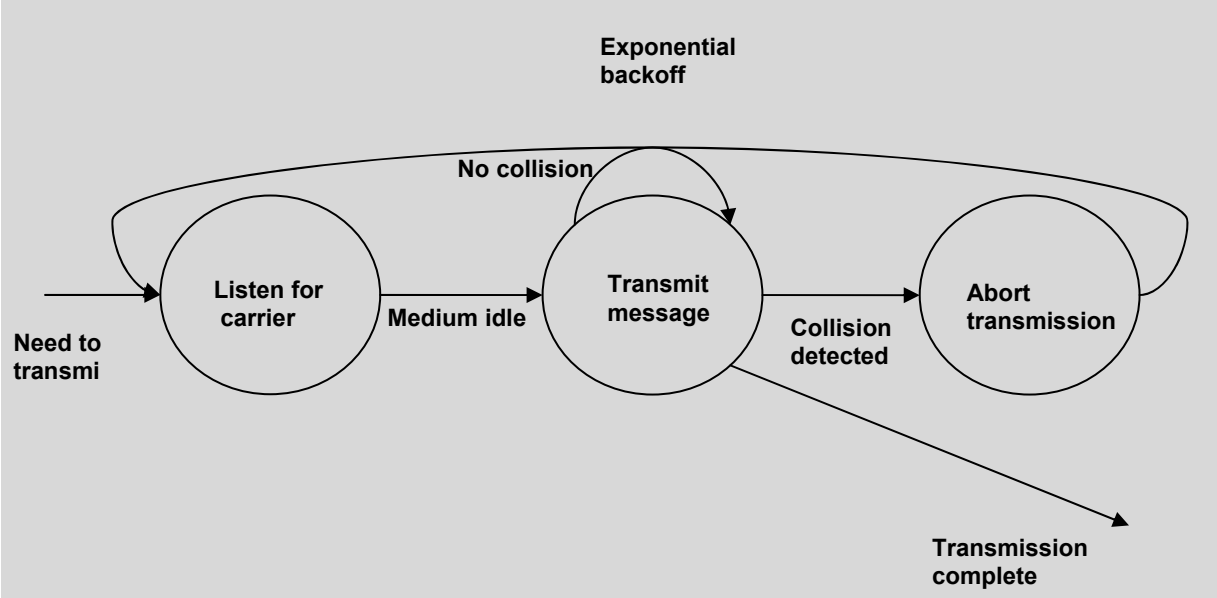



**Figure 13.27: State transitions in CSMA/CD**

Let us dig a bit deeper into the protocol name and basic idea behind the CSMA/CD protocol.

1. A *station* (i.e., a computer) wanting to transmit *senses* the *medium* (i.e., the cable) whether there is an ongoing frame transmission. If yes, then the station waits until the medium is *idle* and starts its transmission. If the medium is idle, it can start its frame transmission immediately. Absence of any electrical activity on the medium is the indication of idleness.
2. Multiple stations sense the cable simultaneously for idleness. The *multiple access* term in the protocol comes from this fact. Thus, simultaneously multiple units may come to the same conclusion that the medium is idle, and start their respective frame transmissions. This is a problem. We will see shortly how the protocol deals with this problem.
3. Each station, after starting a frame transmission, listens for a collision. Each station knows the electrical activity it is causing on the medium. If what it observes (via listening) is different from this activity, then it knows that some other station has also assumed idleness of the medium. We refer to this as *collision detection* in the protocol. The transmitting station immediately aborts the transmission, and sends out a *noise burst* (think of this as similar to static noise you may hear on your radio sometimes, or continuing our conversation analogy saying "I'm sorry" when multiple people start to talk at the same time) to warn all other stations that a collision has occurred. The station then waits for a *random* amount of time before repeating the cycle of *sense, transmit, and observe for collision*, until it successfully completes its transmission. The algorithm chooses the random number for determining the amount of time a station waits before attempting to retransmit from a sequence that grows exponentially with the number of collisions experienced during a transmission. Hence, this phase of the algorithm is commonly referred to as *exponential backoff*.

We define *collision domain* as the set of computers that can hear the transmissions of one another[16].

Let us understand how exactly the frame is transmitted and how idleness of the medium is detected. The protocol uses *base band signaling*, i.e., the frame is transmitted *digitally* on the medium (i.e., the cable) directly as 0's and 1's.

Just as a point of distinction, *broadband* refers to simultaneous *analog* transmission of multiple messages on the same medium. Different services use different frequencies, respectively, to send their message content simultaneously. For example, a cable service that you may have in your home is an example of broadband. The same piece of wire *simultaneously* brings in your television signal, perhaps your phone service, and perhaps your internet connection.

### 13.8.3 IEEE 802.3

Ethernet uses a particular version of the CSMA/CD protocol standardized by IEEE, namely, IEEE 802.3. In this version of the CSMA/CD protocol, digital frame transmission uses *Manchester code* (see Figure 13.28), a particular type of bit encoding. This code represents logic 0 as a *low followed by a high transition*, and logic 1 as a *high followed by a low transition*. In the IEEE 802.3 standard, low signal is -0.85 volts and high signal is +0.85 volts and idle is 0 volts. Each bit transmission occupies a fixed amount of time, and the Manchester coding technique ensures that there is a voltage transition in the middle of each bit transmission thus enabling synchronization of the sender and receiver stations. Thus, there is always electrical activity when there is a frame transmission on the wire. If there is **no voltage transition** in the duration of a bit transmission then a station assumes the medium is *idle*. Since Ethernet uses *base band* signaling technique, there can only be one frame at a time on the medium.

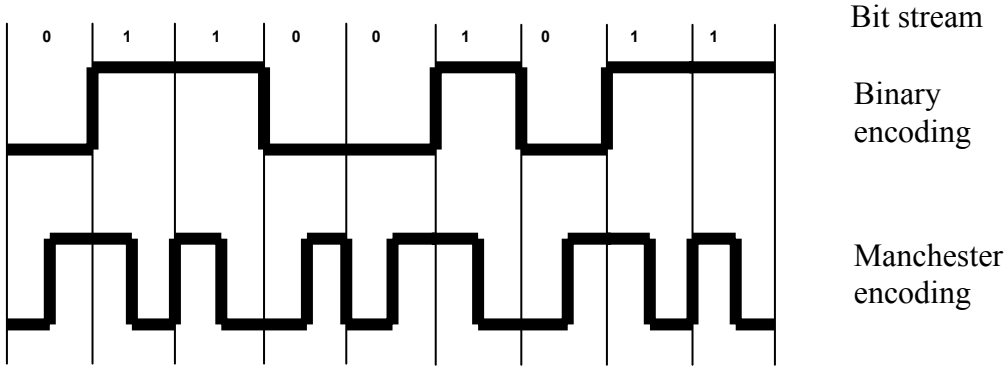


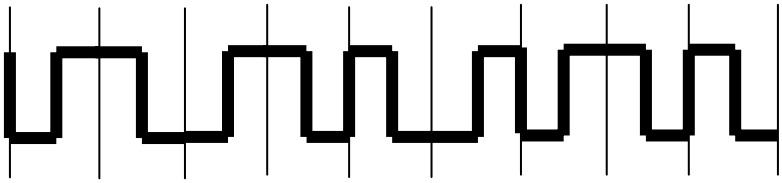

**Figure 13.28: Manchester Encoding**

[16] With the view of Ethernet as a bus, every host connected to the Ethernet is part of the collision domain. However, we will see later on in Section 13.9 that with modern networking gear, collision domain is limited to the set of hosts connected to the same hub.

It should be mentioned that while early Ethernet used CSMA/CD, most Ethernet LANs in use today are *switched Ethernets* (see Section 13.9) wherein there is *no* collision. It is also interesting to note that 10-Gigabit Ethernet does not even support CSMA/CD since it assumes that the hosts connect via switched links to the network. However, Ethernet as a link layer protocol continues to use the same wire format (i.e., frame headers) for compatibility.

---

**Example 7:**
What is the bit stream represented by the following Manchester encoding of the stream?



**Answer:**
**0 1 0 1 1 0 0 1 1**

---

### 13.8.4 Wireless LAN and IEEE 802.11

Wireless networks bring a number of new challenges compared to wired networks. Just as a point of distinction, we should mention a variation of CSMA protocol that finds use in wireless LANs, namely, *CSMA/CA*. The CA stands for *collision avoidance*. This variant of CSMA is used in situations where the stations cannot determine if there was a collision on the medium. There are two reasons why detecting collisions may be problematic. The first reason is simply a matter of implementation efficiency. Detecting collisions assumes that a station can *simultaneously* send its own transmission and receive to verify if its transmissions is being interfered with by transmissions from other stations. With wired medium implementing such a detection capability in the network interface is economically feasible while it is not the case with wireless medium.

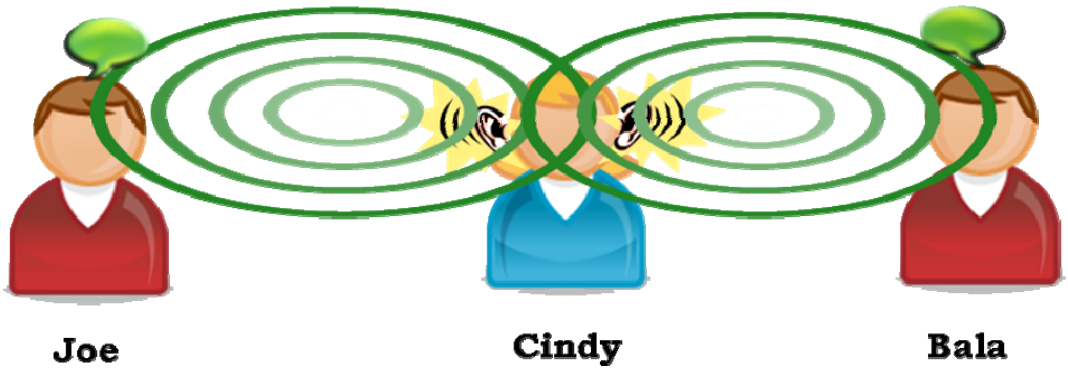



**Figure 13.29: Hidden Terminal Problem**

The second and more interesting reason is the *hidden terminal* problem. Imagine you and two of your buddies are standing along a long corridor (Joe, Cindy, and Bala in Figure

13.29). Cindy can hear Joe and Bala; Joe can hear Cindy; Bala can hear Cindy; however, Joe and Bala cannot hear each other. Thus, Joe may try to talk to Cindy simultaneous with Bala talking to Cindy. There will be a collision at Cindy but neither Bala nor Joe will realize this. This is the hidden terminal problem: Joe is hidden as far as Bala is concerned and vice versa.

One way of avoiding collisions is for the source to explicitly get permission to send by sending a short *Request To Send* (RTS) control packet on the medium to the desired destination. The destination (assuming that this RTS control packet successfully reaches the destination without any interference) responds with a *Clear To Send* (CTS) control packet. Upon receiving CTS, the source then sends the frame to the destination. Of course, the RTS packets from different nodes may collide. Fortunately, these are short packets and hence the damage is not huge and a node that wishes to transmit will be able to get the permission to transmit quickly. All the nodes in the LAN hear the RTS and/or the CTS control packets. Hence, they themselves will not try to send an RTS packet until the data transmission is complete thus ensuring that there will be no collisions.

The RTS-CTS handshake helps overcome the hidden terminal problem as well as avoid collisions.

The IEEE 802.11 RTS-CTS standard is a wireless LAN specification for the implementation of the CSMA/CA protocol using RTS/CTS.

### 13.8.5 Token Ring

As we mentioned at the beginning of this section, an alternative to the random access protocol such as the Ethernet, is for each station to take turns to transmit a frame. One simple idea for taking turns is *polling*. A master node polls each station in some pre-determined order to see if it needs to transmit. Upon getting the "go ahead" from the master, the slave station will transmit its message.

Rather than a centralized approach to taking turns, token ring offers a decentralized approach. The basic idea is to connect the nodes of the network in a ring formation as shown in Figure 13.30**.**

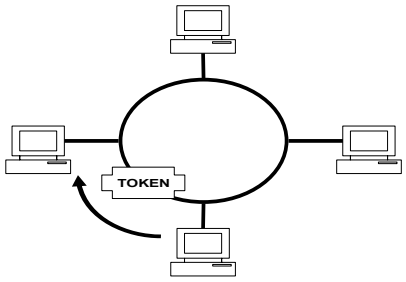



**Figure 13.30 Token Ring Network. Token passes continuously from host to host. Transmission occurs only when a host has the token.**

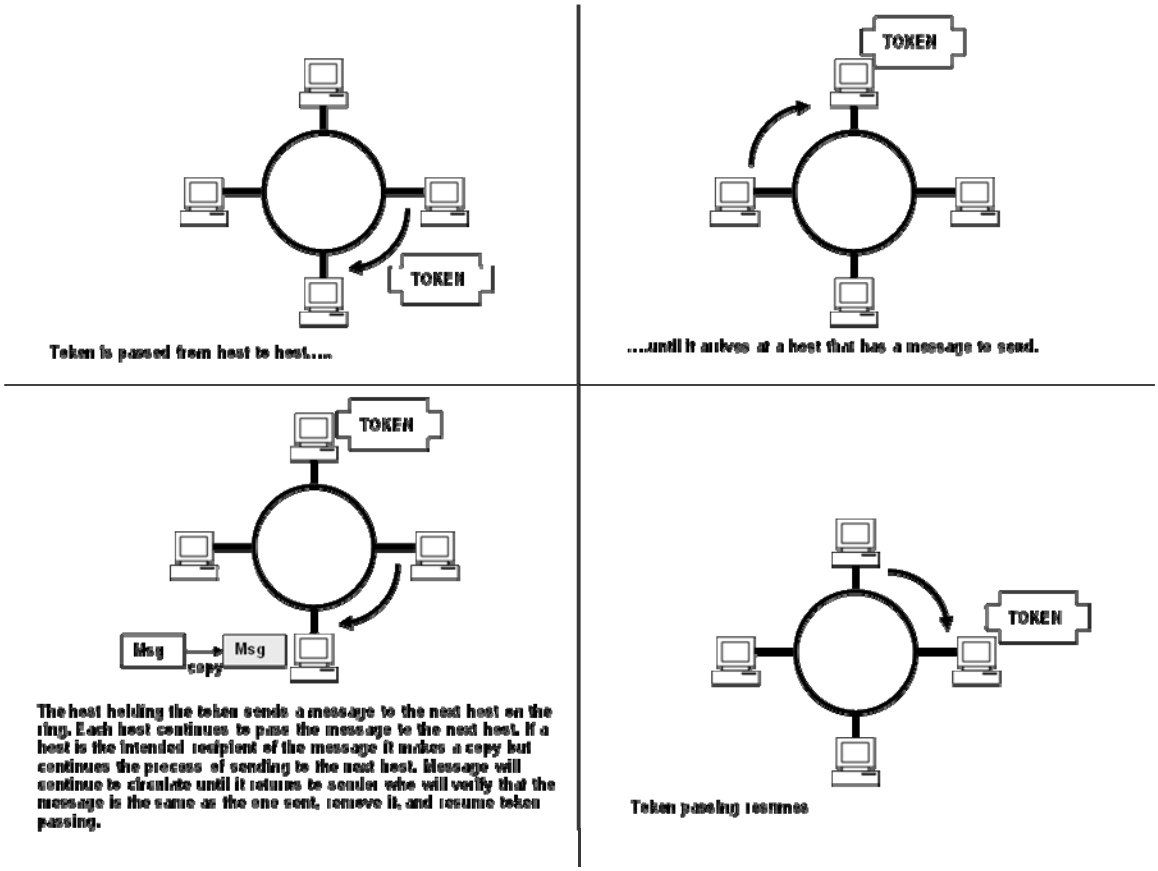



**Figure 13.31 Sequence of steps required to send a frame on the token ring**

A *token*, a special bit pattern, keeps circulating on the wire. A node that wishes to send a frame, waits for the token, grabs it, puts its frame on the wire, and then puts the token back on the wire. Each node examines the header of the frame. If the node determines that it is the destination for this frame then the node takes the frame. Somebody has to be responsible for removing the frame and regenerating the token. Usually, it is the sender of the frame that should remove the frame and put pack the token on the wire. In principle, the token ring also works like a broadcast medium, since the frame travels to all the nodes on the ring. However, if a token ring network spans a geographically large area, it would make sense for the destination node to remove the frame and put back the token on the wire.

Figure 13.31 shows the sequence of steps involved in frame transmission and reception in a token ring. By design, there are no collisions in the token ring. But this scheme has its own pitfalls. One disadvantage is the fact that a node has to wait for the token to send a frame. In a LAN with a large number of nodes, this could lead to considerable latency for frame transmission. The other disadvantage is the fact that if a node dies, then the LAN is broken. Similarly, if for some reason the token on the ring is lost or corrupted, then the LAN cannot function anymore. Of course, it is possible to work around all of these problems but the latency for frame transmission is a serious limitation with this

technology. Token ring has its advantages as well. Ethernet saturates under high traffic loads. The utilization never reaches 100% due to excessive collisions under such high traffic. Token ring works well under heavy load conditions. Table 13.6 gives a comparison of the two LAN protocols.

| Link Layer Protocol | Features | Pros | Cons |
|---|---|---|---|
| **Ethernet** | Member of random access protocol family; opportunistic broadcast using CSMA/CD; exponential backoff on collision | Simple to manage; works well in light load | Too many collisions under high load |
| **Token ring** | Member of taking turns protocol family; Token needed to transmit | Fair access to all competing stations; works well under heavy load | Unnecessary latency for token acquisition under light load |

**Table 13.6: A Comparison of Ethernet and Token Ring Protocols**

### 13.8.6 Other link layer protocols

We conclude this section with a bit of a perspective on LAN technologies. Both Ethernet and Token ring were viable link layer technologies in the late 1980's and early 1990's. However, for a variety of reasons, not all technical as usual, Ethernet has emerged as the clear winner of LAN technologies. While we have studied Ethernet and Token ring as examples of "random access" and "taking turns" protocols, it is worth mentioning that there are several other link layer protocols. Some of them showed tremendous promise when first introduced due to their improved performance over Ethernet. *FDDI* (which stands for *Fiber Distributed Data Interface*) and *ATM* (which stands for *Asynchronous Mode Transfer*) are two such examples. FDDI was originally conceived for fiber-optic physical medium, and was considered especially suitable as a high-bandwidth backbone network for connecting several islands of Ethernet-based LANs together in a large organization such a university campus or a big corporation. It is also a variant of the taking turns protocol similar to the token ring. ATM was appealing due to its ability to provide guaranteed quality of service through bandwidth reservation on the links and admission control to avoid network congestion. It is a connection-oriented link layer protocol and provides many of the network layer functionality as well that a transport layer protocol can be directly implemented on top of it. ATM has some presence in Metropolitan and Wide Area Networks (WAN) for use by telecommunication service providers. However, it has not found as much traction in LAN due to its relative complexity compared to Ethernet.

A link layer technology that has widespread deployment today is *Point to Point Protocol* (*PPP* for short). PPP is the link layer protocol used by dial-up connections and its widespread adoption is due to the large number of user community dependent on dial-up for access to the Internet.

Interestingly, every time there has been a threat of obsolescence, Ethernet has found a way to reinvent itself and prevail. The advent of Gigabit Ethernet stole the thunder away from FDDI for the most part. The advent of 10-Gigabit Ethernet which is now on the horizon (circa 2007) is likely to steal the thunder away from ATM for metropolitan and wide-area networks.

### 13.9 Networking Hardware

We do not get into the physical layer details such as the electrical, radio, or optical properties of the conduits used for data transport. The interested reader can get such information from other sources[17]. Instead, we review the networking gear that is common in everyday use these days from homes to offices for use in an Ethernet-based LAN.

**Repeater.** Electrical signals decay in signal strength with distance. Heat dissipation, due to the resistance and capacitance of the physical conductors carrying the signal, leads to the attenuation of the electrical signal. A repeater is a generic name for an electrical device that boosts the energy level of bits on an incoming connection and sends it out on an outgoing connection. Since LANs spread over fairly large geographical distances (e.g., a building or an entire campus), there is a need to boost the signal strength at regular intervals. Repeaters are commonly used in both LANs and Wide Area Networks (WANs) to overcome signal attenuation problems.

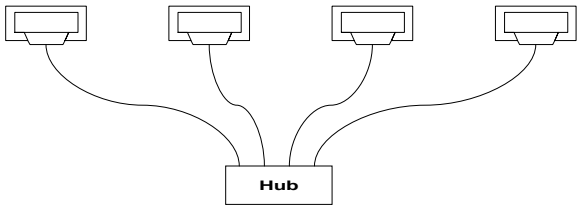



**Figure 13.32 Simple application of a 4-port hub to interconnect 4 computers**

**Hub.** In Section 13.8.1, we mentioned that Ethernet is logically a bus (see Figure 13.26). A hub is essentially Ethernet in a box: as can be seen in Figure 13.32, the Ethernet cable of Figure 13.26 is replaced by a hub. A hub relays the bits received from one host to others (computers and other hubs forming a logical bus). Hubs allow constructing a more complex LAN fairly easily as shown in Figure 13.33. All the hosts reachable via the logical bus formed by the interconnected hubs shown in Figure 13.33 are part of the same collision domain. Recall that the term collision domain refers to the set of computers that hear the transmissions of one another (see Section 13.8.1). Therefore, computers

[17] Kurose and Ross, "Computer Networking: A top down approach featuring the Internet," Addison-Wesley.

connected to a hub have to detect collisions and follow the exponential backoff phase of the Ethernet protocol to resolve collisions to complete their message transmissions. A hub is essentially a *multi-port* repeater, and often the terms are used interchangeably.

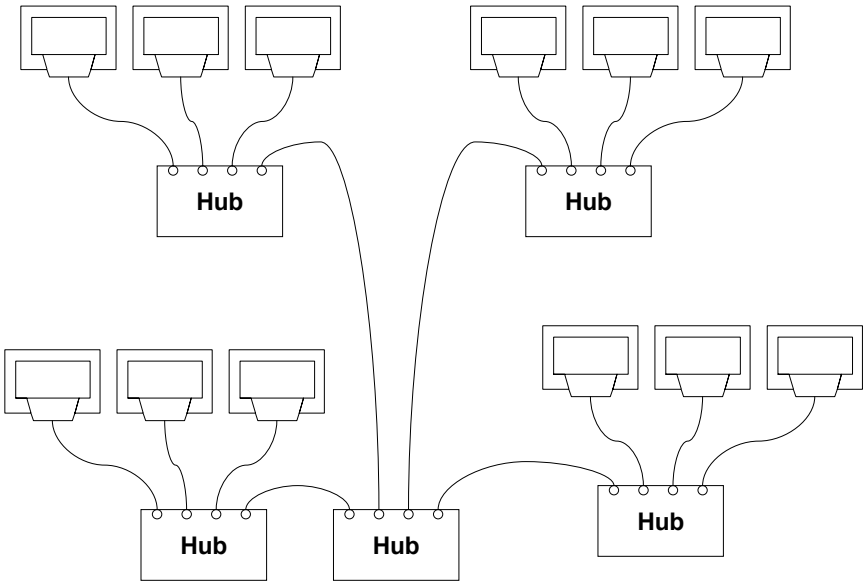



**Figure 13.33 Using hubs to construct a more complex network. All computers shown are part of the same collision domain**

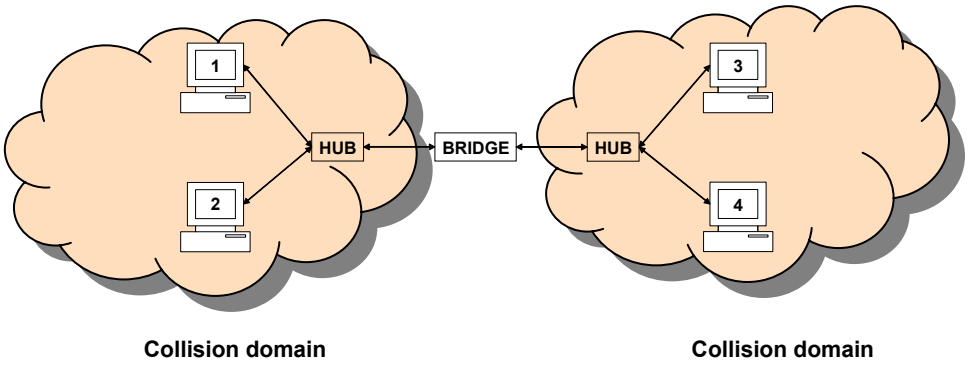



**Figure 13.34: Hosts 1 and 2 are on a separate collision domain from hosts 3 and 4**

**Bridges and Switches.** The late 90's saw yet another landmark in the evolution of local area networking gear, *switched Ethernet*. Bridges and switches separate collision domains from one another. For example, Figure 13.34 shows a bridge separating two collision domains from each other. Hosts 1 and 2 form one collision domain, while hosts 3 and 4

form the other. The data packets of host 1 do not cross the bridge unless the destination is one of hosts 3 or 4. This isolation of the traffic allows hosts 1 and 2 to communicate in parallel with hosts 3 and 4.

What happens when host 1 wants to send a message to host 3 simultaneous with host 4 wanting to send a message to host 2? The bridge recognizes the conflict and serializes the traffic flow between the two sides (for example 1 to 3 in one cycle; followed by 4 to 2 in the next network cycle). For this reason, a bridge contains sufficient buffering to hold the packets when such conflicts arise. The hosts never experience a collision unless they happen to be in the same collision domain. The end points of the bridge, referred to as *ports*, pass packets to the other ports on a need basis.

Though bridge and switch appear interchangeably in the literature, some authors define a bridge as connecting a limited number of collision domains (typically 2) as in Figure 13.34. Functionally a switch is a generalization of the bridge supporting any number of collision domains. Figure 13.35 shows a switch supporting 4 collision domains (each of A, B, C, and D represent an independent collision domain).

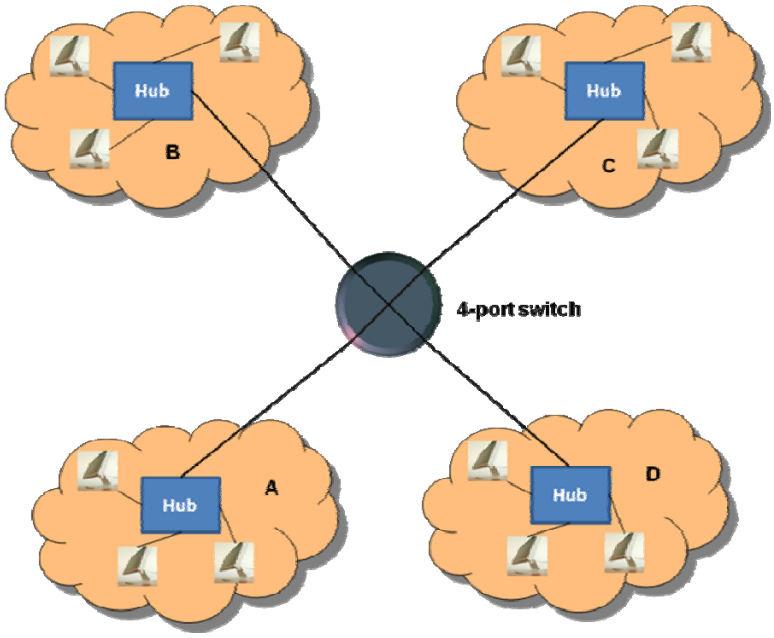



**Figure 13.35 Conceptual Diagram of a 4-port Switch. A host on a collision domain connected to A could communicate with a host on a collision domain connected to C simultaneous with communication between hosts on collision domains connected to B and D.**

In Figure 13.33, if we replace all the hubs by switches, we essentially get a *switched Ethernet* that has no collisions.

**VLAN.** Virtual LANs are a natural next step for switched Ethernet with switches as in Figure 13.36. Nodes 1 and 5 in Figure 13.36 may request to be on the same VLAN;

while nodes 2, 6, and 7 may request to be on the same VLAN. If node 2 sends a broadcast packet, then nodes 6 and 7 will receive it, and no other nodes will. Thus, with a hierarchy of switches, nodes that are on different geographical locations (and hence on distinct switches), still form one broadcast domain.

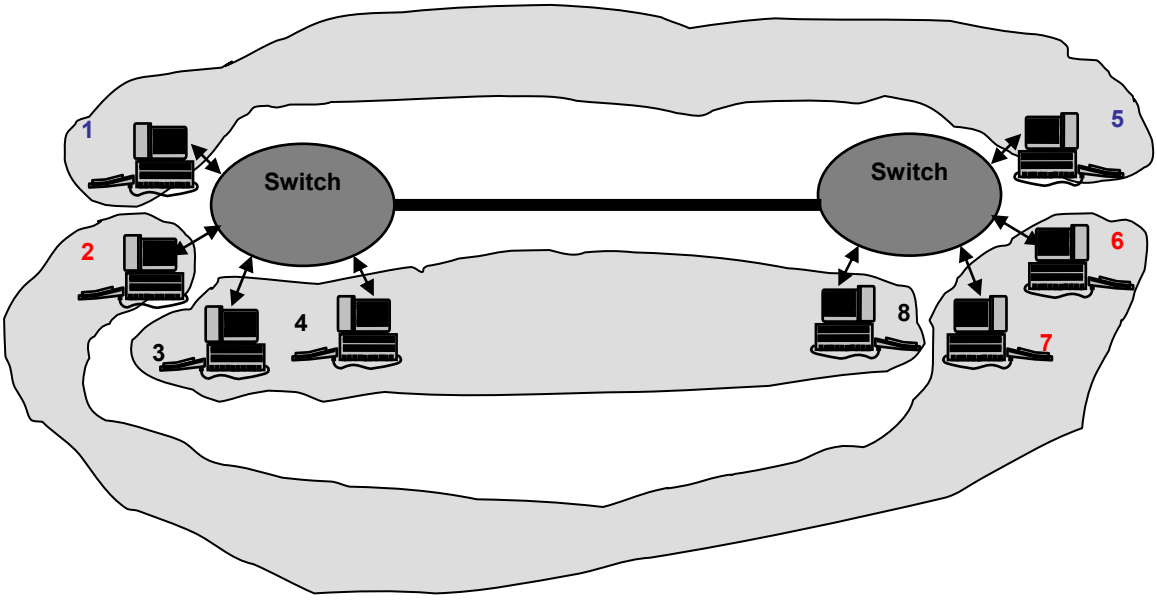



**Figure 13.36: Virtual LAN (VLAN): Three VLANs are formed with {1, 5}, {2, 6, 7}, and {3, 4, 8}.**

**NIC.** Network Interface Card (NIC) allows a computer to connect to the network. It allows a host to connect to a hub or a bridge or a switch. NIC uses half-duplex mode of communication (i.e., at a time it either sends or receives packets but not both) when connected to a hub. A smart NIC recognizes that it is connected to a bridge (when it is) and uses either full/half duplex mode of communication. Every NIC has a *Media Access Control (MAC)* address used by the bridge for routing packets to the desired destination. Bridges automatically learn the MAC addresses of the NICs connected to them.

Bridges and switches understand only MAC addresses. The data traffic flows among the nodes of a LAN entirely based on MAC addresses. A packet looks as shown in Figure 13.37. Payload refers to the actual message intended for the destination node. The packet header contains MAC address of the destination node.

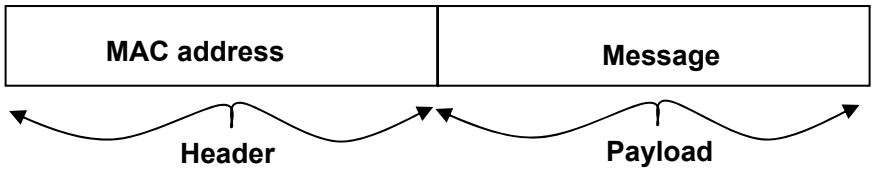



**Figure 13.37: A packet destined to a node on the LAN**

To transmit a packet outside the LAN, i.e., to the Internet, a node simply uses the *Internet Protocol (IP)*. As we have seen already in Section 13.7.2, a packet destined to a node on

the Internet contains an IP address, a 32-bit quantity that uniquely identifies the destination node.

**Router.** In Section 13.7.1, we have already introduced the concept of a router. It is just another host on a LAN with the difference that it understands IP addresses. Any node on a LAN that wishes to send a message to a node on the Internet forms a packet as shown in Figure 13.38 and sends it to the router using the router's MAC address. The "payload" for the router contains a header for the actual message. Embedded in that header is the IP address of the destination node on the Internet. As we have seen already in Section 13.7.1, the router has tables (called *routing tables*) that help it to route the actual message to the node identified by the IP address.

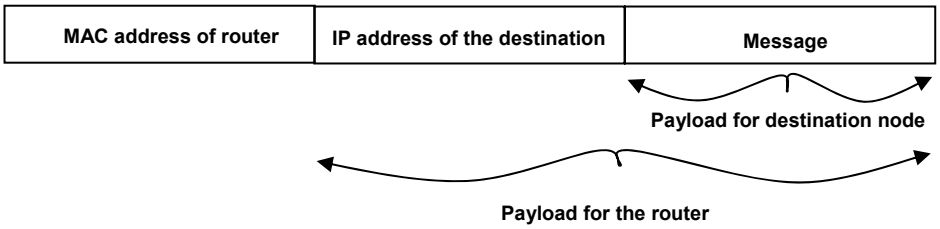



**Figure 13.38: A packet destined for a node on the Internet**

Table 13.7 summarizes the terminologies and gadgetry that are in common use today in deploying computer networks. We have tried to identify each hardware device to the corresponding level of the OSI model (or equivalently, the Internet stack).

| Name of Component | Definition/Function |
|---|---|
| Host | A computer on the network; this is interchangeably referred to as *node* and *station* in computer networking parlance |
| NIC | Network Interface Card; interfaces a computer to the LAN; corresponds to layer 2 (data link) of the OSI model |
| Port | End-point on a repeater/hub/switch for connecting a computer; corresponds to layer 1 (physical) of the OSI model |
| Collision domain | Term used to signify the set of computers that can interfere with one another destructively during message transmission |
| Repeater | Boosts the signal strength on an incoming port and faithfully reproduces the bit stream on an outgoing port; used in LANs and WANs; corresponds to layer 1 (physical) of the OSI model |
| Hub | Connects computers together to form a single collision domain, serving as a multi-port repeater; corresponds to layer 1 (physical) of the OSI model |
| Bridge | Connects independent collision domains, isolating them from one another; typically 2-4 ports; uses MAC addresses to direct the message on an incoming port to an outgoing port; corresponds to layer 2 (data link) of the OSI model |
| Switch | Similar functionality to a bridge but supports several ports (typically 4- |

| | |
|---|---|
| | 32); provides expanded capabilities for dynamically configuring and grouping computers connected to the switch fabric into VLANs; corresponds to layer 2 (data link) of the OSI model |
| Router | Essentially a switch but has expanded capabilities to route a message from the LAN to the Internet; corresponds to layer 3 (network) of the OSI model |
| VLAN | Virtual LAN; capabilities in modern switches allow grouping computers that are physically distributed and connected to different switches to form a LAN; VLANs make higher level network services such as broadcast and multicast in Internet subnets feasible independent of the physical location of the computers; corresponds to layer 2 (data link) of the OSI model |

**Table 13.7: Networking Gear Summary**

### 13.10 Relationship between the Layers of the Protocol Stack

It is interesting to note that each of the transport, network, and link layers deal with data integrity at different levels. For example, TCP – the ubiquitous transport protocol and its cousin IP – the de facto standard for network protocol in the Internet, both include error checking in their specification for the integrity of the packet. At first glance, this may seem redundant. However, in reality, while we say TCP/IP in the same breath thanks to the popularity of the Internet, TCP does not have to run on top of IP. It could use some other network protocol such as ATM. Further, the intermediate routers (which only deal with the network layer) are not required to check the integrity of the packet. Similarly, TCP is not the only transport that may run on top of IP. Therefore, IP specification has to worry about the integrity of the packet and cannot assume that the transport layer will always do the checking.

Similar arguments apply for the network layer not relegating the checking of data integrity to the link layer. Different link layers provide different levels of data integrity. Since in principle the network layer may run on top of different link layer protocols the network layer has to do its own end-to-end assurance of data integrity. Let us return to our original example in Section 13.2 (see Figure 13.3). When Charlie's mom responds to his e-mail from her home computer in Yuba City, the IP packets from her machine uses PPP to talk to the service provider, a protocol called *Frame Relay* to go between service providers, FDDI on the campus backbone, and finally Ethernet to get to Charlie/s computer.

### 13.11 Data structures for packet transmission

Having dived into the different layers of the protocol stack in the previous sections, we will catch our breath for a minute, and slowly surface to the top of the protocol stack. Let us investigate the minimal data structures needed to implement packet transmission.

The transport layer fragments an application level message or a message stream into packets before handing it to the network layer. Further, the network layer routes each packet to the destination individually. Therefore, the destination address is part of each

packet. Further, each packet contains a sequence number to enable the scatter/gather functionality of the transport layer. We refer to such *metadata* that is part of each packet distinct from the actual data as *packet header*. In addition to destination address and sequence number, the header may contain additional information such as source address, and *checksum* (a mechanism for the destination host to verify the integrity of this packet).

The transport layer takes a message from the application layer, breaks it up into packets commensurate with the network characteristics and sticks a header in each packet. Figure 13.39 and Figure 13.40 show the data structures for a simple packet header and a packet, respectively, in C like syntax. The field named **num_packets** allows the transport layer at the destination to know when it has received all the packets to form a complete message and pass it up to the application layer. One might wonder the need for **num_packets** field the header which will go with each packet. Recall, that packets may arrive out of order. The receiving transport layer needs to know on the arrival of the first packet of a new message how much buffer space to allocate for assembling the message in its entirety.

```
struct header_t {
  int destination_address;  /* destination address */
  int source_address;       /* source address */
  int num_packets;          /* total number of packets in
                               the message */
  int sequence_number;      /* sequence number of this
                               packet */
  int packet_size;          /* size of data contained in
                               the packet */
  int checksum;             /* for integrity check of this
                               packet */
};
```

**Figure 13.39: A Sample Transport Level Protocol Packet Header**

```
struct packet_t {
  struct header_t header;   /* packet header */
  char *data;               /* pointer to the memory buffer
                               containing the data of size
                               packet_size  */
};
```

**Figure 13.40: A Sample Transport Level Packet Data Structure**

---

**Example 8:**

A packet header consists of the following fields:

destination_address
source_address
num_packets
sequence_number
packet_size

checksum

Assume that each of these fields occupies 4 bytes. Given that the packet size is 1500 bytes, compute the payload in each packet.

**Answer:**

Size of packet header = size of (destination_address + source_address + num_packets + sequence_number + packet_size + checksum)
= 6 * 4 bytes
= 24 bytes

Total packet size = packet header size + payload in each packet

**Payload in each packet** = Total packet size – packet header size
= 1500 -24
= **1476 bytes**

---

### 13.11.1 TCP/IP Header

It should be emphasized that the actual structure of the header at at each level (transport, network, and link) depends on the specifics of the protocol at that level. For example, as we mentioned earlier, TCP is a byte-stream oriented protocol. It chunks the stream into units of transmission called *segments* (what we have been referring to as packets thus far in the discussion of the transport protocol), and the *sequence number* is really a representation of the position of the first byte in this segment with respect to the total byte stream that is being transported on this connection. Since it is connection-oriented, the header includes fields for naming the endpoints of the connection on which this data segment is being sent. The two endpoints of the connection are called the *source port* and *destination port*, respectively. The data flow in a connection is bi-directional. Thus, the header can piggy-back an acknowledgement for data received on this connection in addition to sending new data. The *acknowledgement number* in the header signifies the next sequence number that is expected on this connection. Further, since TCP has built-in congestion control, the header includes a *window size* field. The interpretation of this field is interesting and important. Each end can monitor the amount of network congestion experienced on this connection based on the observed latencies for messages and the number of retransmissions due to lost packets. Based on such metrics, the sender of a segment advertizes the amount of data it is willing to accept from the other end as the window size field of the header. The sender also chooses the length of each segment it is transmitting based on such metrics. There are also other special fields associated with the segment:

- SYN: This signals the start of a new byte-stream allowing the two endpoints to synchronize their starting sequence number for the transmission.
- FIN: This signals the end of transmission of this byte-stream.
- ACK: This signals that the header includes a piggy-backed ACK, and therefore the acknowledgement number field of the header is meaningful.
- URG: This field indicates that there is "urgent" data in this segment. For example, if you hit *Ctrl-C* to terminate a network program, the application level protocol will translate that to an urgent message at the TCP level.

The IP address of the source and destination are themselves obtained implicitly from the network layer. This is referred to as a *pseudo header* and is passed back and forth between the TCP and IP layers of the Internet protocol stack during segment transmission and reception.
The IP packet format is quite straightforward. Recall that IP may fragment a transport layer "message" (segment as it is called in the case of TCP) into multiple IP packets and reassemble the packets into the original message at the destination. Therefore, in addition to the source and destination IP addresses, the IP header also contains the length of this IP packet and the fragment offset (i.e., the position of this packet in the transport layer message).

### 13.12 Message transmission time

Now that we understand how messages are transported across the network, let us examine more carefully the time taken for message transmission. In Section 13.5, to keep the discussion of the transport protocols simple, we ignored all other timing overheads except the propagation time on the medium itself. Let us understand how the network connectivity of your computer affects the end-to-end transmission time. In order to do this, it is instructive to consider the elements of time involved in message transmission between two end points. This is illustrated in Figure 13.41. We will present a simplified understanding of the message transmission time. The total message transmission time is a composite of the following terms:

1. **Processing delay at the Sender (S):** This is the cumulative time spent at the sender in the various layers of the protocol stack and includes
   - transport level functions such as scattering a message into packets, appending a header to each packet, and affixing a checksum to each packet;
   - network level functions such as looking up a route for the packet;
   - link layer functions such as media access and control as well as framing the data for sending on the physical medium; and
   - physical layer functions specific to the medium used for transmission.

   This factor depends on the details of the software architecture of the protocol stack as well as the efficiency of actual implementation.
2. **Transmission delay ($T_w$):** This is the time needed to put the bits on the wire at the sending end. This is where the connectivity of your computer to the physical medium comes into play. For example, if you have a gigabit link connecting your computer to the network, then for a given message $\mathbf{T_w}$ will be much smaller than if you had a dial up connection to the network.

---

**Example 9:**

Given a message size of 21 Mbytes, (a) compute the transmission delay using a dial-up connection of 56 Kbaud; (b) compute the transmission delay using a gigabit network connectivity.

**Answer:**

(a) Transmission delay = message size / network bandwidth

= (21 * $2^{20}$ * 8 bits) / (56 * $2^{10}$) bits/sec
= 3 * $2^{10}$ secs
= 3072 secs

(b) Transmission delay = message size / network bandwidth
= (21 * $2^{20}$ * 8 bits) / ($10^9$) bits/sec
= 0.176 secs

---

3. **Time of flight ($T_f$)**: This term simplifies a number of things going on in the network from the time a message is placed on the wire at the sending end to the time it is finally in the network interface at the receiving end. In other words, we are lumping together the delays experienced by a message *en route* from source to destination. We enumerate the delays below:
   - **Propagation delay:** This is the time for the bits to travel on the wire from point A to point B. This is dependent on a number of factors. The first factor that comes into play is the distance between the two points and the speed of light. For example, the further apart the two end points, the longer the distance traversed by the bits on the wire, and hence the longer the time taken. This is the reason we get our web accesses to CNN faster when we are in Atlanta as opposed to Bangalore, India. In addition to the distance, other factors come into play in determining $T_f$. $T_w$ already accounts for your physical connectivity to the network. In reality, independent of the specific connection you may have on your computer to the network, the message from your computer may have to traverse several physical links (of different bandwidth) before it reaches the destination. Therefore, in reality, we should account for the propagation delay for each individual physical link *en route* and add them all up to get the propagation delay. For simplicity, we are lumping all of these delays together and calling it the end-to-end propagation delay.
   - **Queuing delay:** Further, as we already mentioned the wide area network is really a network of networks (see Figure 13.3). There are queuing delays incurred by the message in switches along the way. Besides, several intermediate network protocols come into play as your message traverses the network from source to destination. The details of such protocols add to the latency between source and destination. Finally, the destination receives the packet from the wire into a buffer in the network interface.

   We lump all such delays together for the sake of simplicity into the term "time of flight".
4. **Processing Delay at the Receiver (R)**: This is the mirror image of the processing delay incurred by the sender and has all the elements of that delay due to the physical, link, network, and transport layers.

**Total time for message transmission = $S + T_w + T_f + R$** **(1)**

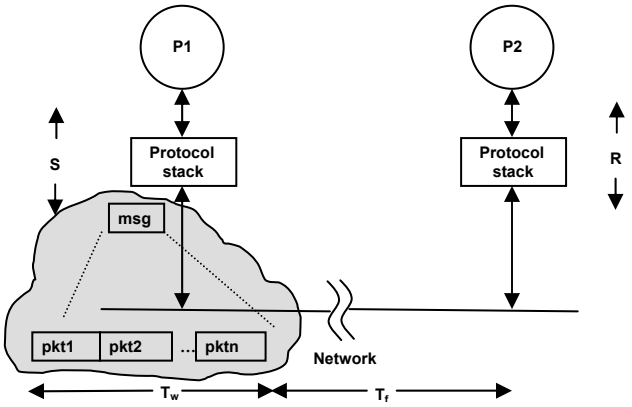



**Figure 13.41: Breakdown of Message Transmission Time**

Equation (1) is also referred to as the *end-to-end latency* for a message. From this equation, we can readily see that the bandwidth of the interface that your computer has to the network does not tell the whole story as to the latency experienced by an application running on the computer. Given the message transmission time, we can compute the *throughput*, defined as the actual transmission rate experienced by an application.

**Throughput = message size / end-to-end latency (2)**

As we already know, a message takes several hops from source to destination. Therefore, between any two network hops, a message experiences each of the following delays: processing delay at the sender, transmission delay, propagation delay on the medium connecting the two hops, queuing delay, and processing delay at the receiver. Just to give the reader a feel for the end-to-end delay experienced in message transmission, we have simplified the exposition of message transmission time in this section by lumping all the delays between source and destination (other than the processing delay and transmission delay at the source, and the processing delay at the destination) as "time of flight".

---

### Example 10:

Given the following:

Processing Delay at the Sender = 1 ms
Message size = 1000 bits
Wire bandwidth = 1,000,000 bits/sec
Time of flight = 7 ms
Processing Delay at the Receiver = 1 ms

Compute the throughput.

### Answer:

Total time for message transmission = $S + T_w + T_f + R$,

Where,

S the Processing Delay at the Sender = 1 ms;

$T_w$ Transmission delay = Message size / wire bandwidth
= 1000/1,000, 000 secs
= 1 ms
$T_f$ time of flight = 7 ms
Processing Delay at the Receiver = 1 ms

Therefore, time for transmitting a message of 1000 bits = 1 + 1 + 7+ 1 ms
= 10 ms

**Throughput** = message size / time for transmission
= 1000/(10 ms)
= **100,000 bits/sec**

---

The following example illustrates the number of packets needed for message transmission in the presence of packet losses.

---

**Example 11:**

Given the following:

| | | |
|---|---|---|
| Message size | = | 1900 Kbits |
| Header size per packet | = | 1000 bits |
| Packet size | = | 20 Kbits |

Assuming a 10% packet errors on DATA packets (no errors on ACK packets, and no lost packets), how many total DATA packets are transmitted by the source to accomplish the above message delivery? Every data packet is individually acknowledged.

**Answer:**

PKT size = header size + payload
20000 = 1000 + payload

Payload in a packet = 19000 bits

Number of packets needed to send the message = 1900 000 / 19000 = 100

With 10% packet loss the total number of DATA packets = 100 + 10 + 1 = **111**

| Total packets Sent | Lose | Successful |
|---|---|---|
| 100 | 10 | 90 |
| 10 | 1 | 9 |
| 1 | 0 | 1 |
| **111** | | **100** |

---

Calculating the cost of message transmission becomes tricky when we have a sliding window protocol.

The following example illustrates the cost of message transmission in the presence of windowing.

---

**Example 12:**

Given the following:

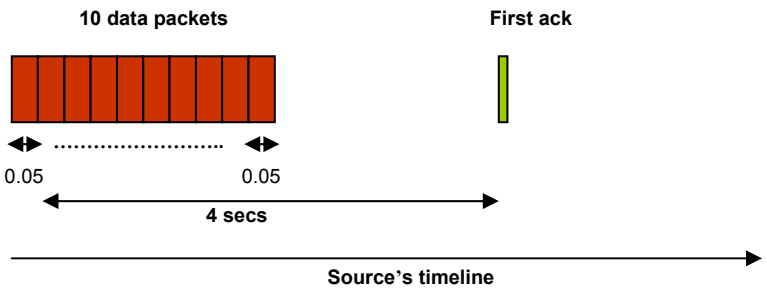
| | | |
|---|---|---|
| Message size | = | 1900 Kbits |
| Header size per packet | = | 1000 bits |
| Packet size | = | 20 Kbits |
| Bandwidth on the wire | = | 400,000 bits/sec |
| Time of flight | = | 2 secs |
| Window size | = | 10 |
| Processing Delay at the Sender | = | 0 |
| Processing Delay at the Receiver | = | 0 |
| Size of ACK message | = | negligible (take it as 0) |

Assuming an error free network and in-order delivery of packets, what is the total time to accomplish the above message delivery?

**Answer:**

PKT size = header size + payload
20000 = 1000 + payload
Payload in a packet = 19000 bits
Number of packets needed to send the message = 1900 000 / 19000 = 100

Transmission delay for each packet = packet size / wire bandwidth
= 20000/400000 secs
= 0.05 secs
Source side latency per packet = $S + T_w$
= Processing Delay at the Sender + Transmission delay
= 0 + 0.05 secs
= 0.05 secs
With a window size of 10, the source places 10 packets on the wire and waits for an Ack. The timing diagram at the source looks as follows:




The destination receives the first data packet 2 secs after it is sent.
End to end latency for a data packet = $S + T_w + T_f + R$

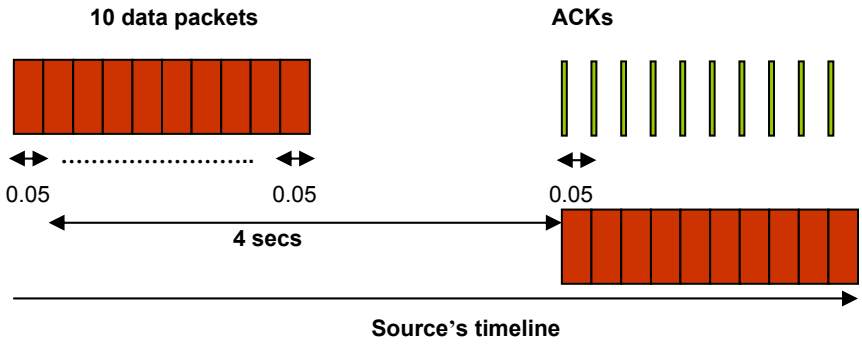
= (0 + 0.05 + 2 + 0) secs
= 2.05 secs

Upon receiving the data packet, the destination immediately prepares an ACK packet.
Destination side overhead for generating an ACK packet
= ($S + T_w$) at destination
= 0 + 0 (since the ACK packet is negligible in size)
= 0

End to end latency for an ACK packet = ($S + T_w + T_f + R$)
= (0 + 0 + 2 + 0) secs
= 2 secs

Thus, the first ACK packet is received 4 secs after the first data packet is placed on the wire as shown in the figure above. More generally, an ACK for a data packet is received 4 secs after it is placed on the wire (assuming no loss of packets).

In a lossless network, the 10 ACKs corresponding to the 10 data packets will follow one another with a 0.05 sec spacing as shown in the figure below:

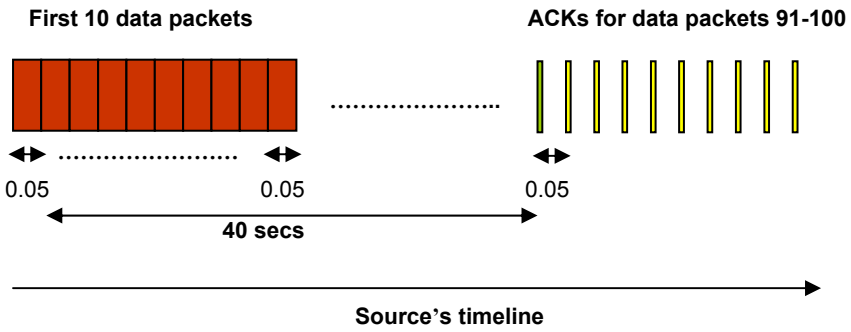



As soon as the first ACK is received, the source can send the next data packet (recall that this is a sliding window of 10). The next 10 data packets follow one another as shown in the figure above.

Thus, in a duty cycle of 4.05 secs, 10 data packets have been sent and 1 ACK has been received at the source. This cycle repeats to send a fresh batch of 10 packets, every 4.05 secs.

To send 100 packets, we need 10 such duty cycles.
Therefore, the time to send all 100 DATA packets = 4.05 * 10 = 40.5 secs

After this time, 100 data packets have been sent and 91 ACKs have been received as shown below. The remaining 9 ACKs (yellow packets in the figure) will arrive with a spacing of 0.05 secs, subsequently as shown in the figure.




Time to receive the remaining 9 ACKs = 9 * 0.05 secs
= 0.45 secs

Total time to accomplish the message delivery
= time for the 100 data packets + time to receive the remaining ACKs
= 40.5 + 0.45 secs
= **40.95 secs**

---

In the above example, the processing delays at the sender and receiver are zero. If either them is non-zero, then it increases the overall end-to-end latency for each packet (See Equation 1). The pipelining of the packets follows similarly as in the above example (see exercises for variations on the above example).

### 13.13 Summary of Protocol Layer Functionalities

To summarize, the functionalities of the five layer Internet protocol stack:

- The application layer includes protocols such as HTTP, SMTP, FTP for supporting specific classes of applications including Web browser, electronic mail, file downloads and uploads, instant messenger, and multimedia conferencing and collaboration. Operating system specific network communication libraries such as sockets and RPC (which stands for remote procedure call, see Section 13.16) also fall in this layer.
- Transport layer provides delivery of application-specific messages between two communication end points. We saw that the functionalities at this level depends on the quality of service requirements of the applications. TCP provides reliable in-order delivery of data streams between the end points including congestion control, while UDP provides message datagram service with no guarantees or message ordering or reliability.
- Network layer delivers to the destination, data handed to it by the transport layer. Functionalities at this layer include fragmentation/reassembly commensurate with the link layer protocol, routing, forwarding, and providing a service model for the transport layer. As we saw earlier, the routing algorithms that determine possible routes to a destination and maintenance of routing and forwarding tables, run as daemon processes at the application level.
- Data link layer provides the interface for the network layer to the physical medium. The functionalities include MAC protocols, framing a packet

commensurate with the wire format of the physical layer, error detection (e.g., detecting packet collision on the Ethernet, or token loss in a token ring network), and error recovery (such as backoff algorithm and retransmission of packet upon collision on Ethernet, or token regeneration on a token ring).

- Physical layer concerns the mechanical and electrical details of the physical medium used for transmission and includes the type of medium (copper, fiber, radio, etc.), the signaling properties of the medium, and the specific handshake for the data link layer to implement the MAC protocol.

### 13.14 Networking Software and the Operating System

As we mentioned in Section 13.3, there are three strong touch points between the operating system and the networking software.

#### 13.14.1 Socket Library

This is the interface provided by the operating system to the network protocol stack. TCP/IP is the backbone for all the Internet related services that we have come to rely on in our everyday life including web browsing, blogging, instant messaging, and so on. Therefore, it is instructive to understand a little bit more about the interfaces presented by TCP/IP for programming distributed applications.

The fact of the matter is that if you are writing a distributed program you will not be directly dealing with the intricacies of TCP/IP. We will explain why in the following paragraphs.

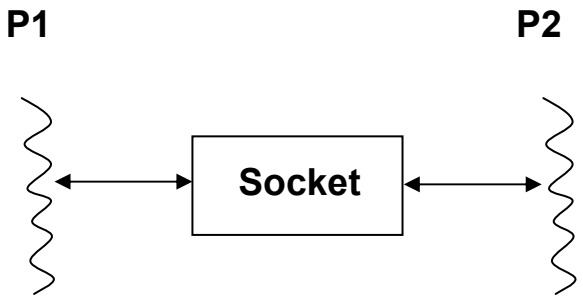



**Figure 13.42: Inter-process Communication using Socket**

If you look at the Internet protocol stack (Figure 13.5), you will recognize that your program lives above the protocol stack shown in this figure. An operating system provides well-defined interfaces for programming distributed applications. For example, the Unix operating system provides the *socket* as an abstraction for processes to communicate with one other (see Figure 13.42).

Figure 13.42 suggests that the socket abstraction is agnostic as to where the two processes P1 and P2 live. For example, they may both be processes executing on the same processor, or different processors of a shared memory multiprocessor (for e.g., an SMP discussed in Chapter 12), or on two different boxes connected by a network. In

other words, the socket abstraction is independent of how the bits move from process P1 to P2.

Protocols such as TCP/IP are vehicles for implementing the socket abstraction. Let us see why it may be advantageous to support *multiple* protocols underneath the socket abstraction. From the point of view of processes P1 and P2, it is immaterial. However, from an efficiency standpoint it is better to use different protocols depending on the location of the endpoints of the communication. For example, consider that the two processes are on the same processor or on different processors of an SMP. In this case, the communication never goes on any external wire from the box. Many of the lower level issues (such as loss of packets, out of order arrival of packets, and transmission errors) disappear. Even if P1 and P2 are on different machines of a local area network (say, a home network), where the chances of packet losses are negligible to nil, there is very little need to worry about such low-level issues. On the other hand, the situation demands a more sophisticated protocol that addresses these low-level issues if P1 is a process running on your computer in your dorm room and P2 is a process running on your friend's workstation on campus.

The desired *type* of communication is another consideration in establishing a communication channel as shown in Figure 13.42. A useful analogy is our postal service. You affix a $0.39 stamp and send a postcard with no guarantee that it will reach its destination. On the other hand, if you need to know that your addressee actually received your letter, you pay slightly more to get an acknowledgement back upon delivery of the letter. Analogous to the postcard example, P1 and P2 may simply need to exchange fixed size (typically small) messages, called *datagrams*, sporadically with no guarantees of reliability. On the other hand, Figure 13.42 may represent downloading a movie from a friend. In this case, it is convenient to think of the communication as a *stream* and require that it be reliable and optimized for transporting a continuous stream of bits from source to destination. The desired type of communication channel is orthogonal to the choice of protocol. Application property drives the former choice while the physical location of the endpoints of communication and the quality of the physical interconnect between the endpoints drive the latter choice.

In Unix, at the point of creating a socket, one can specify the desired properties of the socket including the type of communication (datagram, stream oriented, etc.), and the specific protocol family to use (Unix internal, Internet, etc.).

The operating system level effort in implementing the socket API is somewhat akin to that involved with implementing the threads library, which we discussed in Chapter 12. Such details are outside the scope of this textbook. The interested reader is referred to other textbooks that deal with these issues[18].

Other operating systems in extensive use such as Microsoft Windows XP and MacOS X also support the socket library API.

[18] The Design and Implementation of the FreeBSD Operating System, by Marshall Kirk McKusick, George V. Neville-Neil

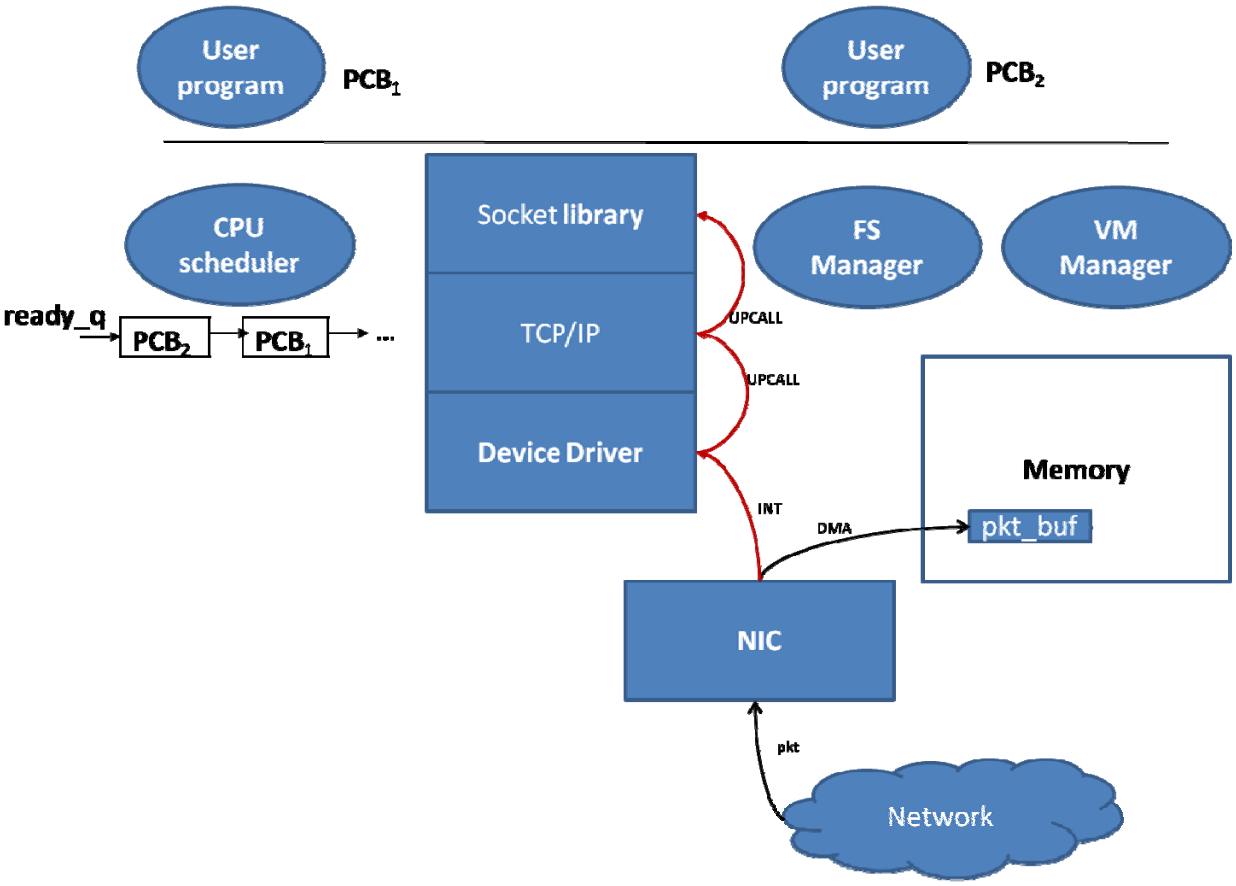



**Figure 13.43: Packet Arrival from the Network. We are already familiar with the other OS functionalities such CPU scheduler, Virtual Memory Manager, and File System Manager from the previous Chapters. The figure shows the layers of the protocol stack in the OS for supporting network programming and the actions taken by the hardware (NIC) and the software upon a network packet arrival.**

### 13.14.2 Implementation of the Protocol Stack in the Operating System

In this day and age, regardless of the specifics of the APIs provided by the operating system for network programming, an operating system has to necessarily provide an implementation of the different layers of the protocol stack to be network ready. While the transport and network layers of the protocol stack are typically implemented in software and form part of the operating system, the link layer (i.e., layer 2) of the protocol stack is usually in hardware. For example, most of you have an Ethernet card in your laptop that provides the functionality of the layer 2 of the protocol stack. With mobile devices, wireless interfaces have become commonplace. These interfaces implement their own link layer protocols that are variants of the random access protocol exemplified by Ethernet. In fact, the IEEE standards for the family of wireless LAN protocols are all derivatives of the IEEE 802 family to which Ethernet belongs. We had a glimpse of the wireless LAN protocol when we discussed CSMA/CA earlier in Section

13.8.4). We will defer detailed discussions on wireless LAN protocols to more advanced courses[19].

As we have seen already, TCP/IP is the dominant transport layer/network layer protocol combination on the Internet today. Correspondingly, standard operating systems (different flavors of Unix, MacOS, and Microsoft Windows) invariably include efficient implementation of the TCP/IP protocol stack. Further, most new research endeavors concerning operating system specialization and fine-tuning will invariably use the protocol stack as the compelling example for justifying new operating system mechanisms.

The protocol stack is a complex piece of software, and typically represents several 10's of thousands of lines of code and several person-years of software development by expert programmers. The interested reader is referred to textbooks that are devoted entirely to this topic[20].

### 13.14.3 Network Device Driver

The discussion of the operating support for networking will be incomplete without a brief look into the network device driver. The Network Interface Card (NIC) allows a host to be connected to a network. A host may have multiple network interfaces depending on the number of networks it is connected to. Correspondingly, the operating system has a device driver for each NIC available in the host. As we mentioned earlier (see Section 13.9), a NIC usually incorporates some of the link layer functionalities of the protocol stack in hardware. The device driver for a specific NIC complements this hardware functionality to enable the NIC to complete the link layer chores. In Chapter 10, we discussed DMA controllers for high-speed I/O. A NIC incorporates a DMA engine to directly move data in/out of host memory out/in to the network. However, there is one fundamental difference between network I/O and disk I/O. The data movement in both directions (i.e., to/from the disk) is initiated by the operating system in response to some user level or system level need (e.g., opening a file or servicing a page fault). Now, consider the network. Sending a packet on the network is certainly initiated by a user level or system level need. On the other hand, the operating system has no control over the arrival of network packets. Therefore, the operating system has to be ready at anytime for this eventuality. This is where the device driver for a NIC comes in. It implements a set of functionality that allows interfacing the NIC to the operating system.

For e.g., in the case of a NIC that connects the host to an Ethernet NIC, the corresponding device driver incorporates the following functionalities:

- Allocating/deallocating *network buffers* in the host memory for sending and receiving packets,
- Upon being handed a packet to send on the wire, setting up the transmit network buffers, and initiating DMA action by the NIC,

[19] The textbook by Kurose and Ross, "Computer Networking: A top down approach featuring the Internet," Addison-Wesley, has a good basic coverage of wireless LAN technologies.
[20] TCP/IP Illustrated, Volume 2: The Implementation (Addison-Wesley Professional Computing Series) by Gary R. Wright, W. Richard Stevens

- Registering network interrupt handlers with the operating system,
- Setting up network receive buffers for the NIC to DMA incoming network packets into the host memory, and
- Fielding hardware interrupts from the NIC, and if necessary, and making upcalls (see Chapter 11 for a discussion of upcalls) to the upper layers of the protocol stack (e.g., communicating a packet arrival event up the food chain),

Figure 13.43 captures the actions both in hardware and software for dealing with a network packet arrival. Since the NIC is made aware of network buffers allocated for packet reception by the device driver, it can immediately use DMA to transfer the incoming packet into the host memory using these pre-allocated buffers. The NIC notifies the device driver about the packet arrival using the hardware interrupt mechanism of the processor (Chapters 4 and 10). Using the upcall mechanism that we discussed in Chapter 11, the device driver alerts the upper layers of the protocol stack about the incoming packet. The interested reader is referred to literature on developing network device drivers for more detail[21].

### 13.15 Network Programming using Unix Sockets

To make the network programming discussion concrete, let us take a closer look at how Unix sockets work from an application program standpoint.

The socket creation call on Unix takes on three parameters **`domain, type, protocol`**.

- **Domain**: this parameter helps pick the protocol family to use for the communication. For example, the choice would be IP if the communicating processes were on the Internet; the choice would be Unix internal if the processes were on the same machine or a LAN.
- **Type:** this parameter specifies the application property desired such as datagram or stream.
- **Protocol:** this parameter specifies the protocol that belongs to the protocol family (given by the domain parameter) that satisfies the desired property (given by the type parameter).

Generality is the intent for providing this choice of parameters in the socket creation call. For example, there could be multiple protocols for a particular protocol family (say Unix internal) that may satisfy the desired type of communication. In reality, there may be exactly one protocol that belongs to a particular protocol family for a desired type of communication. For example, if the protocol family is IP and the type is stream, then TCP may be the only choice available as the transport. Still, separating the choices explicitly is an illustration of the power of abstraction, which we have been emphasizing throughout in this book.

[21] Network Device Driver Programming Guide, http://developer.apple.com/documentation/DeviceDrivers/Conceptual/NetworkDriver/NetDoc.pdf

Figure 13.42 paints a picture as though the two endpoints P1 and P2 are symmetric. However, this is not quite the case; at least, not in the way the communication channel is established. Let us delve a deeper into that aspect in this section.

Interprocess communication using sockets follow a *client/server* paradigm. The communication using sockets may be set up as a connection-oriented one or a connection-less one. The **type** parameter mentioned above allows making this distinction at the creation time of a socket.

Let us consider connection-oriented communication using Unix sockets. One can liken the establishment of the connection between a client and a server to making a phone call. The caller needs to know the number he/she is calling; the callee can accept calls from anyone who cares to call him/her. The client is the caller. The server is the callee.

Let us carry this analogy further. The server does the following in preparation for interprocess communication.

1. Create a socket for communication.

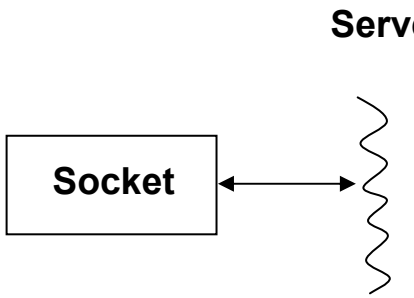



**Figure 13.44: Server side Socket call**

   Notice that this has resulted in creating an endpoint of communication for the server. This is analogous to having a telephone with no connection yet to the outside world. To make the phone useful, one first has to get a number and associate with this phone. That is the next step.
2. The equivalent of a phone number is a *name* (also referred to as *address*), a unique identifier to associate with a socket. The system call to associate a name with a socket is **`bind.`** If the intent is to communicate via the Internet using IP protocol, then the name has two parts <**`host address, port number>.`** The host address is the IP address of the host on which the server is running; the port number is a 16-bit unsigned number. The operating system checks if the port number specified by the server is already in use at the time of the bind call.

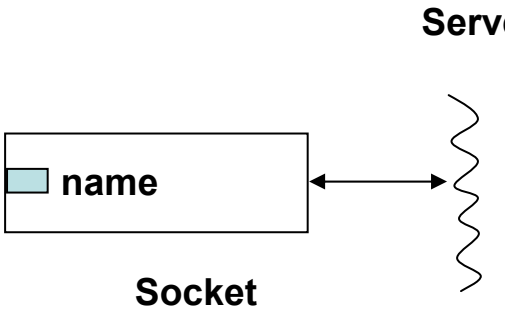



**Figure 13.45: Server socket after bind call**

Once bound the name can be given to others who want to communicate with this server. If it is a telephone number, you may publish it in a directory so that others can look it up. The analogous thing to do in the case of client/server communication is to create a *name server*, a well-known machine that potential clients contact to get the name of the server.

3. Of course, to receive a phone call you have to plug the instrument to the wall socket so that it can listen to incoming calls. First, you have to tell the phone company that you want to allow incoming calls, and whether you would like "call waiting," i. e., how many simultaneous calls you are willing to entertain on your phone line. The equivalent thing to do on the server side is for the server to execute the **`listen`** system call. It tells the operating system how many calls can be queued up on this socket.

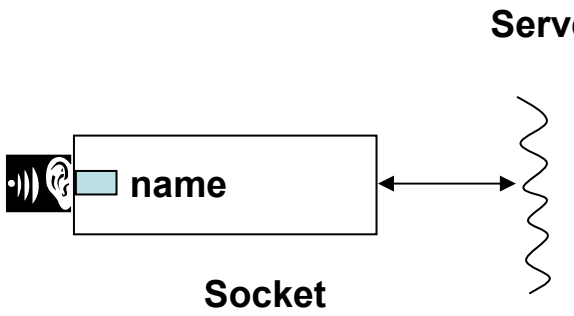



**Figure 13.46: Server socket after listen and accept calls**

4. There is another step on the server side to complete reception of an incoming connection request. This is the **`accept`** system call. This is the equivalent of the phone receiver being on the hook awaiting an incoming call once the phone company has turned on your phone service.

Now let us see what the client has to do to establish communication with the server.

1. Create a client side socket. This is equivalent to getting a phone for the caller.
2.

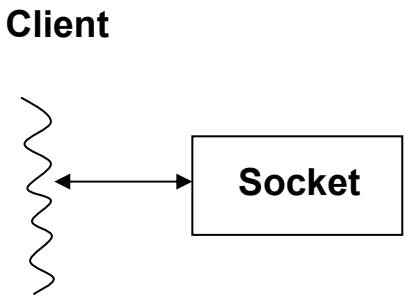



**Figure 13.47: Client side Socket call**

3. To make a phone call, the caller has to dial a number. The **`connect`** system call accomplishes the equivalent thing on the client side. It connects the client side socket to the **`name`** published by the server as shown in Figure 13.48.

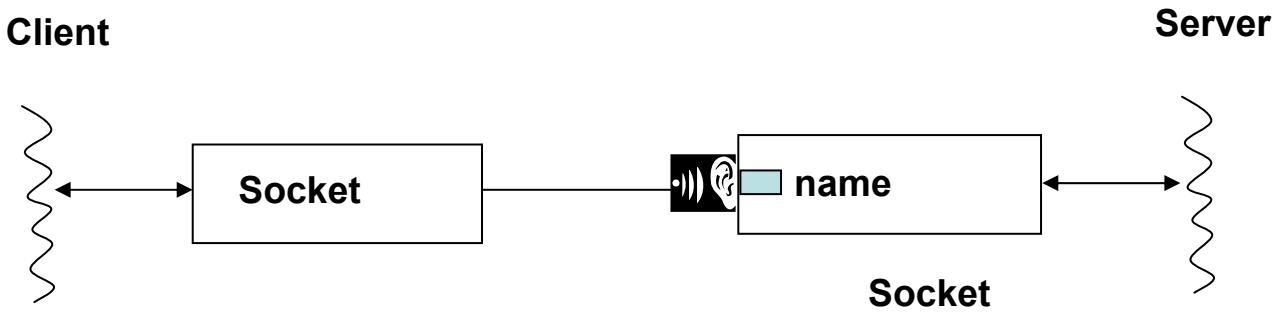



**Figure 13.48: Client-server relationship after client executes connect system call**

The client and server are not quite ready to communicate yet. The connect system call is analogous to dialing the phone number. Unless the callee picks up the phone the call is not complete. The **`accept`** system call on the server side accomplishes the analogous thing in this client/server set up. In a telephone call, one has to pick up the receiver to establish the connection. In the socket world, recall that the server has already indicated its willingness to accept incoming calls on this connection. Therefore, the operating system completes the establishment of the connection between the client and the server as shown in Figure 13.49. Note that if you have call waiting on your phone, you will still be able to know about incoming calls while you are talking to someone. The operating system provides the same facility with sockets. It creates a *new data socket* for the newly established connection. The operating system will implicitly queue up connection requests that may be coming in from other clients on the original named socket on which a listen call has been placed.

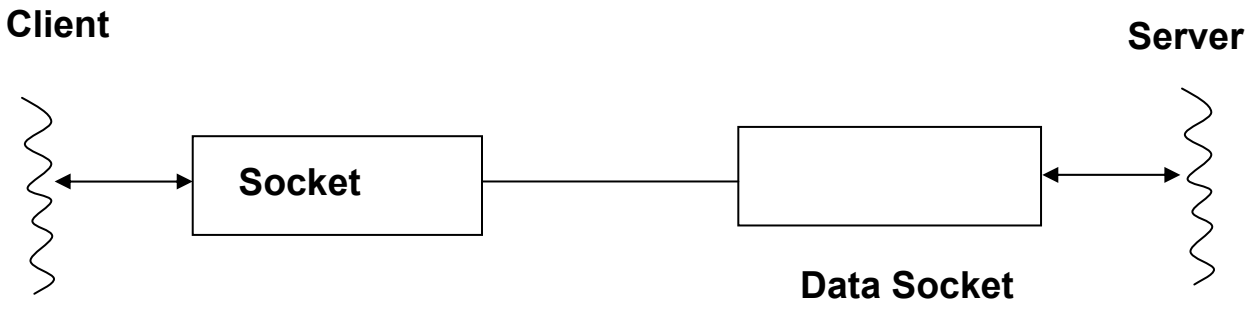



**Figure 13.49: Client-server relationship after connection establishment**

At this point, both the client and the server are ready to exchange messages in a symmetric manner. Thus although there is asymmetry in the way the connection is established (just as in the case of a phone call), the actual communication is truly symmetric.

You may be wondering why both the client and the server have to execute **`socket`** system calls. From earlier chapters, we know that each process executes in its own address space. The data structures created because of a system call live in the address space of a particular process. Thus, we need a representation of the socket abstraction in each of the address spaces of the communicating processes (see Figure 13.50). All the other system calls (bind, listen, accept, and connect) enable the two sockets to be connected together via the operating system and the network protocol stack to realize the abstract picture shown in Figure 13.42.

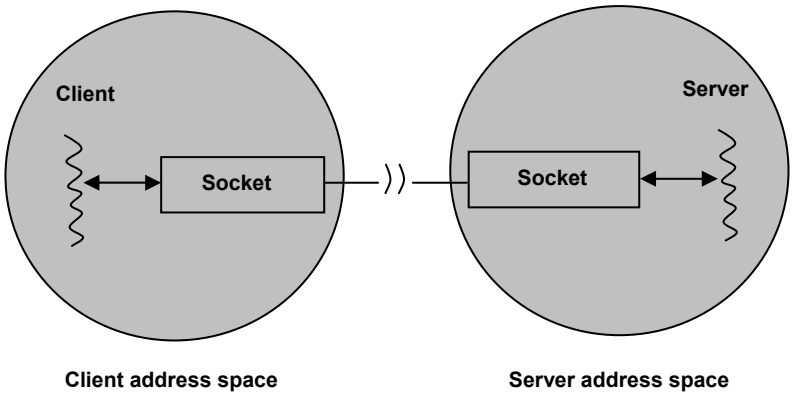



**Figure 13.50: Sockets in Client-Server Address spaces**

While the above discussion focused on a stream-type socket that requires connection establishment, the communication is much more simplified if the socket type is a datagram. In this case, the server neither has to execute a listen nor an accept call at all. The datagram received from a client is simply stored in the datagram socket created by the server. Similarly, if the client creates a datagram socket, then it can simply send and

receive data on this socket by providing the address of the server-side socket (i.e., host address and port number).

To summarize, Unix provides the following basic system calls for processes to communicate with one another regardless where they execute:

- **`Socket`**: create an endpoint of communication
- **`Bind`**: bind a socket to a name or an address
- **`Listen`**: listen for incoming connections requests on the socket
- **`Accept`**: accept an incoming connection request on a socket
- **`Connect`**: send a connection request to a name (or address) associated with a remote socket; establish the connection by creating a data socket for this connection if the server has already posted an accept system call
- **`Recv`**: receive incoming data on a socket from a remote peer
- **`Send`**: send data to a remote peer via a socket

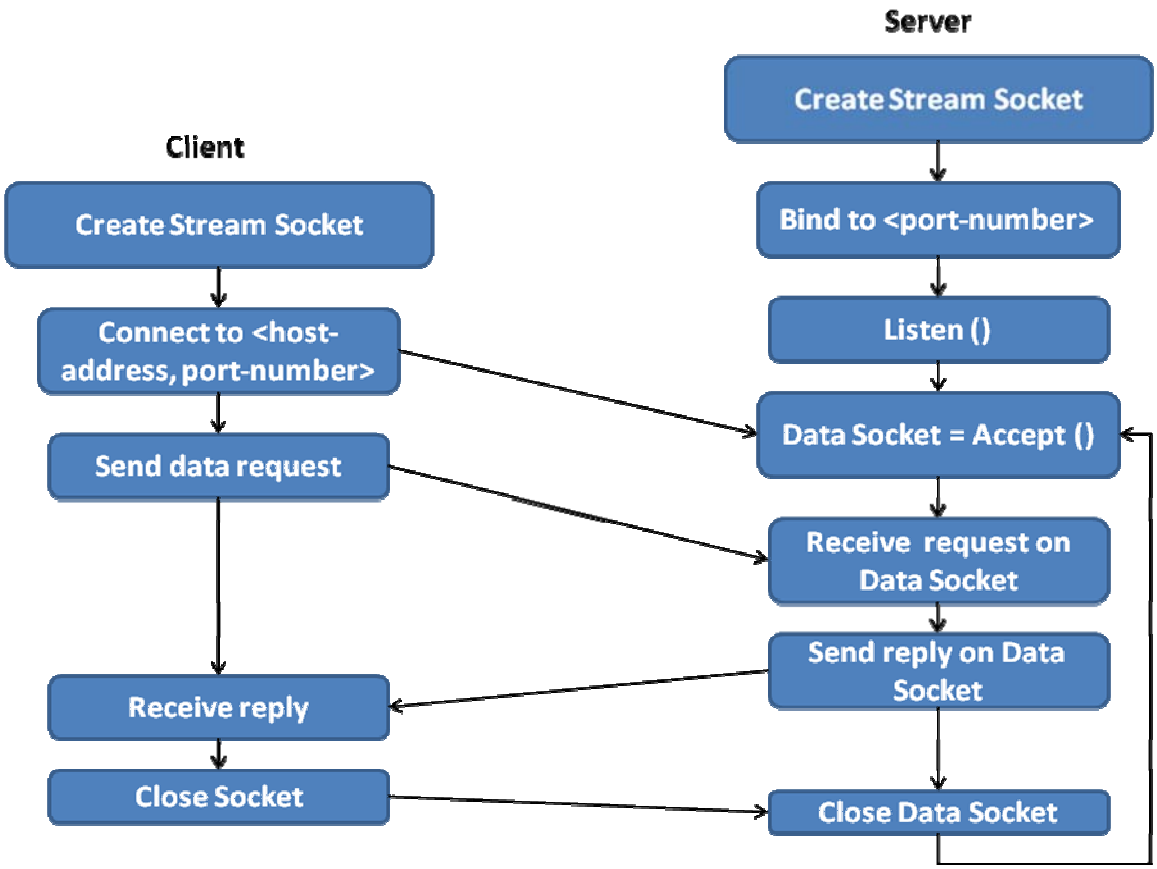



**Figure 13.51: Data Communication on a Stream Socket. The server creates a socket, binds a port number to it, and makes the blocking accept system call indicating its willingness to accept connections requests. The client creates a socket and connects to the server using the <host address, port-name> tuple. The operating system creates a new data socket for the newly established connection on**

**which the client and server can exchange data streams. The data socket is specific to this connection and will be closed when the client closes the socket. At that point, the server can go back to waiting for new connection requests on the original stream socket.**

Figure 13.51 shows the protocol between the client and the server for establishing a stream-oriented socket for communication. The listen system call essentially lets the operating system know how many connections are expected to be simultaneously supported on the stream socket. Each time a new connection is established (client connect call matches with server accept call), a new data socket is created by the operating system. The life of this new data socket is determined by the lifetime of the connection. As soon as the client closes the connection, the newly created data socket is closed as well.

The listen call is a powerful mechanism for implementing a multi-threaded server. Once a socket is created and bound, multiple threads can post **`accept`** calls on the socket. This allows multiple simultaneous client-server connections to be entertained on the same socket (each with its own individual data socket) subject only to the limit specified in the listen call.

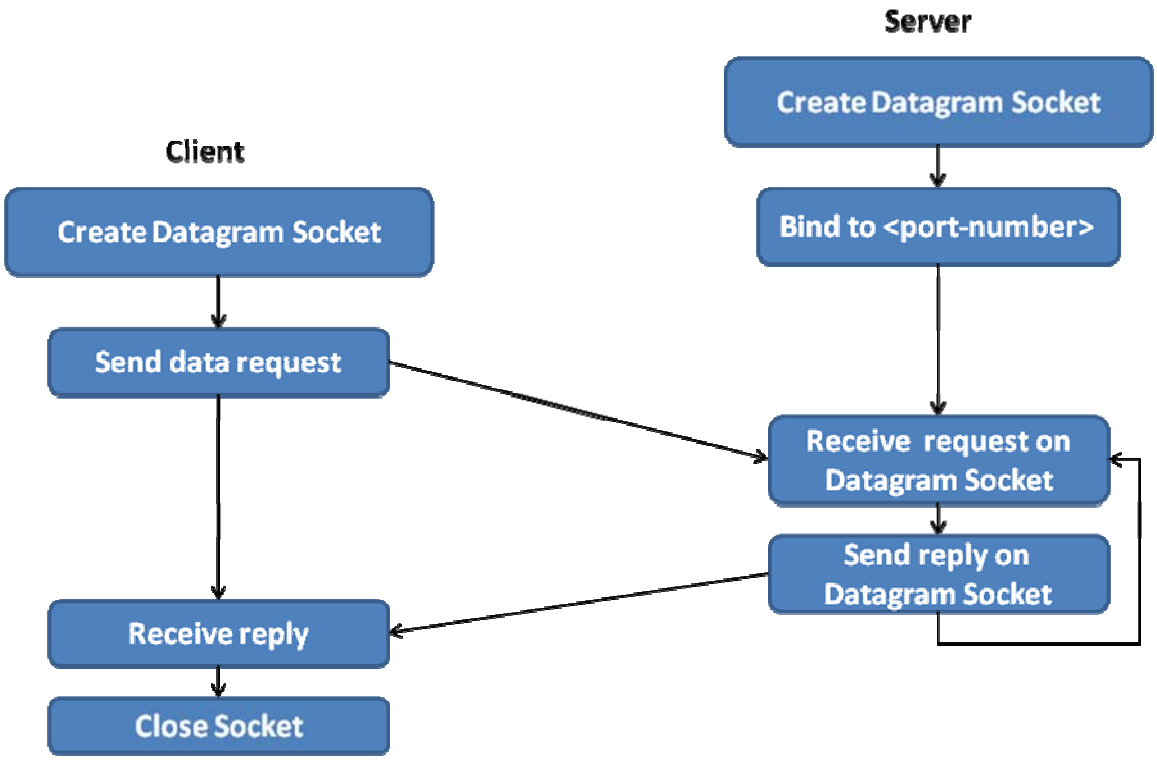



**Figure 13.52: Communication on a Datagram Socket. Once the server creates a datagram socket and binds a port number to it, data transmission and reception can start on this socket. There is no need for a connect call from the client for the data exchange between the client and the server.**

Figure 13.52 shows the protocol between the client and server for establishing a datagram oriented socket for communication. Note that the server is ready to send/receive on a datagram socket as soon as it is bound to port number. Similarly, client can start sending and receiving on a datagram socket without the need for an explicit connect call.

Please see Appendix A for an example client-server program using Unix sockets.

There are a number of issues that demand the attention of a programmer using Unix sockets to implement a distributed program. Let us itemize some of these issues:

- Can two processes communicate if the associated socket properties (domain and/or type) do not match?
- How does the server choose a port number?
- What does name translate to for Unix domain sockets?
- What happens if the name that the client wants to connect to does not exist?
- What happens if physical link between the client and server breaks for some reason?
- Can multiple clients try to execute a connect call to the same name?

The answers to these questions depend on the exact semantics of the socket library calls as well as the implementation choices exercised by the developer who implements the socket library. The purpose of raising these questions is quite simply to perk the interest of the reader. The answers to these questions can be readily obtained by perusing the man pages[22] of the socket library calls on any Unix system.

The operating system uses the appropriate transport protocol (commensurate with the desired socket semantics) to implement the socket system calls. Discussion of the implementation details of the socket library is beyond the scope of this book. The reader is referred to more advanced books in networking and operating systems for such a discussion.

Of course, we have glossed over many messy details to keep the discussion simple. The Unix operating system provides a number of utilities to assist in network programming with sockets.

With the popularity of Unix operating systems for network servers (such as file servers and web servers), and the advent of the World Wide Web (WWW), network programming with sockets has taken on enormous importance. The interested reader is encouraged to take advanced courses in operating systems to get hands on experience with distributed programming with sockets, as well as to learn how to build socket abstraction in the operating system.

[22] See man pages on line at: http://www.freebsd.org/cgi/man.cgi

### 13.16 Network Services and Higher Level Protocols

As a concrete example, let us understand a network application such as *ftp (file transfer protocol)*. There is a *client* and *server* part to such services. The host machine at the client end opens a network connection with the remote host using the transport protocol (TCP/IP for example) to talk to the ftp server on the remote host. Upon completion of the file transfer, the client and server agree to close the network connection. Other network applications such as a *web browser* and *mail* work similarly.

It is interesting to understand how some LAN services work quite differently from these WAN (Wide Area Network) services. For example, all of us routinely use file servers in our everyday computing. When we open a file (for example using **`fopen`** on Unix), we may actually be going across the network to a remote file server to access the file. However, such LAN services do not use the traditional network stack. Instead, they use a facility called *remote procedure call* (RPC) in the Unix Operating System. Figure 13.53 shows the idea behind RPC. Process P1 makes what appears like a normal procedure call to *foo*. This procedure foo executes remotely in process P2 on another machine across the network. The fact that the procedure executes remotely is invisible to the users of RPC.

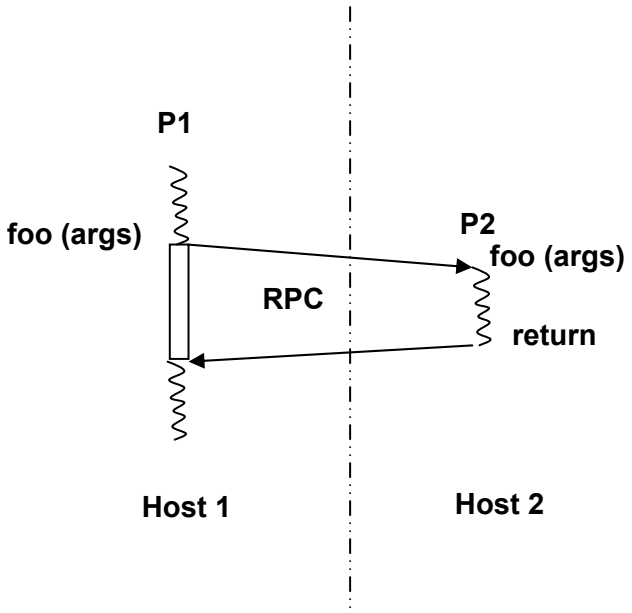



**Figure 13.53: RPC in Unix**

*Network File System (NFS)* exists on top of the RPC mechanism. Figure 13.54 shows at a high-level the handshake involved in a network facility such as NFS. The **`fopen`** call of the user becomes an RPC call to the NFS server, which performs the file open system command on the named file residing at the server.

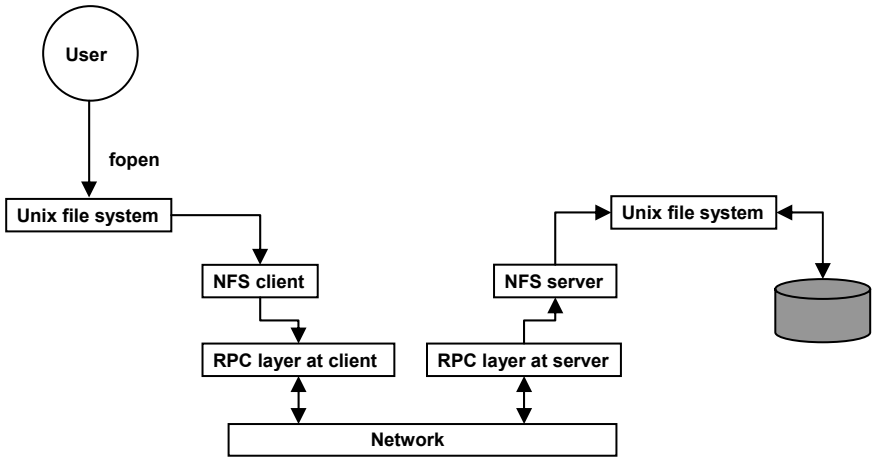



**Figure 13.54: Unix Network File System (NFS)**

The brief discussion presented in this subsection on network services and higher-level protocols raises several interesting questions:

1. How do we design RPC systems?
2. What are the semantics of a remote procedure call in comparison to local procedure calls?
3. How does RPC deal with sources of failures in network transmissions?
4. We have seen implementation of file systems earlier in Chapter 11. What are the semantic differences in implementing a network file system?

This subsection gives a glimpse into the fascinating area of distributes systems. We hope the reader pursues advanced courses to get a deeper understanding on these and other distributed systems topics.

### 13.17 Summary

In this chapter, we covered the basics of computer networking. We reviewed the basic requirements for supporting networked communication in Section 13.3, and introduced the 5-layer Internet protocol stack in Section 13.4.1. At the heart of the Internet protocol stack is the transport layer, which we covered in detail in Section 13.6. It is responsible for providing an abstraction to the applications that is independent of the details of the network. Its primary functionality is ensuring in-order delivery of messages, supporting arbitrary sized messages, and shielding the applications from loss of data during message transmission. There is a tension between fairness, reliability, and eagerness to utilize the communication channel fully, all of which combine to present a number of possible protocols at the transport layer.

We started the transport protocol discussion with the simple stop-and-wait protocol (Section 13.6.1), and ended with the transport protocols used in the Internet today (Section 13.6.5). We covered network layer functionalities in detail in Section 13.7. This layer is responsible for providing a logical addressing scheme for the nodes in the network (Section 13.7.2), routing algorithms for the passage of packets of a message

from source to destination (Section 13.7.1), and offering a service model for packet delivery (Section 13.7.3). The routing algorithms discussed in Section 13.7.1 include Dijkstra's shortest path, distance vector, and hierarchical routing. In Section 13.7.3, we covered circuit switching, packet switching, and message switching as three possible options for using the network resources (such as intermediate nodes and routers between source and destination). Packet switching is the paradigm widely used in the Internet, and we discussed two possible service models on top of packet switching, namely, virtual circuit and datagram. In Section 13.8, we discussed link layer technologies, paying special attention to Ethernet (Section 13.8.1), which has become the de facto standard for local area networks. This section also covered other link layer technologies such as Token Ring (Section 13.8.5), FDDI and ATM (Section 13.8.6). The networking hardware that connects the host to the physical layer is discussed in Section 13.9. Protocol layering is a modular approach to building system software, and we discussed the relationship between the different layers of the Internet Protocol stack in Section 13.10. We discussed the data structures for packet transmission and the components of the message transmission time on the network in the subsequent two sections (Sections 13.11 and 13.12). We summarized the functionalities of the 5-layer Internet Protocol stack in Section 13.13.

Section 13.14 went into the issues in incorporating the network protocol stack into the operating system including a discussion of the socket library that sits at the top of and the device driver that sits below the TCP/IP protocol in most production operating systems. We presented an intuitive feel for network programming with Unix sockets in Section 13.15, and a glimpse of higher level network services (such as network file system) in Section 13.16.

This chapter concludes with a historical perspective of computer networking.

### 13.18 Historical Perspective

This section starts with a journey through the evolution of networking from the early days of computing.

#### 13.18.1 From Telephony to Computer Networking

Let us first review the evolution of telephony since computer networking owes a lot to the advances in telephony. Prior to 1960's, the telephone infrastructure was entirely analog. That is, when you picked up a phone and called someone, the wires that connect the two devices carried actual voice signal. In principle, one could climb on a telephone pole and eavesdrop on private conversations, with a couple of electrical connections and a pair of headphones. In the 60's, telephony switched from analog to digital. That is, the infrastructure converts the analog voice signal to digital, and sends the bits over the wire. The network still uses circuit switching (see Section 13.7.3) but the audio signal is sent as 1's and 0's. At the receiving end, the infrastructure converts the digital data back to the original analog voice signal, and delivers them to the end user via the telephone (Figure 13.55).

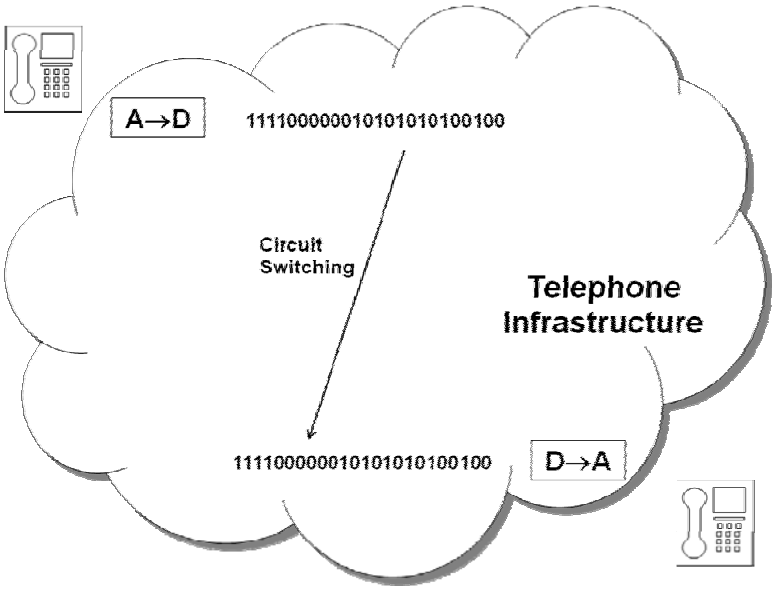



**Figure 13.55: Circuit-switched digital Telephony**

1960's marked the era of *mainframes* in the computing evolution. These machines operated in a batch-oriented multiprogramming environment using *punched cards* as input/output medium. *Cathode Ray Terminal (CRT)* based display devices and keyboards made their way displacing *punched cards* as input/output medium for users to interact with the computer. This started the era of *interactive* computing and *time-shared* operating systems on the mainframes (Figure 13.56).

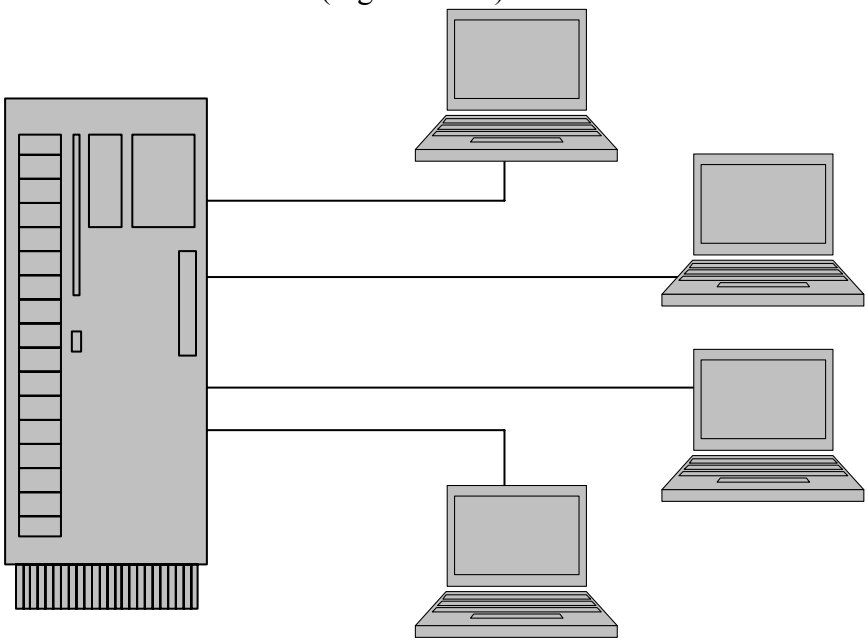


**Figure 13.56: Terminals connected to a Mainframe**

With the parallel advances in telephony, the advent of CRTs opened up a new possibility, namely, the CRTs need not necessarily be near the mainframe computer, but it could be located at a remote location, such as the home of a user. After all, the telephone

infrastructure was carrying digital data. Therefore, the infrastructure does not really care if the bits on the wire represent voice or data. Unfortunately, the telephone infrastructure assumes that the input/output is analog (since it is meant for voice transmission) even though internally it is all digital. Therefore, there is a missing link from the remote CRT to get to the telephone infrastructure; and a similar missing link from the analog output of the infrastructure back to the digital data expected by the mainframe (Figure 13.57). This missing link is the *modem* – modulator/demodulator – an electronic device that converts digital data to analog (modulator) at the sending end, and analog data to digital at the receiving end. Of course, both directions (CRT to mainframe, and mainframe to CRT) require this digital-analog-digital conversion.

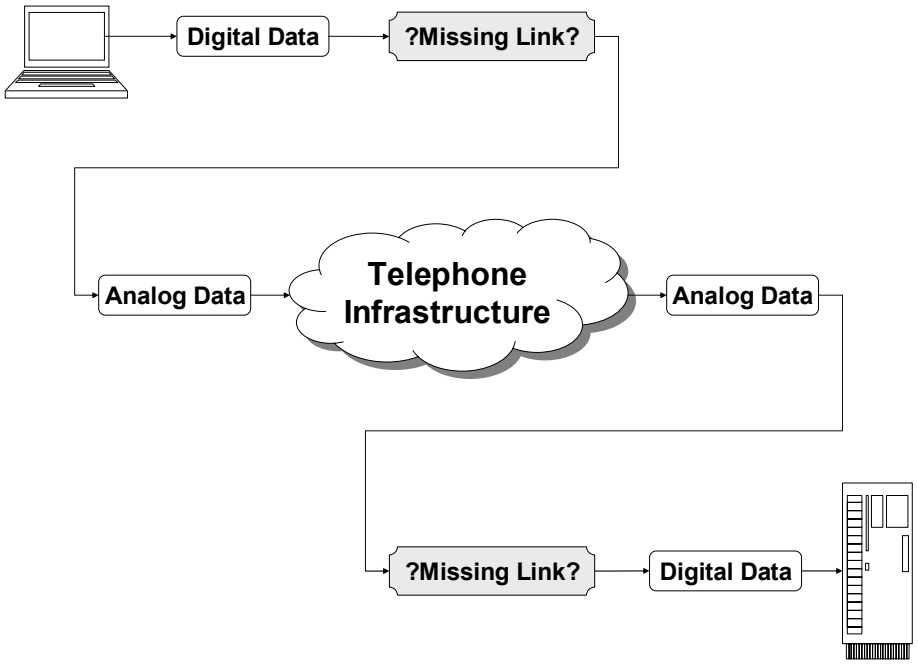



**Figure 13.57 Even though the telephone infrastructure operates internally on digital data, the input/output to the infrastructure is analog (voice).**

In 1962, AT&T Bell Labs introduced the first commercial *full-duplex* modem that does both the modulation and demodulation at each end (Figure 13.58). This marked the birth of what we know today as *telecommunication*. In 1977, a Georgia Tech graduate named Dennis Hayes invented *PC modem* that standardized the communication language between terminal and modem. This laid the groundwork for online and Internet industries to emerge and grow, and to this day has come to stay as industry standard for modems.

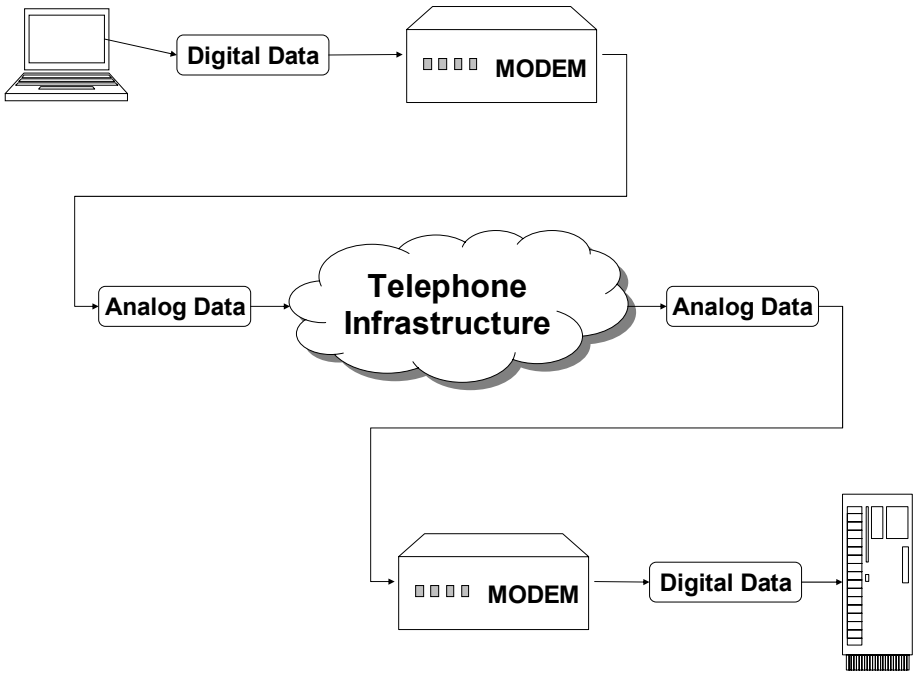



**Figure 13.58 Modems connecting terminals to the mainframe.**

### 13.18.2 Evolution of the Internet

Creating the technology for computers to talk to one another over a long distance was one of the passions of many a computer pioneers. The military was very interested in this technology coming to fruition for strategic reasons. In parallel with the development of the modem for connecting terminals over phone lines to the mainframe, several sharp minds were thinking about connecting computer over long distances from the early 1960's. In 1968, a federal funding agency, *DARPA* (which stands for *Defense Advanced Research Projects Agency*) with the help such sharp minds, drafted a plan for the first computer network. The first packet-switched computer network called ARPANET (which stands for Advanced Research Projects Agency Network), connected four computers, one at UCLA, the second at Stanford, the third at UC-Santa Barbara, and the fourth at University of Utah. The "router" in the network was called *Interface Message Processor (IMP)*, built by a company called BBN (which stands for Bolt, Beranak, and Newman Inc.). The IMP system architecture required a careful balance of the hardware and software that would allow it to be used as a store-and-forward packet switch among these computers. The IMPs used modems and leased telephone lines to connect to one another. ARPANET was first tested in 1969, and a networking luminary Leonard Kleinrock, is credited with successfully sending the first network "message" from UCLA to Stanford.

Of course, after this initial test, ARPANET soon burgeoned into the modern day Internet. One of the key developments that spurred this growth is the development of reliable communication protocols, namely, TCP/IP by two networking pioneers, Vinton Cerf and Robert Kahn. Fittingly, in 2004, they were honored with the highest award in computer science, the *Turing Award*, in recognition of their contribution to the development of

network communication protocols. It is interesting to note that the basic networking ideas we covered in this chapter including transport protocols and store-and-forward packet-switched routing were fully operational in the Internet by the mid to late 70's. Thanks to the Berkeley Software Distribution effort from the University of California, Berkeley, the TCP/IP protocol stack made it into the Unix operating system in the early 80's. Owing to the subsequent popularity of Unix, initially in the desktop market and subsequently in the server markets, pretty soon the Internet protocol stack became a standard feature in all flavors of Unix. Even IBM, in the early 80's, decided to include the Internet protocol stack in its VM operating system through a partnership with the University of Wisconsin-Madison[23]. Thus, pretty much all the major vendors had adopted the Internet protocol stack as standard offering in their computers. However, it is only in the late 90's that Internet really took off as a household name. There are a couple of reasons for that time lag. First, computers did not reach the masses until the invention of the PC and its commercial success. Second, there was not a killer application for the Internet until the birth of the World Wide Web. Today, of course, even our grandmother would tell us that a computer is pretty much useless unless it is connected to the Internet. The explosive growth of the Internet gave birth to companies such as CISCO that specialized in making specialized hardware boxes to serve as routers.

Of course, one could write an entire book on the evolution of the Internet, but the purpose here is to give a glimpse of the road we have traversed until now. One has to simply look at the evolution of networking to appreciate the connectedness of the three topics we have been stressing in this textbook, namely, architecture, operating systems, and networking.

### 13.18.3 PC and the arrival of LAN

In 1972, a document company called Xerox Corporation designed the world's first *personal computer* called *Alto*, named after the location of its Palo Alto Research Center (PARC).

In the mid-70, Metcalf and Boggs, working in Xerox PARC, invented *Ethernet* a network for computers within a building to talk to one another. This marked the birth of the *Local Area Network (LAN)*. Ironically, Xerox never marketed the PC or the Ethernet technology. Other companies, first Apple and later IBM picked up the PC idea and the rest is history. In 1979, Metcalf founded 3Com to develop the LAN market, and successfully convinced the computer industry to adopt Ethernet as a LAN standard.

### 13.18.4 Evolution of LAN

**Thicknet.** A *coaxial cable* serves as the physical medium in Ethernet (Figure 13.59); the innermost thick copper wire carries the signal and the outer conductor that runs concentric to the first one (separated by the white insulation) serves as the ground. The very first iteration of Ethernet used a thick coaxial cable (called *thicknet* for that reason), and *vampire taps* (Figure 13.60) on the cable to establish a connection for each computer. The coaxial cable runs through an entire office complex connecting all the computers together. The cable that connects an office computer to Ethernet is the *Attachment Unit*

[23] One of the co-authors of this textbook (Ramachandran), implemented the mail transfer protocol, SMTP, as part of this IBM project while he was a graduate student at UW-Madison.

*Interface (AUI)* cable, whose length can be at most 50 meters. It is customary to use the notation *xBASEy* to denote the type of Ethernet connection. For example, 10BASE5 refers to an Ethernet supporting *10* Mbit/sec data transfer rate using *BASE* band signaling, and a maximum distance of 500 meters between any two computers. Thicknet with vampire taps were in use from 1979 to 1985.

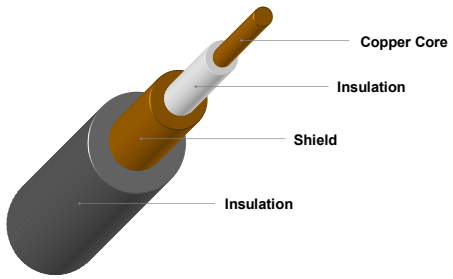



**Figure 13.59: Coaxial cable**

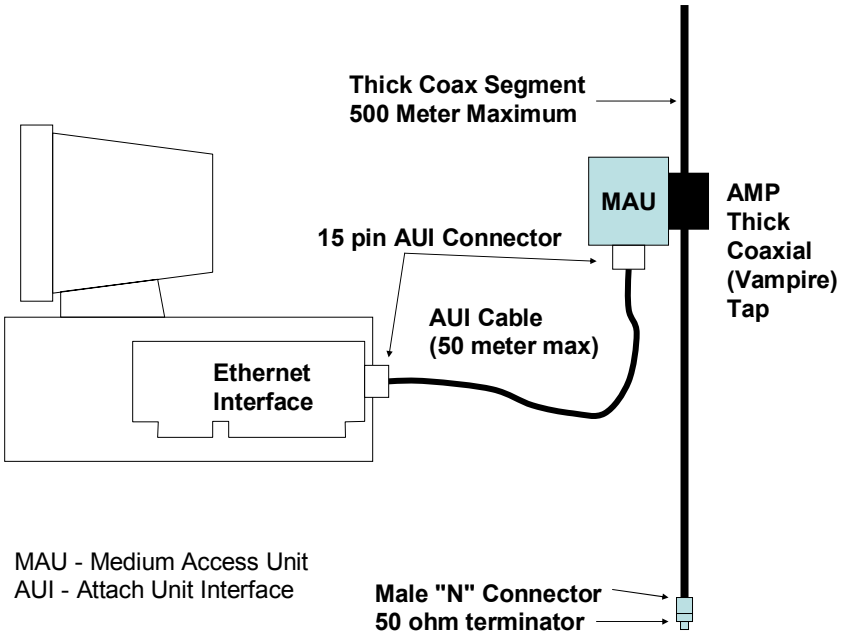



**Figure 13.60: Computer connected to Ethernet coaxial cable via vampire tap. The maximum distance between any two nodes connected by the coaxial cable can be 500 meters.**

**Thinnet.** As the name suggests, this kind of Ethernet uses thinner coaxial cable for the Ethernet medium, with BNC[24] connectors for connecting the computer to the coaxial cable. Due to the relative thinness of the wire (implying greater resistance and hence greater signal degradation with length), 200 meters is the maximum length of a thinnet cable connecting two computers. The logical bus is a daisy chain of these cables from unit to unit as shown in Figure 13.61. Thinnet (a 10 Mbit/sec data transfer rate thinnet would be denoted 10BASE2) was in vogue from 1985 to 1993.

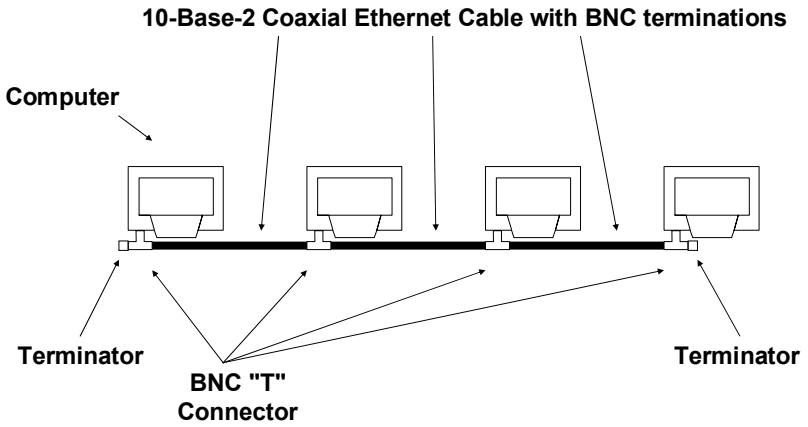



**Figure 13.61 Daisy chain arrangement of Thinnet 10BASE2 Ethernet**

**Fast Ethernet.** The daisy-chained units were a source of connector problems in maintaining the LAN. For example, the LAN is broken even if one BNC connector gets unplugged. Advances in circuit design and VLSI came to the rescue. In the early 90's, instead of a physical wire connecting the units, electrical engineers designed a *hub*, which is a multiplexer/transceiver. The hub is an electronic component that is logically equivalent to a bus. A hub has several *ports*, each of which is a connection point to a computer. Riding on the advances in Telephony and the use of modular jacks (called RJ45 connectors), Ethernet switched to using a *twisted pair of wires* similar to the telephone wires and RJ45 connectors at the two ends to connect the computers to the hub (Figure 13.32). This changed the landscape of LAN quite dramatically. Deploying an Ethernet LAN is simply a matter of hooking up computers to hubs and connecting the hubs together (Figure 13.33). Logically, all the computers exist on a bus via the hubs. Length of the cable is irrelevant since the computers connect to a hub over a short distance (few tens of meters). 100BASE-T (T for twisted pair), often referred to as *Fast*

[24] BNC connector is the kind used quite commonly for video connections. The letters in the acronym stand for the style of locking (Bayonet mount locking) and the names of the inventors (Neil and Concelman).

*Ethernet*, denotes a 100 Mbit/sec data transfer rate Ethernet using a twisted pair for connecting the computers to the hub.

**1GBase-T and 10GBase-T.** Circa 2009, Gigabit Ethernet is becoming a standard for LAN interconnects. NICs that provide such high-bandwidth network connectivity for the hosts use a twisted pair of wires to connect the host to a switch. The length of the wire can be at most 100 meters.

### 13.19 Review Questions

1. Describe the function of a modem. Include in your answer communication in both directions.

2. Telephone networks use analog or digital transmission of voice traffic. Are calls routed over wires dedicated to one call at a time?

3. Compare and contrast Ethernet and token ring networks

4. What does the abbreviation CSMA/CD stand for and what does it mean?

5. Compare and contrast each of the following: NIC, Hub, Repeater, Bridge, Switch, Router

6. How many collision domains are formed when **n** nodes are connected to the **p** ports on a switch?

7. What is each layer of the 7 layer OSI model responsible for?

8. Normally, when a router discards a packet it sends a message back to the sender letting it know. When doesn't this happen?

9. Why do we need Network Protocols?

10. Distinguish between circuit switching and virtual circuit.

11. A knowledgeable computer engineer designs a switch (such as the one discussed in this chapter) that was made using bridges. In order to save money he uses repeaters instead of bridges. Would you buy one of these switches? Why, or why not?

12. What is the purpose of a sliding window?

13. How do protocols such as TCP/IP deal with problems such as out of order delivery, lost packets, duplicate packets, etc.

14. Describe the basic function of a token ring network.

15. How is a checksum used and why is it necessary?

16. A message has 13 packets, and the time to send a packet from source to destination is 2 msec. Assuming that the time to send/receive the packet and the ACK are negligible compared to the propagation time on the medium, and no packet loss, how much time is required to complete the transmission with the sliding window protocol with a window size of 5?

17. Given the following:

| | | |
|---|---|---|
| Message size | = | 1900 Kbits |
| Header size per packet | = | 1000 bits |
| Packet size | = | 20 Kbits |
| Bandwidth on the wire | = | 400,000 bits/sec |
| Time of flight | = | 2 secs |
| Window size | = | 8 |
| Processing Delay at the Sender | = | 0 |
| Processing Delay at the Receiver | = | 0 |
| Size of ACK message | = | negligible (take it as 0) |

Assuming an error free network and in-order delivery of packets, what is the total time to accomplish the above message delivery?
Hint: Note that the window size specifies the maximum number of outstanding packets at any point of time. After all the data packets have been sent out, the source has to wait for the remaining ACKs to come back from the destination for the transmission to complete.

18. Given the following about the transport and network layers:

| | | |
|---|---|---|
| Data packet size | = | 20000 bits |
| Size of ACK message | = | negligible (take it as 0) |
| (Each data packet individually acknowledged) | | |
| Transport Window size | = | 20 |
| Processing Delay at the Sender | = | 0.025 secs per packet (only for data) |
| Processing Delay at the Receiver | = | 0.025 secs per packet (only for data) |
| Packet loss | = | 0% |
| Packet errors | = | 0% |
| Bandwidth on the wire | = | 400,000 bits/sec |
| Time of flight | = | 4 secs |

What is the total time (including the reception of the ACK packets) to complete the transmission of 400 packets with the above transport?
Hint: Note that from the point of view of determining the duty cycle at the source, all the components of the end-to-end latencies for a packet (shown in equation 1 of Section 13.10) can be lumped together.

19. Consider a reliable pipelined transport protocol that uses cumulative ACKs. The window size is 10. The receiver sends an ACK according to the following rule:

- send an ACK if 10 consecutive packets are received in order thus allowing the sender to advance its sliding window.
- if a packet is received with a sequence number different from the expected next sequence number, then send an ACK corresponding to the highest sequence number received thus far.

(a) 100 packets are sent by the sender. 1 in 10 data packets are lost in transit. Assume no ACK packets are lost. How many ACK packets are actually sent to successfully complete the transmission?

(b) How many ACK packets would have been sent without cumulative ACKs? In this case assume the protocol ACKs every packet individually and if a data packet is received out of order then it re-sends the last ACK packet.