6

THE TRANSPORT LAYER

The transport layer is not just another layer. It is the heart of the whole protocol hierarchy. Its task is to provide reliable, cost-effective data transport from the source machine to the destination machine, independent of the physical network or networks currently in use. Without the transport layer, the whole concept of layered protocols would make little sense. In this chapter we will study the transport layer in detail, including its services, design, protocols, and performance.

6.1. THE TRANSPORT SERVICE

In the following sections we will provide an introduction to the transport service. We look at what kind of service is provided to the application layer (or session layer, if one exists), and especially how one can characterize the quality of service. Then we will look at how applications access the transport service, that is, what the interface is like.

6.1.1. Services Provided to the Upper Layers

The ultimate goal of the transport layer is to provide efficient, reliable, and cost-effective service to its users, normally processes in the application layer. To achieve this goal, the transport layer makes use of the services provided by the network layer. The hardware and/or software within the transport layer that does the work is called the **transport entity**. The transport entity can be in the operating system kernel, in a separate user process, in a library package bound into network applications, or on the network interface card. In some cases, the carrier may even provide reliable transport service, in which case the transport entity lives on special interface machines at the edge of the subnet to which hosts connect. The (logical) relationship of the network, transport, and application layers is illustrated in Fig. 6-1.

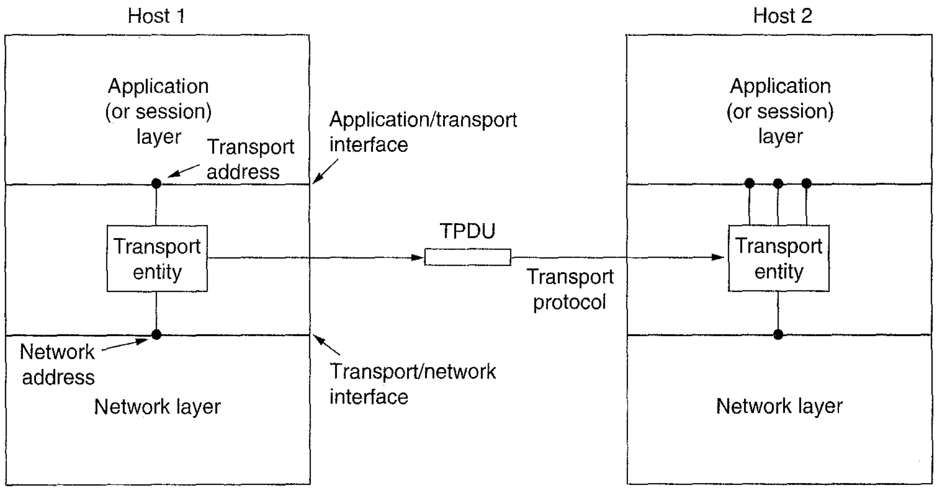


Fig. 6-1. The network, transport, and application layers.

Just as there are two types of network service, connection-oriented and connectionless, there are also the same two types of transport service. The connection-oriented transport service is similar to the connection-oriented network service in many ways. In both cases, connections have three phases: establishment, data transfer, and release. Addressing and flow control are also similar in both layers. Furthermore, the connectionless transport service is also very similar to the connectionless network service.

The obvious question is then: If the transport layer service is so similar to the network layer service, why are there two distinct layers? Why is one layer not adequate? The answer is subtle, but crucial, and goes back to Fig. 1-16. In this figure we can see that the network layer is part of the communication subnet and is run by the carrier (at least for WANs). What happens if the network layer offers connection-oriented service but is unreliable? Suppose that it frequently loses packets? What happens if routers crash from time to time?

Problems occur, that's what. The users have no control over the subnet, so they cannot solve the problem of poor service by using better routers or putting more error handling in the data link layer. The only possibility is to put another layer on top of the network layer that improves the quality of the service. If a transport entity is informed halfway through a long transmission that its network connection has been abruptly terminated, with no indication of what has happened to the data currently in transit, it can set up a new network connection to the remote transport entity. Using this new network connection, it can send a query to its peer asking which data arrived and which did not, and then pick up from where it left off.

In essence, the existence of the transport layer makes it possible for the transport service to be more reliable than the underlying network service. Lost packets and mangled data can be detected and compensated for by the transport layer. Furthermore, the transport service primitives can be designed to be independent of the network service primitives which may vary considerably from network to network (e.g., connectionless LAN service may be quite different than connection-oriented WAN service).

Thanks to the transport layer, it is possible for application programs to be written using a standard set of primitives, and to have these programs work on a wide variety of networks, without having to worry about dealing with different subnet interfaces and unreliable transmission. If all real networks were flawless and all had the same service primitives, the transport layer would probably not be needed. However, in the real world it fulfills the key function of isolating the upper layers from the technology, design, and imperfections of the subnet.

For this reason, many people have made a distinction between layers 1 through 4 on the one hand, and layer(s) above 4 on the other. The bottom four layers can be seen as the **transport service provider**, whereas the upper layer(s) are the **transport service user**. This distinction of provider versus user has a considerable impact on the design of the layers and puts the transport layer in a key position, since it forms the major boundary between the provider and user of the reliable data transmission service.

6.1.2. Quality of Service

Another way of looking at the transport layer is to regard its primary function as enhancing the **QoS** (**Quality of Service**) provided by the network layer. If the network service is impeccable, the transport layer has an easy job. If, however, the network service is poor, the transport layer has to bridge the gap between what the transport users want and what the network layer provides.

While at first glance, quality of service might seem like a vague concept (getting everyone to agree what constitutes "good" service is a nontrivial exercise), QoS can be characterized by a number of specific parameters, as we saw in Chap. 5. The transport service may allow the user to specify preferred, acceptable, and minimum values for various service parameters at the time a connection is set up. Some of the parameters also apply to connectionless transport. It is up to the transport layer to examine these parameters, and depending on the kind of network service or services available to it, determine whether it can provide the required service. In the remainder of this section we will discuss some possible QoS parameters. They are summarized in Fig. 6-2. Note that few networks or protocols provide all of these parameters. Many just try their best to reduce the residual error rate and leave it at that. Others have elaborate QoS architectures (Campbell et al., 1994).

Connection establishment delay
Connection establishment failure probability
Throughput
Transit delay
Residual error ratio
Protection
Priority
Resilience

Fig. 6-2. Typical transport layer quality of service parameters.

The *Connection establishment delay* is the amount of time elapsing between a transport connection being requested and the confirmation being received by the user of the transport service. It includes the processing delay in the remote transport entity. As with all parameters measuring a delay, the shorter the delay, the better the service.

The *Connection establishment failure probability* is the chance of a connection not being established within the maximum establishment delay time, for example, due to network congestion, lack of table space somewhere, or other internal problems.

The *Throughput* parameter measures the number of bytes of user data transferred per second, measured over some time interval. The throughput is measured separately for each direction.

The *Transit delay* measures the time between a message being sent by the transport user on the source machine and its being received by the transport user on the destination machine. As with throughput, each direction is handled separately.

The *Residual error ratio* measures the number of lost or garbled messages as a fraction of the total sent. In theory, the residual error rate should be zero, since it is the job of the transport layer to hide all network layer errors. In practice it may have some (small) finite value.

The *Protection* parameter provides a way for the transport user to specify interest in having the transport layer provide protection against unauthorized third parties (wiretappers) reading or modifying the transmitted data.

The *Priority* parameter provides a way for a transport user to indicate that some of its connections are more important than other ones, and in the event of congestion, to make sure that the high-priority connections get serviced before the low-priority ones.

Finally, the *Resilience* parameter gives the probability of the transport layer itself spontaneously terminating a connection due to internal problems or congestion.

The QoS parameters are specified by the transport user when a connection is requested. Both the desired and minimum acceptable values can be given. In some cases, upon seeing the QoS parameters, the transport layer may immediately realize that some of them are unachievable, in which case it tells the caller that the connection attempt failed, without even bothering to contact the destination. The failure report specifies the reason for the failure.

In other cases, the transport layer knows it cannot achieve the desired goal (e.g., 600 Mbps throughput), but it can achieve a lower, but still acceptable rate (e.g., 150 Mbps). It then sends the lower rate and the minimum acceptable rate to the remote machine, asking to establish a connection. If the remote machine cannot handle the proposed value, but it can handle a value above the minimum, it may make a counteroffer. If it cannot handle any value above the minimum, it rejects the connection attempt. Finally, the originating transport user is informed of whether the connection was established or rejected, and if it was established, the values of the parameters agreed upon.

This process is called **option negotiation**. Once the options have been negotiated, they remain that way throughout the life of the connection. To keep customers from being too greedy, most carriers have the tendency to charge more money for better quality service.

6.1.3. Transport Service Primitives

The transport service primitives allow transport users (e.g., application programs) to access the transport service. Each transport service has its own access primitives. In this section, we will first examine a simple (hypothetical) transport service and then look at a real example.

The transport service is similar to the network service, but there are also some important differences. The main difference is that the network service is intended to model the service offered by real networks, warts and all. Real networks can lose packets, so the network service is generally unreliable.

The (connection-oriented) transport service, in contrast, is reliable. Of course, real networks are not error-free, but that is precisely the purpose of the transport layer—to provide a reliable service on top of an unreliable network.

As an example, consider two processes connected by pipes in UNIX. They assume the connection between them is perfect. They do not want to know about acknowledgements, lost packets, congestion, or anything like that. What they want is a 100 percent reliable connection. Process *A* puts data into one end of the pipe, and process *B* takes it out of the other. This is what the connection-oriented transport service is all about—hiding the imperfections of the network service so that user processes can just assume the existence of an error-free bit stream.

As an aside, the transport layer can also provide unreliable (datagram) service, but there is relatively little to say about that, so we will concentrate on the connection-oriented transport service in this chapter.

A second difference between the network service and transport service is whom the services are intended for. The network service is used only by the transport entities. Few users write their own transport entities, and thus few users or programs ever see the bare network service. In contrast, many programs (and thus programmers) see the transport primitives. Consequently, the transport service must be convenient and easy to use.

To get an idea of what a transport service might be like, consider the five primitives listed in Fig. 6-3. This transport interface is truly bare bones but it gives the essential flavor of what a connection-oriented transport interface has to do. It allows application programs to establish, use, and release connections, which is sufficient for many applications.

Primitive	TPDU sent	Meaning
LISTEN	(none)	Block until some process tries to connect
CONNECT	CONNECTION REQ.	Actively attempt to establish a connection
SEND	DATA	Send information
RECEIVE	(none)	Block until a DATA TPDU arrives
DISCONNECT	DISCONNECTION REQ.	This side wants to release the connection

Fig. 6-3. The primitives for a simple transport service.

To see how these primitives might be used, consider an application with a server and a number of remote clients. To start with, the server executes a LISTEN primitive, typically by calling a library procedure that makes a system call to block the server until a client turns up. When a client wants to talk to the server, it executes a CONNECT primitive. The transport entity carries out this primitive by blocking the caller and sending a packet to the server. Encapsulated in the payload of this packet is a transport layer message for the server's transport entity.

A quick note on terminology is now in order. For lack of a better term, we will reluctantly use the somewhat ungainly acronym **TPDU** (**Transport Protocol Data Unit**) for messages sent from transport entity to transport entity. Thus TPDUs (exchanged by the transport layer) are contained in packets (exchanged by the network layer). In turn, packets are contained in frames (exchanged by the data link layer). When a frame arrives, the data link layer processes the frame header and passes the contents of the frame payload field up to the network entity.

The network entity processes the packet header and passes the contents of the packet payload up to the transport entity. This nesting is illustrated in Fig. 6-4.

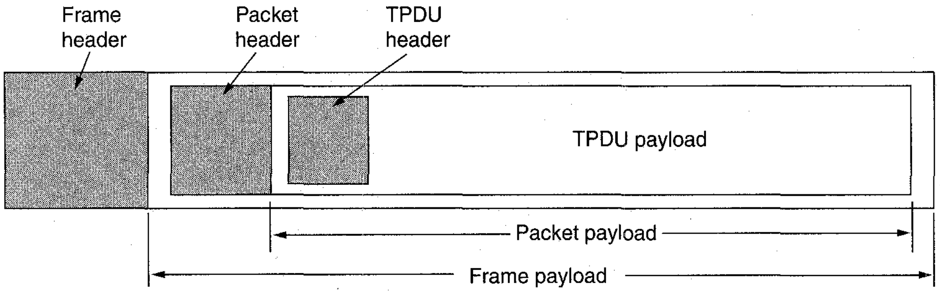


Fig. 6-4. Nesting of TPDUs, packets, and frames.

Getting back to our client-server example, the client's CONNECT call causes a CONNECTION REQUEST TPDU to be sent to the server. When it arrives, the transport entity checks to see that the server is blocked on a LISTEN (i.e., is interested in handling requests). It then unblocks the server and sends a CONNECTION ACCEPTED TPDU back to the client. When this TPDU arrives, the client is unblocked and the connection is established.

Data can now be exchanged using the SEND and RECEIVE primitives. In the simplest form, either party can do a (blocking) RECEIVE to wait for the other party to do a SEND. When the TPDU arrives, the receiver is unblocked. It can then process the TPDU and send a reply. As long as both sides can keep track of whose turn it is to send, this scheme works fine.

Note that at the network layer, even a simple unidirectional data exchange is more complicated than at the transport layer. Every data packet sent will also be acknowledged (eventually). The packets bearing control TPDUs are also acknowledged, implicitly or explicitly. These acknowledgements are managed by the transport entities using the network layer protocol and are not visible to the transport users. Similarly, the transport entities will need to worry about timers and retransmissions. None of this machinery is seen by the transport users. To the transport users, a connection is a reliable bit pipe: one user stuffs bits in and they magically appear at the other end. This ability to hide complexity is the reason that layered protocols are such a powerful tool.

When a connection is no longer needed, it must be released to free up table space within the two transport entities. Disconnection has two variants: asymmetric and symmetric. In the asymmetric variant, either transport user can issue a DISCONNECT primitive, which results in a DISCONNECT TPDU being sent to the remote transport entity. Upon arrival, the connection is released.

In the symmetric variant, each direction is closed separately, independently of the other one. When one side does a DISCONNECT, that means it has no more data to send, but it is still willing to accept data from its partner. In this model, a connection is released when both sides have done a DISCONNECT.

A state diagram for connection establishment and release for these simple primitives is given in Fig. 6-5. Each transition is triggered by some event, either a primitive executed by the local transport user or an incoming packet. For simplicity, we assume here that each TPDU is separately acknowledged. We also assume that a symmetric disconnection model is used, with the client going first. Please note that this model is quite unsophisticated. We will look at more realistic models later on.

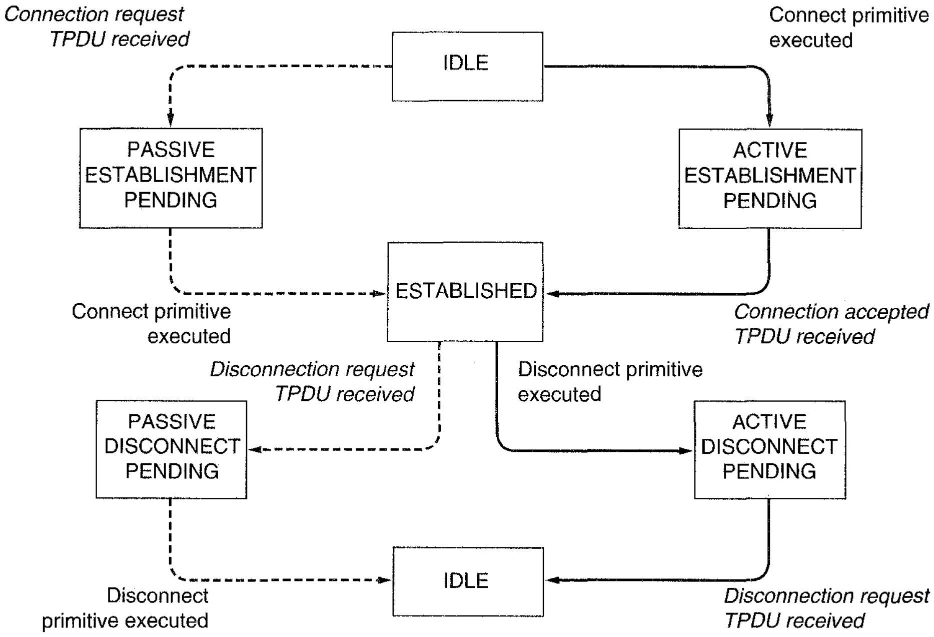

Connection request TPDU received
Connect primitive executed
IDLE
PASSIVE ESTABLISHMENT PENDING
ACTIVE ESTABLISHMENT PENDING
ESTABLISHED
Connect primitive executed
Connection accepted TPDU received
Disconnection request TPDU received
Disconnect primitive executed
PASSIVE DISCONNECT PENDING
ACTIVE DISCONNECT PENDING
IDLE
Disconnect primitive executed
Disconnection request TPDU received

Fig. 6-5. A state diagram for a simple connection management scheme. Transitions labeled in italics are caused by packet arrivals. The solid lines show the client's state sequence. The dashed lines show the server's state sequence.

Berkeley Sockets

Let us now briefly inspect another set of transport primitives, the socket primitives used in Berkeley UNIX for TCP. They are listed in Fig. 6-6. Roughly speaking, they follow the model of our first example but offer more features and flexibility. We will not look at the corresponding TPDUs here. That discussion will have to wait until we study TCP later in this chapter.

The first four primitives in the list are executed in that order by servers. The SOCKET primitive creates a new end point and allocates table space for it within

Primitive	Meaning
SOCKET	Create a new communication end point
BIND	Attach a local address to a socket
LISTEN	Announce willingness to accept connections; give queue size
ACCEPT	Block the caller until a connection attempt arrives
CONNECT	Actively attempt to establish a connection
SEND	Send some data over the connection
RECEIVE	Receive some data from the connection
CLOSE	Release the connection

Fig. 6-6. The socket primitives for TCP.

the transport entity. The parameters of the call specify the addressing format to be used, the type of service desired (e.g., reliable byte stream), and the protocol. A successful SOCKET call returns an ordinary file descriptor for use in succeeding calls, the same way an OPEN call does.

Newly created sockets do not have addresses. These are assigned using the BIND primitive. Once a server has bound an address to a socket, remote clients can connect to it. The reason for not having the SOCKET call create an address directly is that some processes care about their address (e.g., they have been using the same address for years and everyone knows this address), whereas others do not care.

Next comes the LISTEN call, which allocates space to queue incoming calls for the case that several clients try to connect at the same time. In contrast to LISTEN in our first example, in the socket model LISTEN is not a blocking call.

To block waiting for an incoming connection, the server executes an ACCEPT primitive. When a TPDU asking for a connection arrives, the transport entity creates a new socket with the same properties as the original one and returns a file descriptor for it. The server can then fork off a process or thread to handle the connection on the new socket and go back to waiting for the next connection on the original socket.

Now let us look at the client side. Here, too, a socket must first be created using the SOCKET primitive, but BIND is not required since the address used does not matter to the server. The CONNECT primitive blocks the caller and actively starts the connection process. When it completes (i.e., when the appropriate TPDU is received from the server), the client process is unblocked and the connection is established. Both sides can now use SEND and RECEIVE to transmit and receive data over the full-duplex connection.

Connection release with sockets is symmetric. When both sides have executed a CLOSE primitive, the connection is released.

6.2. ELEMENTS OF TRANSPORT PROTOCOLS

The transport service is implemented by a **transport protocol** used between the two transport entities. In some ways, transport protocols resemble the data link protocols we studied in detail in Chap. 3. Both have to deal with error control, sequencing, and flow control, among other issues.

However, significant differences between the two also exist. These differences are due to major dissimilarities between the environments in which the two protocols operate, as shown in Fig. 6-7. At the data link layer, two routers communicate directly via a physical channel, whereas at the transport layer, this physical channel is replaced by the entire subnet. This difference has many important implications for the protocols.

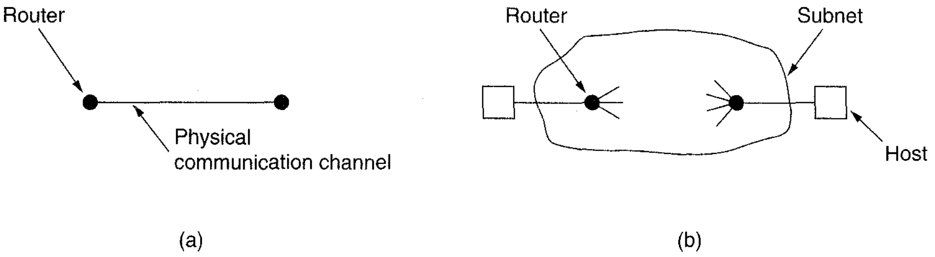


Fig. 6-7. (a) Environment of the data link layer. (b) Environment of the transport layer.

For one thing, in the data link layer, it is not necessary for a router to specify which router it wants to talk to—each outgoing line uniquely specifies a particular router. In the transport layer, explicit addressing of destinations is required.

For another thing, the process of establishing a connection over the wire of Fig. 6-7(a) is simple: the other end is always there (unless it has crashed, in which case it is not there). Either way, there is not much to do. In the transport layer, initial connection establishment is more complicated, as we will see.

Another, exceedingly annoying, difference between the data link layer and the transport layer is the potential existence of storage capacity in the subnet. When a router sends a frame, it may arrive or be lost, but it cannot bounce around for a while, go into hiding in a far corner of the world, and then suddenly emerge at an inopportune moment 30 sec later. If the subnet uses datagrams and adaptive routing inside, there is a nonnegligible probability that a packet may be stored for a number of seconds and then delivered later. The consequences of this ability of the subnet to store packets can sometimes be disastrous and require the use of special protocols.

A final difference between the data link and transport layers is one of amount rather than of kind. Buffering and flow control are needed in both layers, but the presence of a large and dynamically varying number of connections in the transport layer may require a different approach than we used in the data link layer. In Chap. 3, some of the protocols allocate a fixed number of buffers to each line, so that when a frame arrives there is always a buffer available. In the transport layer, the larger number of connections that must be managed make the idea of dedicating many buffers to each one less attractive. In the following sections, we will examine all of these important issues and others.

6.2.1. Addressing

When an application process wishes to set up a connection to a remote application process, it must specify which one to connect to. (Connectionless transport has the same problem: To whom should each message be sent?) The method normally used is to define transport addresses to which processes can listen for connection requests. In the Internet, these end points are (IP address, local port) pairs. In ATM networks, they are AAL-SAPs. We will use the neutral term **TSAP (Transport Service Access Point)**. The analogous end points in the network layer (i.e., network layer addresses) are then called **NSAPs**. IP addresses are examples of NSAPs.

Figure 6-8 illustrates the relationship between the NSAP, TSAP, network connection, and transport connection for a connection-oriented subnet (e.g., ATM). Note that a transport entity normally supports multiple TSAPs. On some networks, multiple NSAPs also exist, but on others each machine has only one NSAP (e.g., one IP address). A possible connection scenario for a transport connection over a connection-oriented network layer is as follows.

1. A time-of-day server process on host 2 attaches itself to TSAP 122 to wait for an incoming call. How a process attaches itself to a TSAP is outside the networking model and depends entirely on the local operating system. A call such as our LISTEN might be used, for example.

2. An application process on host 1 wants to find out the time-of-day, so it issues a CONNECT request specifying TSAP 6 as the source and TSAP 122 as the destination.

3. The transport entity on host 1 selects a network address on its machine (if it has more than one) and sets up a network connection between them. (With a connectionless subnet, establishing this network layer connection would not be done.) Using this network connection, host 1's transport entity can talk to the transport entity on host 2.

4. The first thing the transport entity on 1 says to its peer on 2 is: "Good morning. I would like to establish a transport connection between my TSAP 6 and your TSAP 122. What do you say?"

5. The transport entity on 2 then asks the time-of-day server at TSAP 122 if it is willing to accept a new connection. If it agrees, the transport connection is established.

Note that the transport connection goes from TSAP to TSAP, whereas the network connection only goes part way, from NSAP to NSAP.

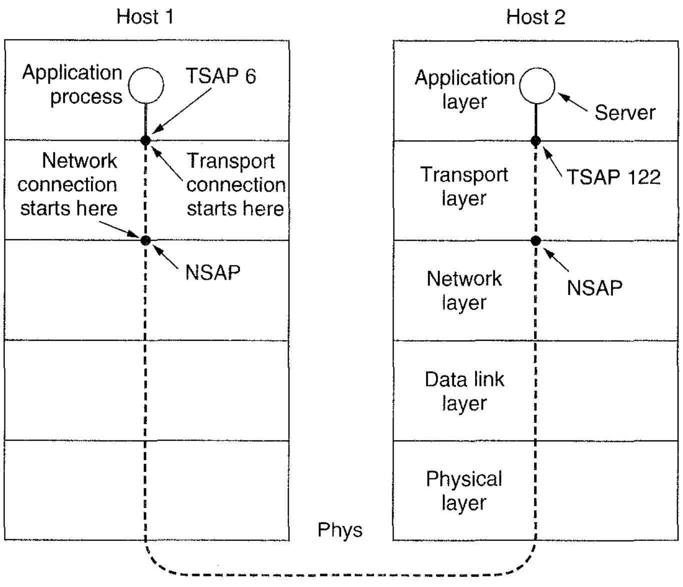


Fig. 6-8. TSAPs, NSAPs, and connections.

The picture painted above is fine, except we have swept one little problem under the rug: How does the user process on host 1 know that the time-of-day server is attached to TSAP 122? One possibility is that the time-of-day server has been attaching itself to TSAP 122 for years, and gradually all the network users have learned this. In this model, services have stable TSAP addresses which can be printed on paper and given to new users when they join the network.

While stable TSAP addresses might work for a small number of key services that never change, in general, user processes often want to talk to other user processes that only exist for a short time and do not have a TSAP address that is known in advance. Furthermore, if there are potentially many server processes, most of which are rarely used, it is wasteful to have each of them active and listening to a stable TSAP address all day long. In short, a better scheme is needed.

One such scheme, used by UNIX hosts on the Internet, is shown in Fig. 6-9 in a simplified form. It is known as the **initial connection protocol**. Instead of every conceivable server listening at a well-known TSAP, each machine that wishes to offer service to remote users has a special **process server** that acts as a proxy for less-heavily used servers. It listens to a set of ports at the same time, waiting for a TCP connection request. Potential users of a service begin by doing a CONNECT request, specifying the TSAP address (TCP port) of the service they want. If no server is waiting for them, they get a connection to the process server, as shown in Fig. 6-9(a).

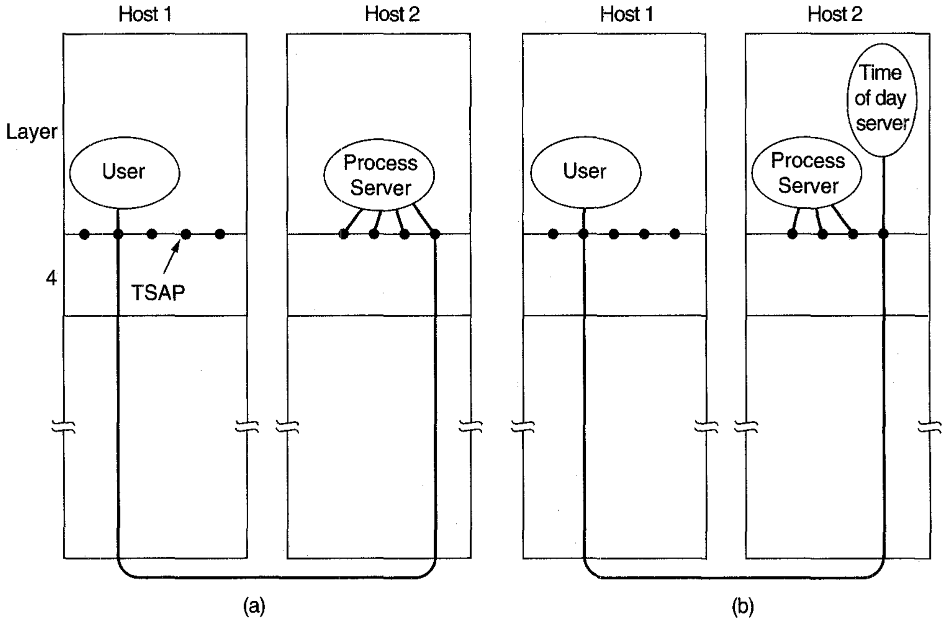


Fig. 6-9. How a user process in host 1 establishes a connection with a time-of-day server in host 2.

After it gets the incoming request, the process server spawns off the requested server, allowing it to inherit the existing connection with the user. The new server then does the requested work, while the process server goes back to listening for new requests, as shown in Fig. 6-9(b).

While the initial connection protocol works fine for those servers that can be created as they are needed, there are many situations in which services do exist independently of the process server. A file server, for example, needs to run on special hardware (a machine with a disk) and cannot just be created on-the-fly when someone wants to talk to it.

To handle this situation, an alternative scheme is often used. In this model, there exists a special process called a **name server** or sometimes a **directory server**. To find the TSAP address corresponding to a given service name, such as "time-of-day," a user sets up a connection to the name server (which listens to a well-known TSAP). The user then sends a message specifying the service name, and the name server sends back the TSAP address. Then the user releases the connection with the name server and establishes a new one with the desired service.

In this model, when a new service is created, it must register itself with the name server, giving both its service name (typically an ASCII string) and the address of its TSAP. The name server records this information in its internal database, so that when queries come in later, it will know the answers.

The function of the name server is analogous to the directory assistance operator in the telephone system—it provides a mapping of names onto numbers. Just as in the telephone system, it is essential that the address of the well-known TSAP used by the name server (or the process server in the initial connection protocol) is indeed well known. If you do not know the number of the information operator, you cannot call the information operator to find it out. If you think the number you dial for information is obvious, try it in a foreign country some time.

Now let us suppose that the user has successfully located the address of the TSAP to be connected to. Another interesting question is how does the local transport entity know on which machine that TSAP is located? More specifically, how does the transport entity know which network layer address to use to set up a network connection to the remote transport entity that manages the TSAP requested?

The answer depends on the structure of TSAP addresses. One possible structure is that TSAP addresses are **hierarchical addresses**. With hierarchical addresses, the address consists of a sequence of fields used to disjointly partition the address space. For example, a truly universal TSAP address might have the following structure:

address = <galaxy> <star> <planet> <country> <network> <host> <port>

With this scheme, it is straightforward to locate a TSAP anywhere in the known universe. Equivalently, if a TSAP address is a concatenation of an NSAP address and a port (a local identifier specifying one of the local TSAPs), then when a transport entity is given a TSAP address to connect to, it uses the NSAP address contained in the TSAP address to reach the proper remote transport entity.

As a simple example of a hierarchical address, consider the telephone number 19076543210. This number can be parsed as 1-907-654-3210, where 1 is a country code (United States + Canada), 907 is an area code (Alaska), 654 is an end office in Alaska, and 3210 is one of the "ports" (subscriber lines) in that end office.

The alternative to a hierarchical address space is a **flat address space**. If the TSAP addresses are not hierarchical, a second level of mapping is needed to locate the proper machine. There would have to be a name server that took transport addresses as input and returned network addresses as output. Alternatively, in some situations (e.g., on a LAN), it is possible to broadcast a query asking the destination machine to please identify itself by sending a packet.

6.2.2. Establishing a Connection

Establishing a connection sounds easy, but it is actually surprisingly tricky. At first glance, it would seem sufficient for one transport entity to just send a CONNECTION REQUEST TPDU to the destination and wait for a CONNECTION ACCEPTED reply. The problem occurs when the network can lose, store, and duplicate packets.

Imagine a subnet that is so congested that acknowledgements hardly ever get back in time, and each packet times out and is retransmitted two or three times. Suppose that the subnet uses datagrams inside, and every packet follows a different route. Some of the packets might get stuck in a traffic jam inside the subnet and take a long time to arrive, that is, they are stored in the subnet and pop out much later.

The worst possible nightmare is as follows. A user establishes a connection with a bank, sends messages telling the bank to transfer a large amount of money to the account of a not-entirely-trustworthy person, and then releases the connection. Unfortunately, each packet in the scenario is duplicated and stored in the subnet. After the connection has been released, all the packets pop out of the subnet and arrive at the destination in order, asking the bank to establish a new connection, transfer money (again), and release the connection. The bank has no way of telling that these are duplicates. It must assume that this is a second, independent transaction, and transfers the money again. For the remainder of this section we will study the problem of delayed duplicates, with special emphasis on algorithms for establishing connections in a reliable way, so that nightmares like the one above cannot happen.

The crux of the problem is the existence of delayed duplicates. It can be attacked in various ways, none of them very satisfactory. One way is to use throwaway transport addresses. In this approach, each time a transport address is needed, a new one is generated. When a connection is released, the address is discarded. This strategy makes the process server model of Fig. 6-9 impossible.

Another possibility is to give each connection a connection identifier (i.e., a sequence number incremented for each connection established), chosen by the initiating party, and put in each TPDU, including the one requesting the connection. After each connection is released, each transport entity could update a table listing obsolete connections as (peer transport entity, connection identifier) pairs. Whenever a connection request came in, it could be checked against the table, to see if it belonged to a previously released connection.

Unfortunately, this scheme has a basic flaw: it requires each transport entity to maintain a certain amount of history information indefinitely. If a machine crashes and loses its memory, it will no longer know which connection identifiers have already been used.

Instead, we need to take a different tack. Rather than allowing packets to live forever within the subnet, we must devise a mechanism to kill off aged packets that are still wandering about. If we can ensure that no packet lives longer than some known time, the problem becomes somewhat more manageable.

Packet lifetime can be restricted to a known maximum using one of the following techniques:

1. Restricted subnet design.

2. Putting a hop counter in each packet.

3. Timestamping each packet.

The first method includes any method that prevents packets from looping, combined with some way of bounding congestion delay over the (now known) longest possible path. The second method consists of having the hop count incremented each time the packet is forwarded. The data link protocol simply discards any packet whose hop counter has exceeded a certain value. The third method requires each packet to bear the time it was created, with the routers agreeing to discard any packet older than some agreed upon time. This latter method requires the router clocks to be synchronized, which itself is a nontrivial task unless synchronization is achieved external to the network, for example by listening to WWV or some other radio station that broadcasts the precise time periodically.

In practice, we will need to guarantee not only that a packet is dead, but also that all acknowledgements to it are also dead, so we will now introduce *T*, which is some small multiple of the true maximum packet lifetime. The multiple is protocol-dependent and simply has the effect of making *T* longer. If we wait a time *T* after a packet has been sent, we can be sure that all traces of it are now gone and that neither it nor its acknowledgements will suddenly appear out of the blue to complicate matters.

With packet lifetimes bounded, it is possible to devise a foolproof way to establish connections safely. The method described below is due to Tomlinson (1975). It solves the problem but introduces some peculiarities of its own. The method was further refined by Sunshine and Dalal (1978). Variants of it are widely used in practice.

To get around the problem of a machine losing all memory of where it was after a crash, Tomlinson proposed equipping each host with a time-of-day clock. The clocks at different hosts need not be synchronized. Each clock is assumed to take the form of a binary counter that increments itself at uniform intervals. Furthermore, the number of bits in the counter must equal or exceed the number of bits in the sequence numbers. Last, and most important, the clock is assumed to continue running even if the host goes down.

The basic idea is to ensure that two identically numbered TPDUs are never outstanding at the same time. When a connection is set up, the low-order *k* bits of the clock are used as the initial sequence number (also *k* bits). Thus, unlike our protocols of Chap. 3, each connection starts numbering its TPDUs with a different sequence number. The sequence space should be so large that by the time sequence numbers wrap around, old TPDUs with the same sequence number are long gone. This linear relation between time and initial sequence numbers is shown in Fig. 6-10.

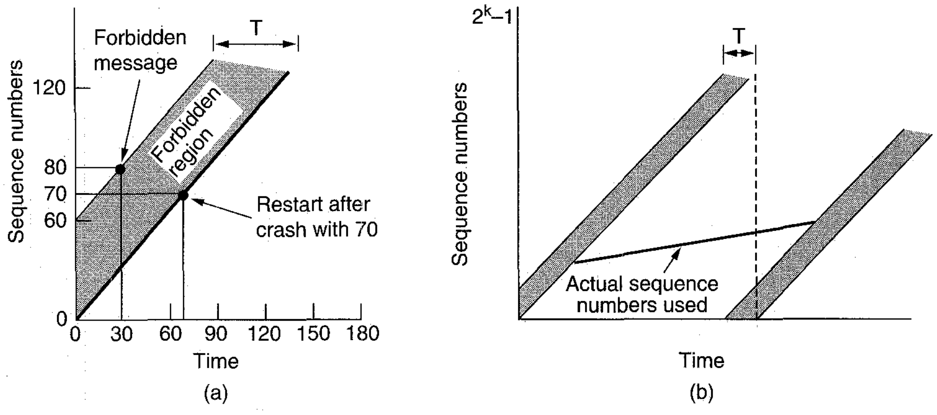


Fig. 6-10. (a) TPDUs may not enter the forbidden region. (b) The resynchronization problem.

Once both transport entities have agreed on the initial sequence number, any sliding window protocol can be used for data flow control. In reality, the initial sequence number curve (shown by the heavy line) is not really linear, but a staircase, since the clock advances in discrete steps. For simplicity we will ignore this detail.

A problem occurs when a host crashes. When it comes up again, its transport entity does not know where it was in the sequence space. One solution is to require transport entities to be idle for T sec after a recovery to let all old TPDUs die off. However, in a complex internetwork, T may be large, so this strategy is unattractive.

To avoid requiring T sec of dead time after a crash, it is necessary to introduce a new restriction on the use of sequence numbers. We can best see the need for this restriction by means of an example. Let T, the maximum packet lifetime, be 60 sec and let the clock tick once per second. As shown in Fig. 6-10, the initial sequence number for a connection opened at time x will be x. Imagine that at $t = 30$ sec, an ordinary data TPDU being sent on (a previously opened) connection 5 is given sequence number 80. Call this TPDU X. Immediately after sending TPDU X, the host crashes and then quickly restarts. At $t = 60$, it begins reopening connections 0 through 4. At $t = 70$, it reopens connection 5, using initial sequence number 70 as required. Within the next 15 sec it sends data TPDUs 70 through 80. Thus at $t = 85$, a new TPDU with sequence number 80 and connection 5 has been injected into the subnet. Unfortunately, TPDU X still exists. If it should arrive at the receiver before the new TPDU 80, TPDU *X* will be accepted and the correct TPDU 80 will be rejected as a duplicate.

To prevent such problems, we must prevent sequence numbers from being used (i.e., assigned to new TPDUs) for a time *T* before their potential use as initial sequence numbers. The illegal combinations of time and sequence number are shown as the **forbidden region** in Fig. 6-10(a). Before sending any TPDU on any connection, the transport entity must read the clock and check to see that it is not in the forbidden region.

The protocol can get itself into trouble in two different ways. If a host sends too much data too fast on a newly opened connection, the actual sequence number versus time curve may rise more steeply than the initial sequence number versus time curve. This means that the maximum data rate on any connection is one TPDU per clock tick. It also means that the transport entity must wait until the clock ticks before opening a new connection after a crash restart, lest the same number be used twice. Both of these points argue for a short clock tick (a few milliseconds).

Unfortunately, entering the forbidden region from underneath by sending too fast is not the only way to get into trouble. From Fig. 6-10(b), it should be clear that at any data rate less than the clock rate, the curve of actual sequence numbers used versus time will eventually run into the forbidden region from the left. The greater the slope of the actual sequence number curve, the longer this event will be delayed. As we stated above, just before sending every TPDU, the transport entity must check to see if it is about to enter the forbidden region, and if so, either delay the TPDU for *T* sec or resynchronize the sequence numbers.

The clock-based method solves the delayed duplicate problem for data TPDUs, but for this method to be useful, a connection must first be established. Since control TPDUs may also be delayed, there is a potential problem in getting both sides to agree on the initial sequence number. Suppose, for example, that connections are established by having host 1 send a CONNECTION REQUEST TPDU containing the proposed initial sequence number and destination port number to a remote peer, host 2. The receiver, host 2, then acknowledges this request by sending a CONNECTION ACCEPTED TPDU back. If the CONNECTION REQUEST TPDU is lost but a delayed duplicate CONNECTION REQUEST suddenly shows up at host 2, the connection will be established incorrectly.

To solve this problem, Tomlinson (1975) introduced the **three-way handshake**. This establishment protocol does not require both sides to begin sending with the same sequence number, so it can be used with synchronization methods other than the global clock method. The normal setup procedure when host 1 initiates is shown in Fig. 6-11(a). Host 1 chooses a sequence number, *x*, and sends a CONNECTION REQUEST TPDU containing it to host 2. Host 2 replies with a CONNECTION ACCEPTED TPDU acknowledging *x* and announcing its own initial sequence number, *y*. Finally, host 1 acknowledges host 2's choice of an initial sequence number in the first data TPDU that it sends.

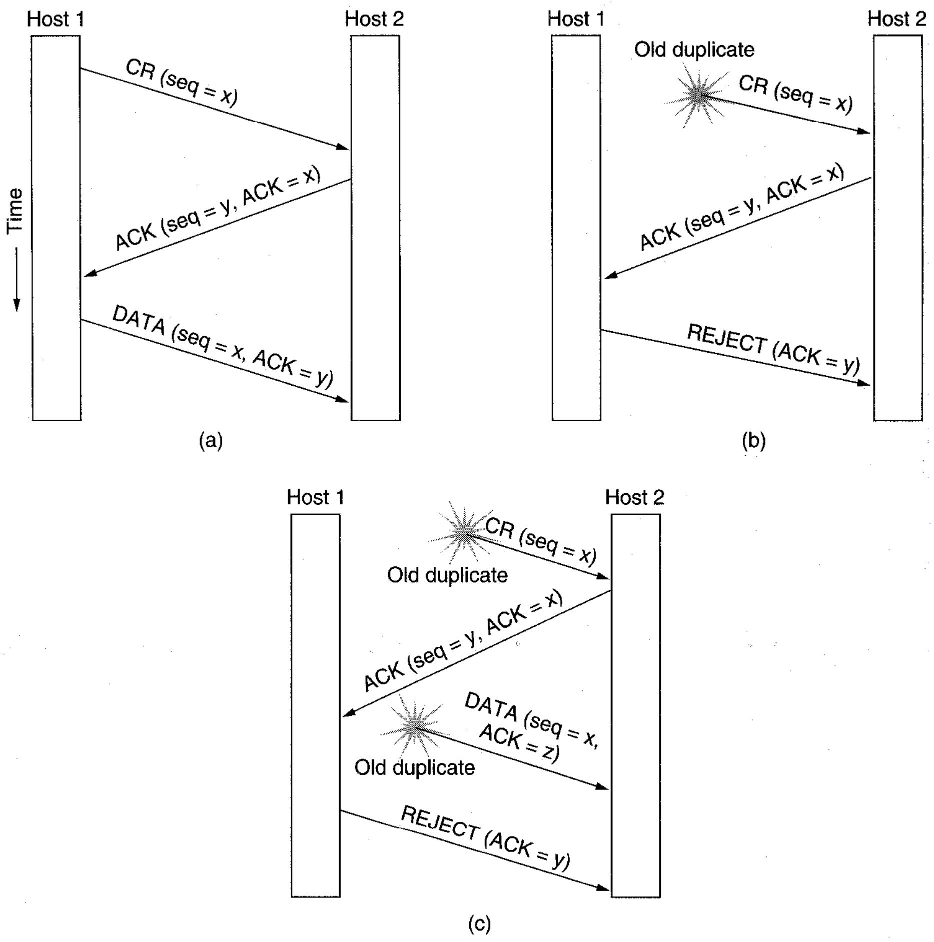


Fig. 6-11. Three protocol scenarios for establishing a connection using a three-way handshake. CR and ACC denote CONNECTION REQUEST and CONNECTION ACCEPTED, respectively. (a) Normal operation. (b) Old duplicate CONNECTION REQUEST appearing out of nowhere. (c) Duplicate CONNECTION REQUEST and duplicate ACK.

Now let us see how the three-way handshake works in the presence of delayed duplicate control TPDUs. In Fig. 6-12(b), the first TPDU is a delayed duplicate CONNECTION REQUEST from an old connection. This TPDU arrives at host 2 without host 1's knowledge. Host 2 reacts to this TPDU by sending host 1 a CONNECTION ACCEPTED TPDU, in effect asking for verification that host 1 was indeed trying to set up a new connection. When host 1 rejects host 2's attempt to establish, host 2 realizes that it was tricked by a delayed duplicate and abandons the connection. In this way, a delayed duplicate does no damage.

The worst case is when both a delayed CONNECTION REQUEST and an acknowledgement to a CONNECTION ACCEPTED are floating around in the subnet. This case is shown in Fig. 6-11(c). As in the previous example, host 2 gets a delayed CONNECTION REQUEST and replies to it. At this point it is crucial to realize that host 2 has proposed using *y* as the initial sequence number for host 2 to host 1 traffic, knowing full well that no TPDUs containing sequence number *y* or acknowledgements to *y* are still in existence. When the second delayed TPDU arrives at host 2, the fact that *z* has been acknowledged rather than *y* tells host 2 that this, too, is an old duplicate. The important thing to realize here is that there is no combination of old CONNECTION REQUEST, CONNECTION ACCEPTED, or other TPDUs that can cause the protocol to fail and have a connection set up by accident when no one wants it.

An alternative scheme for establishing connections reliably in the face of delayed duplicates is described in (Watson, 1981). It uses multiple timers to exclude undesired events.

6.2.3. Releasing a Connection

Releasing a connection is easier than establishing one. Nevertheless, there are more pitfalls than one might expect. As we mentioned earlier, there are two styles of terminating a connection: asymmetric release and symmetric release. Asymmetric release is the way the telephone system works: when one party hangs up, the connection is broken. Symmetric release treats the connection as two separate unidirectional connections and requires each one to be released separately.

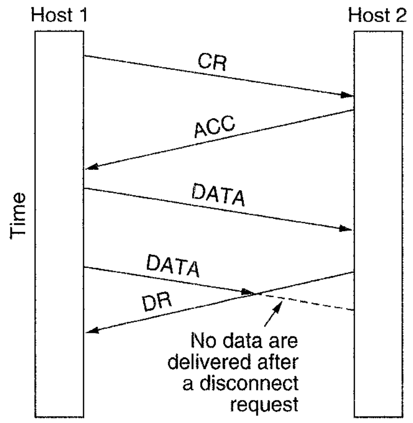


Fig. 6-12. Abrupt disconnection with loss of data.

Asymmetric release is abrupt and may result in data loss. Consider the scenario of Fig. 6-12. After the connection is established, host 1 sends a TPDU that arrives properly at host 2. Then host 1 sends another TPDU. Unfortunately, host 2 issues a DISCONNECT before the second TPDU arrives. The result is that the connection is released and data are lost.

Clearly, a more sophisticated release protocol is required to avoid data loss. One way is to use symmetric release, in which each direction is released independently of the other one. Here, a host can continue to receive data even after it has sent a DISCONNECT TPDU.

Symmetric release does the job when each process has a fixed amount of data to send and clearly knows when it has sent it. In other situations, determining that all the work has been done and the connection should be terminated is not so obvious. One can envision a protocol in which host 1 says: "I am done. Are you done too?" If host 2 responds: "I am done too. Goodbye." the connection can be safely released.

Unfortunately, this protocol does not always work. There is a famous problem that deals with this issue. It is called the **two-army problem**. Imagine that a white army is encamped in a valley, as shown in Fig. 6-13. On both of the surrounding hillsides are blue armies. The white army is larger than either of the blue armies alone, but together they are larger than the white army. If either blue army attacks by itself, it will be defeated, but if the two blue armies attack simultaneously, they will be victorious.

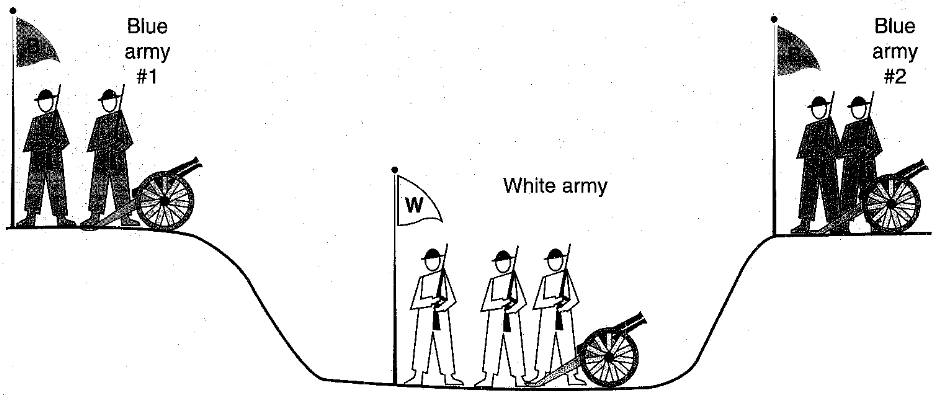


Fig. 6-13. The two-army problem.

The blue armies want to synchronize their attacks. However, their only communication medium is to send messengers on foot down into the valley, where they might be captured and the message lost (i.e., they have to use an unreliable communication channel). The question is: Does a protocol exist that allows the blue armies to win?

Suppose that the commander of blue army #1 sends a message reading: "I propose we attack at dawn on March 29. How about it?" Now suppose that the message arrives, and the commander of blue army #2 agrees, and that his reply gets safely back to blue army #1. Will the attack happen? Probably not, because commander #2 does not know if his reply got through. If it did not, blue army #1 will not attack, so it would be foolish for him to charge into battle.

Now let us improve the protocol by making it a three-way handshake. The initiator of the original proposal must acknowledge the response. Assuming no messages are lost, blue army #2 will get the acknowledgement, but the commander of blue army #1 will now hesitate. After all, he does not know if his acknowledgement got through, and if it did not, he knows that blue army #2 will not attack. We could now make a four-way handshake protocol, but that does not help either.

In fact, it can be proven that no protocol exists that works. Suppose that some protocol did exist. Either the last message of the protocol is essential or it is not. If it is not, remove it (and any other unessential messages) until we are left with a protocol in which every message is essential. What happens if the final message does not get through? We just said that it was essential, so if it is lost, the attack does not take place. Since the sender of the final message can never be sure of its arrival, he will not risk attacking. Worse yet, the other blue army knows this, so it will not attack either.

To see the relevance of the two-army problem to releasing connections, just substitute "disconnect" for "attack." If neither side is prepared to disconnect until it is convinced that the other side is prepared to disconnect too, the disconnection will never happen.

In practice, one is usually prepared to take more risks when releasing connections than when attacking white armies, so the situation is not entirely hopeless. Figure 6-14 illustrates four scenarios of releasing using a three-way handshake. While this protocol is not infallible, it is usually adequate.

In Fig. 6-14(a), we see the normal case in which one of the users sends a DR (DISCONNECTION REQUEST) TPDU in order to initiate the connection release. When it arrives, the recipient sends back a DR TPDU, too, and starts a timer, just in case its DR is lost. When this DR arrives, the original sender sends back an ACK TPDU and releases the connection. Finally, when the ACK TPDU arrives, the receiver also releases the connection. Releasing a connection means that the transport entity removes the information about the connection from its table of open connections and signals the connection's owner (the transport user) somehow. This action is different from a transport user issuing a DISCONNECT primitive.

If the final ACK TPDU is lost, as shown in Fig. 6-14(b), the situation is saved by the timer. When the timer expires, the connection is released anyway.

Now consider the case of the second DR being lost. The user initiating the disconnection will not receive the expected response, will time out, and will start all over again. In Fig. 6-14(c) we see how this works, assuming that the second time no TPDUs are lost and all TPDUs are delivered correctly and on time.

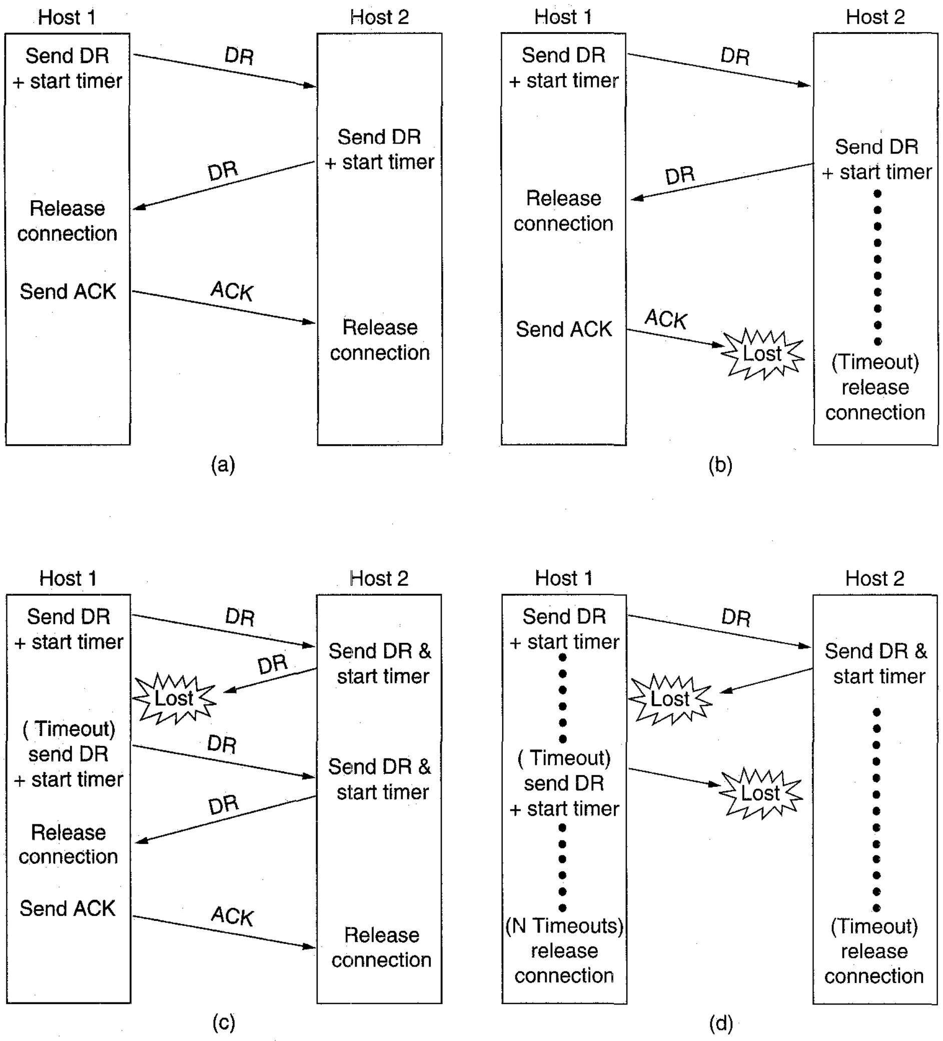


Fig. 6-14. Four protocol scenarios for releasing a connection. (a) Normal case of three-way handshake. (b) Final ACK lost. (c) Response lost. (d) Response lost and subsequent DRs lost.

Our last scenario, Fig. 6-14(d), is the same as Fig. 6-14(c) except that now we assume all the repeated attempts to retransmit the DR also fail due to lost TPDUs. After *N* retries, the sender just gives up and releases the connection. Meanwhile, the receiver times out and also exits.

While this protocol usually suffices, in theory it can fail if the initial DR and *N* retransmissions are all lost. The sender will give up and release the connection, while the other side knows nothing at all about the attempts to disconnect and is still fully active. This situation results in a half-open connection.

We could have avoided this problem by not allowing the sender to give up after *N* retries but forcing it to go on forever until it gets a response. However, if the other side is allowed to time out, then the sender will indeed go on forever, because no response will ever be forthcoming. If we do not allow the receiving side to time out, then the protocol hangs in Fig. 6-14(b).

One way to kill off half-open connections is to have a rule saying that if no TPDUs have arrived for a certain number of seconds, the connection is automatically disconnected. That way, if one side ever disconnects, the other side will detect the lack of activity and also disconnect. Of course, if this rule is introduced, it is necessary for each transport entity to have a timer that is stopped and then restarted whenever a TPDU is sent. If this timer expires, a dummy TPDU is transmitted, just to keep the other side from disconnecting. On the other hand, if the automatic disconnect rule is used and too many dummy TPDUs in a row are lost on an otherwise idle connection, first one side, then the other side will automatically disconnect.

We will not belabor this point any more, but by now it should be clear that releasing a connection is not nearly as simple as it at first appears.

6.2.4. Flow Control and Buffering

Having examined connection establishment and release in some detail, let us now look at how connections are managed while they are in use. One of the key issues has come up before: flow control. In some ways the flow control problem in the transport layer is the same as in the data link layer, but in other ways it is different. The basic similarity is that in both layers a sliding window or other scheme is needed on each connection to keep a fast transmitter from overrunning a slow receiver. The main difference is that a router usually has relatively few lines whereas a host may have numerous connections. This difference makes it impractical to implement the data link buffering strategy in the transport layer.

In the data link protocols of Chap. 3, frames were buffered at both the sending router and at the receiving router. In protocol 6, for example, both sender and receiver are required to dedicate *MaxSeq* + 1 buffers to each line, half for input and half for output. For a host with a maximum of, say, 64 connections, and a 4-bit sequence number, this protocol would require 1024 buffers.

In the data link layer, the sending side must buffer outgoing frames because they might have to be retransmitted. If the subnet provides datagram service, the sending transport entity must also buffer, and for the same reason. If the receiver knows that the sender buffers all TPDUs until they are acknowledged, the receiver may or may not dedicate specific buffers to specific connections, as it sees fit. The receiver may, for example, maintain a single buffer pool shared by all connections. When a TPDU comes in, an attempt is made to dynamically acquire a new buffer. If one is available, the TPDU is accepted; otherwise, it is discarded. Since the sender is prepared to retransmit TPDUs lost by the subnet, no harm is done by having the receiver drop TPDUs, although some resources are wasted. The sender just keeps trying until it gets an acknowledgement.

In summary, if the network service is unreliable, the sender must buffer all TPDUs sent, just as in the data link layer. However, with reliable network service, other trade-offs become possible. In particular, if the sender knows that the receiver always has buffer space, it need not retain copies of the TPDUs it sends. However, if the receiver cannot guarantee that every incoming TPDU will be accepted, the sender will have to buffer anyway. In the latter case, the sender cannot trust the network layer's acknowledgement, because the acknowledgement means only that the TPDU arrived, not that it was accepted. We will come back to this important point later.

Even if the receiver has agreed to do the buffering, there still remains the question of the buffer size. If most TPDUs are nearly the same size, it is natural to organize the buffers as a pool of identical size buffers, with one TPDU per buffer, as in Fig. 6-15(a). However, if there is wide variation in TPDU size, from a few characters typed at a terminal to thousands of characters from file transfers, a pool of fixed-sized buffers presents problems. If the buffer size is chosen equal to the largest possible TPDU, space will be wasted whenever a short TPDU arrives. If the buffer size is chosen less than the maximum TPDU size, multiple buffers will be needed for long TPDUs, with the attendant complexity.

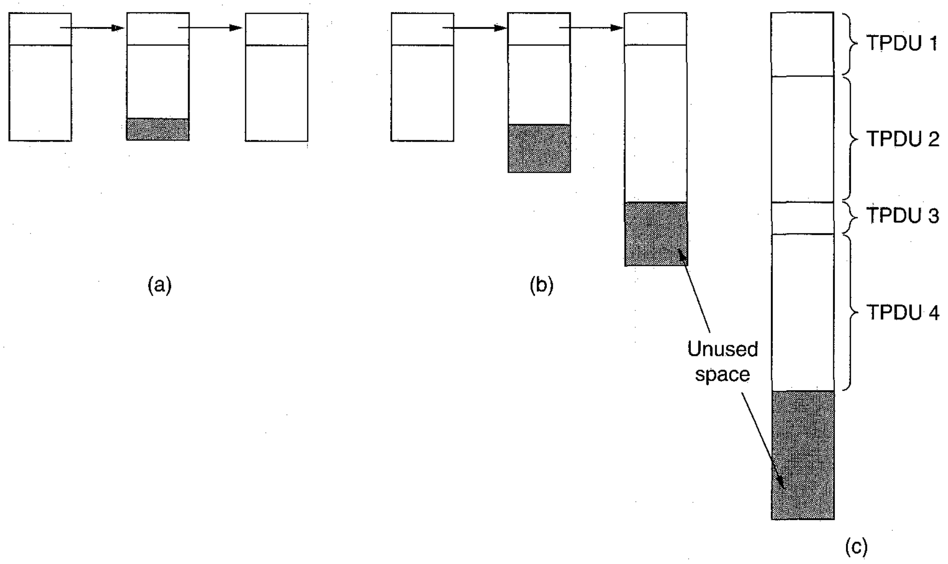


Fig. 6-15. (a) Chained fixed-size buffers. (b) Chained variable-size buffers. (c) One large circular buffer per connection.

Another approach to the buffer size problem is to use variable-size buffers, as in Fig. 6-15(b). The advantage here is better memory utilization, at the price of more complicated buffer management. A third possibility is to dedicate a single large circular buffer per connection, as in Fig. 6-15(c). This system also makes good use of memory, provided that all connections are heavily loaded but is poor if some connections are lightly loaded.

The optimum trade-off between source buffering and destination buffering depends on the type of traffic carried by the connection. For low-bandwidth bursty traffic, such as that produced by an interactive terminal, it is better not to dedicate any buffers, but rather to acquire them dynamically at both ends. Since the sender cannot be sure the receiver will be able to acquire a buffer, the sender must retain a copy of the TPDU until it is acknowledged. On the other hand, for file transfer and other high-bandwidth traffic, it is better if the receiver does dedicate a full window of buffers, to allow the data to flow at maximum speed. Thus for low-bandwidth bursty traffic, it is better to buffer at the sender, and for high-bandwidth, smooth traffic, it is better to buffer at the receiver.

As connections are opened and closed, and as the traffic pattern changes, the sender and receiver need to dynamically adjust their buffer allocations. Consequently, the transport protocol should allow a sending host to request buffer space at the other end. Buffers could be allocated per connection, or collectively, for all the connections running between the two hosts. Alternatively, the receiver, knowing its buffer situation (but not knowing the offered traffic) could tell the sender "I have reserved *X* buffers for you." If the number of open connections should increase, it may be necessary for an allocation to be reduced, so the protocol should provide for this possibility.

A reasonably general way to manage dynamic buffer allocation is to decouple the buffering from the acknowledgements, in contrast to the sliding window protocols of Chap. 3. Dynamic buffer management means, in effect, a variable-sized window. Initially, the sender requests a certain number of buffers, based on its perceived needs. The receiver then grants as many of these as it can afford. Every time the sender transmits a TPDU, it must decrement its allocation, stopping altogether when the allocation reaches zero. The receiver then separately piggybacks both acknowledgements and buffer allocations onto the reverse traffic.

Figure 6-16 shows an example of how dynamic window management might work in a datagram subnet with 4-bit sequence numbers. Assume that buffer allocation information travels in separate TPDUs, as shown, and is not piggybacked onto reverse traffic. Initially, *A* wants eight buffers, but is granted only four of these. It then sends three TPDUs, of which the third is lost. TPDU 6 acknowledges receipt of all TPDUs up to and including sequence number 1, thus allowing *A* to release those buffers, and furthermore informs *A* that it has permission to send three more TPDUs starting beyond 1 (i.e., TPDUs 2, 3, and 4). *A* knows that it has already sent number 2, so it thinks that it may send TPDUs 3 and 4, which it proceeds to do. At this point it is blocked and must wait for more buffer allocation. Timeout induced retransmissions (line 9), however, may occur while blocked, since they use buffers that have already been allocated. In line 10, *B* acknowledges receipt of all TPDUs up to and including 4, but refuses to let *A* continue. Such a situation is impossible with the fixed window protocols of Chap. 3. The next TPDU from *B* to *A* allocates another buffer and allows *A* to continue.

	A	Message	B	Comments
1	→	< request 8 buffers>	→	A wants 8 buffers
2	←	<ack = 15, buf = 4>	←	B grants messages 0-3 only
3	→	<seq = 0, data = m0>	→	A has 3 buffers left now
4	→	<seq = 1, data = m1>	→	A has 2 buffers left now
5	→	<seq = 2, data = m2>	•••	Message lost but A thinks it has 1 left
6	←	<ack = 1, buf = 3>	←	B acknowledges 0 and 1, permits 2-4
7	→	<seq = 3, data = m3>	→	A has buffer left
8	→	<seq = 4, data = m4>	→	A has 0 buffers left, and must stop
9	→	<seq = 2, data = m2>	→	A times out and retransmits
10	←	<ack = 4, buf = 0>	←	Everything acknowledged, but A still blocked
11	←	<ack = 4, buf = 1>	←	A may now send 5
12	←	<ack = 4, buf = 2>	←	B found a new buffer somewhere
13	→	<seq = 5, data = m5>	→	A has 1 buffer left
14	→	<seq = 6, data = m6>	→	A is now blocked again
15	←	<ack = 6, buf = 0>	←	A is still blocked
16	•••	<ack = 6, buf = 4>	←	Potential deadlock

Fig. 6-16. Dynamic buffer allocation. The arrows show the direction of transmission. An ellipsis (...) indicates a lost TPDU.

Potential problems with buffer allocation schemes of this kind can arise in datagram networks if control TPDUs can get lost. Look at line 16. *B* has now allocated more buffers to *A*, but the allocation TPDU was lost. Since control TPDUs are not sequenced or timed out, *A* is now deadlocked. To prevent this situation, each host should periodically send control TPDUs giving the acknowledgement and buffer status on each connection. That way, the deadlock will be broken, sooner or later.

Up until now we have tacitly assumed that the only limit imposed on the sender's data rate is the amount of buffer space available in the receiver. As memory prices continue to fall dramatically, it may become feasible to equip hosts with so much memory that lack of buffers is rarely, if ever, a problem.

When buffer space no longer limits the maximum flow, another bottleneck will appear: the carrying capacity of the subnet. If adjacent routers can exchange at most *x* frames/sec and there are *k* disjoint paths between a pair of hosts, there is no way that those hosts can exchange more than *kx* TPDUs/sec, no matter how much buffer space is available at each end. If the sender pushes too hard (i.e., sends more than *kx* TPDUs/sec), the subnet will become congested, because it will be unable to deliver TPDUs as fast as they are coming in.

What is needed is a mechanism based on the subnet's carrying capacity rather than on the receiver's buffering capacity. Clearly, the flow control mechanism must be applied at the sender to prevent it from having too many unacknowledged TPDUs outstanding at once. Belsnes (1975) proposed using a sliding window flow control scheme in which the sender dynamically adjusts the window size to match the network's carrying capacity. If the network can handle *c* TPDUs/sec and the cycle time (including transmission, propagation, queueing, processing at the receiver, and return of the acknowledgement) is *r*, then the sender's window should be *cr*. With a window of this size the sender normally operates with the pipeline full. Any small decrease in network performance will cause it to block.

In order to adjust the window size periodically, the sender could monitor both parameters and then compute the desired window size. The carrying capacity can be determined by simply counting the number of TPDUs acknowledged during some time period and then dividing by the time period. During the measurement, the sender should send as fast as it can, to make sure that the network's carrying capacity, and not the low input rate, is the factor limiting the acknowledgement rate. The time required for a transmitted TPDU to be acknowledged can be measured exactly and a running mean maintained. Since the capacity of the network depends on the amount of traffic in it, the window size should be adjusted frequently, to track changes in the carrying capacity. As we will see later, the Internet uses a similar scheme.

6.2.5. Multiplexing

Multiplexing several conversations onto connections, virtual circuits, and physical links plays a role in several layers of the network architecture. In the transport layer the need for multiplexing can arise in a number of ways. For example, in networks that use virtual circuits within the subnet, each open connection consumes some table space in the routers for the entire duration of the connection. If buffers are dedicated to the virtual circuit in each router as well, a user who left a terminal logged into a remote machine during a coffee break is nevertheless consuming expensive resources. Although this implementation of packet switching defeats one of the main reasons for having packet switching in the first place—to bill the user based on the amount of data sent, not the connect time—many carriers have chosen this approach because it so closely resembles the circuit switching model to which they have grown accustomed over the decades.

The consequence of a price structure that heavily penalizes installations for having many virtual circuits open for long periods of time is to make multiplexing of different transport connections onto the same network connection attractive. This form of multiplexing, called **upward multiplexing**, is shown in Fig. 6-17(a). In this figure, four distinct transport connections all use the same network connection (e.g., ATM virtual circuit) to the remote host. When connect time forms the major component of the carrier's bill, it is up to the transport layer to group transport connections according to their destination and map each group onto the minimum number of network connections. If too many transport connections are mapped onto one network connection, the performance will be poor, because the window will usually be full, and users will have to wait their turn to send one message. If too few transport connections are mapped onto one network connection, the service will be expensive. When upward multiplexing is used with ATM, we have the ironic (tragic?) situation of having to identify the connection using a field in the transport header, even though ATM provides more than 4000 virtual circuit numbers per virtual path expressly for that purpose.

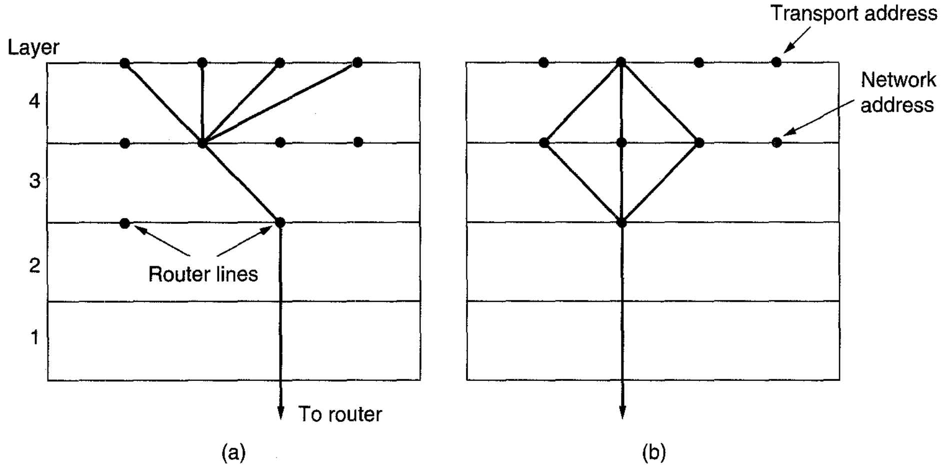


Fig. 6-17. (a) Upward multiplexing. (b) Downward multiplexing.

Multiplexing can also be useful in the transport layer for another reason, related to carrier technical decisions rather than carrier pricing decisions. Suppose, for example, that a certain important user needs a high-bandwidth connection from time to time. If the subnet enforces a sliding window flow control with an n-bit sequence number, the user must stop sending as soon as $2^n - 1$ packets are outstanding and must wait for the packets to propagate to the remote host and be acknowledged. If the physical connection is via a satellite, the user is effectively limited to $2^n - 1$ packets every 540 msec. With, for example, $n = 8$ and 128-byte packets, the usable bandwidth is about 484 kbps, even though the physical channel bandwidth is more than 100 times higher.

One possible solution is to have the transport layer open multiple network connections and distribute the traffic among them on a round-robin basis, as indicated in Fig. 6-17(b). This modus operandi is called **downward multiplexing**. With k network connections open, the effective bandwidth is increased by a factor of k. With 4095 virtual circuits, 128-byte packets, and an 8-bit sequence number, it is theoretically possible to achieve data rates in excess of 1.6 Gbps. Of course, this performance can be achieved only if the output line can support 1.6 Gbps, because all 4095 virtual circuits are still being sent out over one physical line, at least in Fig. 6-17(b). If multiple output lines are available, downward multiplexing can also be used to increase the performance even more.

6.2.6. Crash Recovery

If hosts and routers are subject to crashes, recovery from these crashes becomes an issue. If the transport entity is entirely within the hosts, recovery from network and router crashes is straightforward. If the network layer provides datagram service, the transport entities expect lost TPDUs all the time and know how to cope with them. If the network layer provides connection-oriented service, then loss of a virtual circuit is handled by establishing a new one and then probing the remote transport entity to ask it which TPDUs it has received and which ones it has not received. The latter ones can be retransmitted.

A more troublesome problem is how to recover from host crashes. In particular, it may be desirable for clients to be able to continue working when servers crash and then quickly reboot. To illustrate the difficulty, let us assume that one host, the client, is sending a long file to another host, the file server, using a simple stop-and-wait protocol. The transport layer on the server simply passes the incoming TPDUs to the transport user, one by one. Part way through the transmission, the server crashes. When it comes back up, its tables are reinitialized, so it no longer knows precisely where it was.

In an attempt to recover its previous status, the server might send a broadcast TPDU to all other hosts, announcing that it had just crashed and requesting that its clients inform it of the status of all open connections. Each client can be in one of two states: one TPDU outstanding, *S1*, or no TPDUs outstanding, *S0*. Based on only this state information, the client must decide whether or not to retransmit the most recent TPDU.

At first glance it would seem obvious: the client should retransmit only if it has an unacknowledged TPDU outstanding (i.e., is in state *S1*) when it learns of the crash. However, a closer inspection reveals difficulties with this naive approach. Consider, for example, the situation when the server's transport entity first sends an acknowledgement, and then, when the acknowledgement has been sent, performs the write up to the application process. Writing a TPDU onto the output stream and sending an acknowledgement are two distinct indivisible events that cannot be done simultaneously. If a crash occurs after the acknowledgement has been sent but before the write has been done, the client will receive the acknowledgement and thus be in state *S0* when the crash recovery announcement arrives. The client will therefore not retransmit, (incorrectly) thinking that the TPDU has arrived. This decision by the client leads to a missing TPDU.

At this point you may be thinking: "That problem can be solved easily. All you have to do is reprogram the transport entity to first do the write and then send the acknowledgement." Try again. Imagine that the write has been done but the crash occurs before the acknowledgement can be sent. The client will be in state *S1* and thus retransmit, leading to an undetected duplicate TPDU in the output stream to the server application process.

No matter how the sender and receiver are programmed, there are always situations where the protocol fails to recover properly. The server can be programmed in one of two ways: acknowledge first or write first. The client can be programmed in one of four ways: always retransmit the last TPDU, never retransmit the last TPDU, retransmit only in state *S0*, or retransmit only in state *S1*. This gives eight combinations, but as we shall see, for each combination there is some set of events that makes the protocol fail.

Three events are possible at the server: sending an acknowledgement (*A*), writing to the output process (*W*), and crashing (*C*). The three events can occur in six different orderings: *AC*(*W*), *AWC*, *C*(*AW*), *C*(*WA*), *WAC*, and *WC*(*A*), where the parentheses are used to indicate that neither *A* nor *W* may follow *C* (i.e., once it has crashed, it has crashed). Figure 6-18 shows all eight combinations of client and server strategy and the valid event sequences for each one. Notice that for each strategy there is some sequence of events that causes the protocol to fail. For example, if the client always retransmits, the *AWC* event will generate an undetected duplicate, even though the other two events work properly.

	Strategy used by receiving host					
	First ACK, then write			First write, then ACK		
Strategy used by sending host	AC(W)	AWC	C(AW)	C(WA)	W AC	WC(A)
Always retransmit	OK	DUP	OK	OK	DUP	DUP
Never retransmit	LOST	OK	LOST	LOST	OK	OK
Retransmit in S0	OK	DUP	LOST	LOST	DUP	OK
Retransmit in S1	LOST	OK	OK	OK	OK	DUP

OK = Protocol functions correctly
DUP = Protocol generates a duplicate message
LOST = Protocol loses a message

Fig. 6-18. Different combinations of client and server strategy.

Making the protocol more elaborate does not help. Even if the client and server exchange several TPDUs before the server attempts to write, so that the client knows exactly what is about to happen, the client has no way of knowing whether a crash occurred just before or just after the write. The conclusion is inescapable: under our ground rules of no simultaneous events, host crash and recovery cannot be made transparent to higher layers.

Put in more general terms, this result can be restated as recovery from a layer N crash can only be done by layer $N + 1$, and then only if the higher layer retains enough status information. As mentioned above, the transport layer can recover from failures in the network layer, provided that each end of a connection keeps track of where it is.

This problem gets us into the issue of what a so-called end-to-end acknowledgement really means. In principle, the transport protocol is end-to-end and not chained like the lower layers. Now consider the case of a user entering requests for transactions against a remote database. Suppose that the remote transport entity is programmed to first pass TPDUs to the next layer up and then acknowledge. Even in this case, the receipt of an acknowledgement back at the user's machine does not necessarily mean that the remote host stayed up long enough to actually update the database. A truly end-to-end acknowledgement, whose receipt means that the work has actually been done, and lack thereof means that it has not, is probably impossible to achieve. This point is discussed in more detail by Saltzer et al. (1984).

6.3. A SIMPLE TRANSPORT PROTOCOL

To make the ideas discussed so far more concrete, in this section we will study an example transport layer in detail. The example has been carefully chosen to be reasonably realistic, yet still simple enough to be easy to understand. The abstract service primitives we will use are the connection-oriented primitives of Fig. 6-3.

6.3.1. The Example Service Primitives

Our first problem is how to express these transport primitives concretely. CONNECT is easy: we will just have a library procedure *connect* that can be called with the appropriate parameters necessary to establish a connection. The parameters are the local and remote TSAPs. During the call, the caller is blocked (i.e., suspended) while the transport entity tries to set up the connection. If the connection succeeds, the caller is unblocked, and can start transmitting data.

When a process wants to be able to accept incoming calls, it calls *listen*, specifying a particular TSAP to listen to. The process then blocks until some remote process attempts to establish a connection to its TSAP.

Note that this model is highly asymmetric. One side is passive, executing a *listen* and waiting until something happens. The other side is active and initiates the connection. An interesting question arises of what to do if the active side begins first. One strategy is to have the connection attempt fail if there is no listener at the remote TSAP. Another strategy is to have the initiator block (possibly forever) until a listener appears.

A compromise, used in our example, is to hold the connection request at the receiving end for a certain time interval. If a process on that host calls *listen* before the timer goes off, the connection is established; otherwise, it is rejected and the caller is unblocked and given an error return.

To release a connection, we will use a procedure *disconnect*. When both sides have disconnected, the connection is released. In other words, we are using a symmetric disconnection model.

Data transmission has precisely the same problem as connection establishment: sending is active but receiving is passive. We will use the same solution for data transmission as for connection establishment, an active call *send* that transmits data, and a passive call *receive* that blocks until a TPDU arrives.

Our concrete service definition thus consists of five primitives: CONNECT, LISTEN, DISCONNECT, SEND, and RECEIVE. Each primitive corresponds exactly with a library procedure that executes the primitive. The parameters for the service primitives and library procedures are as follows:

```
connum  = LISTEN(local)
connum  = CONNECT(local, remote)
status  = SEND(connum, buffer, bytes)
status  = RECEIVE(connum, buffer, bytes)
status  = DISCONNECT(connum)
```

The LISTEN primitive announces the caller's willingness to accept connection requests directed at the indicated TSAP. The user of the primitive is blocked until an attempt is made to connect to it. There is no timeout.

The CONNECT primitive takes two parameters, a local TSAP (i.e., transport address), *local*, and a remote TSAP, *remote*, and tries to establish a transport connection between the two. If it succeeds, it returns in *connum* a nonnegative number used to identify the connection on subsequent calls. If it fails, the reason for failure is put in *connum* as a negative number. In our simple model, each TSAP may participate in only one transport connection, so a possible reason for failure is that one of the transport addresses is currently in use. Some other reasons are: remote host down, illegal local address, and illegal remote address.

The SEND primitive transmits the contents of the buffer as a message on the indicated transport connection, possibly in several units if it is too big. Possible errors, returned in *status*, are no connection, illegal buffer address, or negative count.

The RECEIVE primitive indicates the caller's desire to accept data. The size of the incoming message is placed in *bytes*. If the remote process has released the connection or the buffer address is illegal (e.g., outside the user's program), *status* is set to an error code indicating the nature of the problem.

The DISCONNECT primitive terminates a transport connection. The parameter *connum* tells which one. Possible errors are *connum* belongs to another process, or *connum* is not a valid connection identifier. The error code, or 0 for success, is returned in *status*.

6.3.2. The Example Transport Entity

Before looking at the code of the example transport entity, please be sure you realize that this example is analogous to the early examples presented in Chap. 3: it is more for pedagogical purposes than a serious proposal. Many of the technical details (such as extensive error checking) that would be needed in a production system have been omitted here for the sake of simplicity.

The transport layer makes use of the network service primitives to send and receive TPDUs. For this example, we need to choose network service primitives to use. One choice would have been unreliable datagram service. We have not made that choice to keep the example simple. With unreliable datagram service, the transport code would have been large and complex, mostly dealing with lost and delayed packets. Furthermore, most of these ideas have already been discussed at length in Chap. 3.

Instead, we have chosen to use a connection-oriented reliable network service. This way we can focus on transport issues that do not occur in the lower layers. These include connection establishment, connection release, and credit management, among others. A simple transport service built on top of an ATM network might look something like this.

In general, the transport entity may be part of the host's operating system or it may be a package of library routines running within the user's address space. It may also be contained on a coprocessor chip or network board plugged into the host's backplane. For simplicity, our example has been programmed as though it were a library package, but the changes needed to make it part of the operating system are minimal (primarily how user buffers are accessed).

It is worth noting, however, that in this example, the "transport entity" is not really a separate entity at all, but part of the user process. In particular, when the user executes a primitive that blocks, such as LISTEN, the entire transport entity blocks as well. While this design is fine for a host with only a single user process, on a host with multiple users, it would be more natural to have the transport entity be a separate process, distinct from all the user processes.

The interface to the network layer is via the procedures *to_net* and *from_net* (not shown). Each has six parameters. First comes the connection identifier, which maps one-to-one onto network virtual circuits. Next come the *Q* and *M* bits, which, when set to 1, indicate control message and more data from this message follows in the next packet, respectively. After that we have the packet type, chosen from the set of six packet types listed in Fig. 6-19. Finally, we have a pointer to the data itself, and an integer giving the number of bytes of data.

Network packet	Meaning
CALL REQUEST	Sent to establish a connection
CALL ACCEPTED	Response to CALL REQUEST
CLEAR REQUEST	Sent to release a connection
CLEAR CONFIRMATION	Response to CLEAR REQUEST
DATA	Used to transport data
CREDIT	Control packet for managing the window

Fig. 6-19. The network layer packets used in our example.

On calls to *to_net*, the transport entity fills in all the parameters for the network layer to read; on calls to *from_net*, the network layer dismembers an incoming packet for the transport entity. By passing information as procedure parameters rather than passing the actual outgoing or incoming packet itself, the transport layer is shielded from the details of the network layer protocol. If the transport entity should attempt to send a packet when the underlying virtual circuit's sliding window is full, it is suspended within *to_net* until there is room in the window. This mechanism is transparent to the transport entity and is controlled by the network layer using commands like *enable_transport_layer* and *disable_transport_layer* analogous to those described in the protocols of Chap. 3. The management of the packet layer window is also done by the network layer.

In addition to this transparent suspension mechanism, there are also explicit *sleep* and *wakeup* procedures (not shown) called by the transport entity. The procedure *sleep* is called when the transport entity is logically blocked waiting for an external event to happen, generally the arrival of a packet. After *sleep* has been called, the transport entity (and the user process, of course) stop executing.

The actual code of the transport entity is shown in Fig. 6-20. Each connection is always in one of seven states, as follows:

1. IDLE—Connection not established yet.
2. WAITING—CONNECT has been executed and CALL REQUEST sent.
3. QUEUED—A CALL REQUEST has arrived; no LISTEN yet.
4. ESTABLISHED—The connection has been established.
5. SENDING—The user is waiting for permission to send a packet.
6. RECEIVING—A RECEIVE has been done.
7. DISCONNECTING—A DISCONNECT has been done locally.

Transitions between states can occur when any of the following events occur: a primitive is executed, a packet arrives, or the timer expires.

```
#define MAX_CONN 32                     /* maximum number of simultaneous connections */
#define MAX_MSG_SIZE 8192               /* largest message in bytes */
#define MAX_PKT_SIZE 512                /* largest packet in bytes */
#define TIMEOUT 20
#define CRED 1
#define OK 0

#define ERR_FULL -1
#define ERR_REJECT -2
#define ERR_CLOSED -3
#define LOW_ERR -3

typedef int transport_address;
typedef enum {CALL_REQ,CALL_ACC,CLEAR_REQ,CLEAR_CONF,DATA_PKT,CREDIT} pkt_type;
typedef enum {IDLE,WAITING,QUEUED,ESTABLISHED,SENDING,RECEIVING,DISCONN} cstate;

/* Global variables. */
transport_address listen_address;       /* local address being listened to */
int listen_conn;                        /* connection identifier for listen */
unsigned char data[MAX_PKT_SIZE];       /* scratch area for packet data */

struct conn {
 transport_address local_address, remote_address;
 cstate state;                          /* state of this connection */
 unsigned char *user_buf_addr;          /* pointer to receive buffer */
 int byte_count;                        /* send/receive count */
 int clr_req_received;                  /* set when CLEAR_REQ packet received */
 int timer;                             /* used to time out CALL_REQ packets */
 int credits;                           /* number of messages that may be sent */
} conn[MAX_CONN];

void sleep(void);                       /* prototypes */
void wakeup(void);
void to_net(int cid, int q, int m, pkt_type pt, unsigned char *p, int bytes);
void from_net(int *cid, int *q, int *m, pkt_type *pt, unsigned char *p, int *bytes);

int listen(transport_address t)
{ /* User wants to listen for a connection. See if CALL_REQ has already arrived. */
 int i = 1, found = 0;

 for (i = 1; i <= MAX_CONN; i++)        /* search the table for CALL_REQ */
     if (conn[i].state == QUEUED && conn[i].local_address == t) {
          found = i;
          break;
     }

 if (found == 0) {
     /* No CALL_REQ is waiting.  Go to sleep until arrival or timeout. */
     listen_address = t;  sleep();  i = listen_conn ;
 }
 conn[i].state = ESTABLISHED;           /* connection is ESTABLISHED */
 conn[i].timer = 0;                     /* timer is not used */
```

```
  listen_conn = 0;                          /* 0 is assumed to be an invalid address */
  to_net(i, 0, 0, CALL_ACC, data, 0);       /* tell net to accept connection */
  return(i);                                /* return connection identifier */
}

int connect(transport_address l, transport_address r)
{ /* User wants to connect to a remote process;  send CALL_REQ packet. */
  int i;
  struct conn *cptr;

  data[0] = r;  data[1] = l;                /* CALL_REQ packet needs these */
  i = MAX_CONN;                             /* search table backward */
  while (conn[i].state != IDLE && i > 1) i = i - 1;
  if (conn[i].state == IDLE) {
      /* Make a table entry that CALL_REQ has been sent. */
      cptr = &conn[i];
      cptr->local_address = l; cptr->remote_address = r;
      cptr->state = WAITING; cptr->clr_req_received = 0;
      cptr->credits = 0; cptr->timer = 0;
      to_net(i, 0, 0, CALL_REQ, data, 2);
      sleep();                              /* wait for CALL_ACC or CLEAR_REQ */
      if (cptr->state == ESTABLISHED) return(i);
      if (cptr->clr_req_received) {
          /* Other side refused call. */
          cptr->state = IDLE;               /* back to IDLE state */
          to_net(i, 0, 0, CLEAR_CONF, data, 0);
          return(ERR_REJECT);
      }
  } else return(ERR_FULL);                  /* reject CONNECT: no table space */
}

int send(int cid, unsigned char bufptr[], int bytes)
{ /* User wants to send a message. */
  int i, count, m;
  struct conn *cptr = &conn[cid];

  /* Enter SENDING state. */
  cptr->state = SENDING;
  cptr->byte_count = 0;                     /* # bytes sent so far this message */
  if (cptr->clr_req_received == 0 && cptr->credits == 0) sleep();
  if (cptr->clr_req_received == 0) {
      /* Credit available; split message into packets if need be. */
      do {
          if (bytes - cptr->byte_count > MAX_PKT_SIZE) {/* multipacket message */
              count = MAX_PKT_SIZE;  m = 1;  /* more packets later */
          } else {                          /* single packet message */
              count = bytes - cptr->byte_count;  m = 0;    /* last pkt of this message */
          }
          for (i = 0; i < count; i++) data[i] = bufptr[cptr->byte_count + i];
          to_net(cid, 0, m, DATA_PKT, data, count);  /* send 1 packet */
          cptr->byte_count = cptr->byte_count + count;      /* increment bytes sent so far */
      } while (cptr->byte_count < bytes);         /* loop until whole message sent */
      cptr->credits--;                  /* each message uses up one credit */
      cptr->state = ESTABLISHED;
      return(OK);
  } else {
      cptr->state = ESTABLISHED;
      return(ERR_CLOSED);               /* send failed: peer wants to disconnect */
  }
}

int receive(int cid, unsigned char bufptr[], int *bytes)
{ /* User is prepared to receive a message. */
  struct conn *cptr = &conn[cid];

  if (cptr->clr_req_received == 0) {
      /* Connection still established; try to receive. */
      cptr->state = RECEIVING;
      cptr->user_buf_addr = bufptr;
      cptr->byte_count = 0;
      data[0] = CRED;
      data[1] = 1;
      to_net(cid, 1, 0, CREDIT, data, 2);   /* send credit */
      sleep();                          /* block awaiting data */
      *bytes = cptr->byte_count;
  }
  cptr->state = ESTABLISHED;
  return(cptr->clr_req_received ? ERR_CLOSED : OK);
}

int disconnect(int cid)
{ /* User wants to release a connection. */
  struct conn *cptr = &conn[cid];

  if (cptr->clr_req_received) {         /* other side initiated termination */
      cptr->state = IDLE;               /* connection is now released */
      to_net(cid, 0, 0, CLEAR_CONF, data, 0);
  } else {                              /* we initiated termination */
      cptr->state = DISCONN;            /* not released until other side agrees */
      to_net(cid, 0, 0, CLEAR_REQ, data, 0);
  }
  return(OK);
}

void packet_arrival(void)
{ /* A packet has arrived, get and process it. */
  int cid;                              /* connection on which packet arrived */
  int count, i, q, m;
  pkt_type ptype;   /* CALL_REQ, CALL_ACC, CLEAR_REQ, CLEAR_CONF, DATA_PKT, CREDIT */
  unsigned char data[MAX_PKT_SIZE];     /* data portion of the incoming packet */
  struct conn *cptr;

  from_net(&cid, &q, &m, &ptype, data, &count);  /* go get it */
  cptr = &conn[cid];
```

```
 switch (ptype) {
   case CALL_REQ:                        /* remote user wants to establish connection */
     cptr->local_address = data[0]; cptr->remote_address = data[1];
     if (cptr->local_address == listen_address) {
          listen_conn = cid; cptr->state = ESTABLISHED; wakeup();
     } else {
          cptr->state = QUEUED; cptr->timer = TIMEOUT;
     }
     cptr->clr_req_received = 0; cptr->credits = 0;
     break;

   case CALL_ACC:                        /* remote user has accepted our CALL_REQ */
     cptr->state = ESTABLISHED;
     wakeup();
     break;

   case CLEAR_REQ:                       /* remote user wants to disconnect or reject call */
     cptr->clr_req_received = 1;
     if (cptr->state == DISCONN) cptr->state = IDLE; /* clear collision */
     if (cptr->state == WAITING || cptr->state == RECEIVING || cptr->state == SENDING) wakeup();
     break;

   case CLEAR_CONF:                      /* remote user agrees to disconnect */
     cptr->state = IDLE;
     break;

   case CREDIT:                          /* remote user is waiting for data */
     cptr->credits += data[1];
     if (cptr->state == SENDING) wakeup();
     break;

   case DATA_PKT:                        /* remote user has sent data */
     for (i = 0; i < count; i++) cptr->user_buf_addr[cptr->byte_count + i] = data[i];
     cptr->byte_count += count;
     if (m == 0 ) wakeup();
 }
}

void clock(void)
{ /* The clock has ticked, check for timeouts of queued connect requests. */
 int i;
 struct conn *cptr;

 for (i = 1; i <= MAX_CONN; i++) {
     cptr = &conn[i];
     if (cptr->timer > 0) {              /* timer was running */
          cptr->timer−−;
          if (cptr->timer == 0) {        /* timer has now expired */
               cptr->state = IDLE;
               to_net(i, 0, 0, CLEAR_REQ, data, 0);
          }
     }
 }
}
```

Fig. 6-20. An example transport entity.

The procedures shown in Fig. 6-20 are of two types. Most are directly callable by user programs. *packet_arrival* and *clock* are different, however. They are spontaneously triggered by external events: the arrival of a packet and the clock ticking, respectively. In effect, they are interrupt routines. We will assume that they are never invoked while a transport entity procedure is running. Only when the user process is sleeping or executing outside the transport entity may they be called. This property is crucial to the correct functioning of the transport entity.

The existence of the Q (Qualifier) bit in the packet header allows us to avoid the overhead of a transport protocol header. Ordinary data messages are sent as data packets with $Q = 0$. Transport protocol control messages, of which there is only one (CREDIT) in our example, are sent as data packets with $Q = 1$. These control messages are detected and processed by the receiving transport entity.

The main data structure used by the transport entity is the array *conn*, which has one record for each potential connection. The record maintains the state of the connection, including the transport addresses at either end, the number of messages sent and received on the connection, the current state, the user buffer pointer, the number of bytes of the current messages sent or received so far, a bit indicating that the remote user has issued a DISCONNECT, a timer, and a permission counter used to enable sending of messages. Not all of these fields are used in our simple example, but a complete transport entity would need all of them, and perhaps more. Each *conn* entry is assumed initialized to the *IDLE* state.

When the user calls CONNECT, the network layer is instructed to send a CALL REQUEST packet to the remote machine, and the user is put to sleep. When the CALL REQUEST packet arrives at the other side, the transport entity is interrupted to run *packet_arrival* to check if the local user is listening on the specified address. If so, a CALL ACCEPTED packet is sent back and the remote user is awakened; if not, the CALL REQUEST is queued for *TIMEOUT* clock ticks. If a LISTEN is done within this period, the connection is established; otherwise, it times out and is rejected with a CLEAR REQUEST packet. This mechanism is needed to prevent the initiator from blocking forever in the event that the remote process does not want to connect to it.

Although we have eliminated the transport protocol header, we still need a way to keep track of which packet belongs to which transport connection, since multiple connections may exist simultaneously. The simplest approach is to use the network layer virtual circuit number as the transport connection number as well. Furthermore, the virtual circuit number can also be used as the index into the *conn* array. When a packet comes in on network layer virtual circuit k, it belongs to transport connection k, whose state is in the record *conn*[k]. For connections initiated at a host, the connection number is chosen by the originating transport entity. For incoming calls, the network layer makes the choice, choosing any unused virtual circuit number.

To avoid having to provide and manage buffers within the transport entity, a flow control mechanism different from the traditional sliding window is used here. Instead, when a user calls RECEIVE, a special **credit message** is sent to the transport entity on the sending machine and is recorded in the *conn* array. When SEND is called, the transport entity checks to see if a credit has arrived on the specified connection. If so, the message is sent (in multiple packets if need be) and the credit decremented; if not, the transport entity puts itself to sleep until a credit arrives. This mechanism guarantees that no message is ever sent unless the other side has already done a RECEIVE. As a result, whenever a message arrives there is guaranteed to be a buffer available into which it can be put. The scheme can easily be generalized to allow receivers to provide multiple buffers and request multiple messages.

You should keep the simplicity of Fig. 6-20 in mind. A realistic transport entity would normally check all user supplied parameters for validity, handle recovery from network layer crashes, deal with call collisions, and support a more general transport service including such facilities as interrupts, datagrams, and nonblocking versions of the SEND and RECEIVE primitives.

6.3.3. The Example as a Finite State Machine

Writing a transport entity is difficult and exacting work, especially for more realistic protocols. To reduce the chance of making an error, it is often useful to represent the state of the protocol as a finite state machine.

We have already seen that our example protocol has seven states per connection. It is also possible to isolate 12 events that can happen to move a connection from one state to another. Five of these events are the five service primitives. Another six are the arrivals of the six legal packet types. The last one is the expiration of the timer. Figure 6-21 shows the main protocol actions in matrix form. The columns are the states and the rows are the 12 events.

Each entry in the matrix (i.e., the finite state machine) of Fig. 6-21 has up to three fields: a predicate, an action, and a new state. The predicate indicates under what conditions the action is taken. For example, in the upper left-hand entry, if a LISTEN is executed and there is no more table space (predicate *P1*), the LISTEN fails and the state does not change. On the other hand, if a CALL REQUEST packet has already arrived for the transport address being listened to (predicate *P2*), the connection is established immediately. Another possibility is that *P2* is false, that is, no CALL REQUEST has come in, in which case the connection remains in the *IDLE* state, awaiting a CALL REQUEST packet.

It is worth pointing out that the choice of states to use in the matrix is not entirely fixed by the protocol itself. In this example, there is no state *LISTENING*, which might have been a reasonable thing to have following a LISTEN. There is no *LISTENING* state because a state is associated with a connection record entry, and no connection record is created by LISTEN. Why not? Because we have decided to use the network layer virtual circuit numbers as the connection

		State						
		Idle	Waiting	Queued	Established	Sending	Receiving	Dis-connecting
Primitives	LISTEN	P1: ~/Idle P2: A1/Estab $\overline{P2}$: A2/Idle		~/Estab				
	CONNECT	P1: ~/Idle $\overline{P1}$: A3/Wait						
	DISCONNECT				P4: A5/Idle $\overline{P4}$: A6/Disc			
	SEND				P5: A7/Estab $\overline{P5}$: A8/Send			
	RECEIVE				A9/Receiving			
Incoming packets	Call_req	P3: A1/Estab $\overline{P3}$: A4/Queu'd						
	Call_acc		~/Estab					
	Clear_req		~/Idle		A10/Estab	A10/Estab	A10/Estab	~/Idle
	Clear_conf							~/Idle
	DataPkt						A12/Estab	
	Credit				A11/Estab	A7/Estab		
Clock	Timeout			~/Idle				

Predicates	Actions	
P1: Connection table full	A1: Send Call_acc	A7: Send message
P2: Call_req pending	A2: Wait for Call_req	A8: Wait for credit
P3: LISTEN pending	A3: Send Call_req	A9: Send credit
P4: Clear_req pending	A4: Start timer	A10: Set Clr_req_received flag
P5: Credit available	A5: Send Clear_conf	A11: Record credit
	A6: Send Clear_req	A12: Accept message

Fig. 6-21. The example protocol as a finite state machine. Each entry has an optional predicate, an optional action, and the new state. The tilde indicates that no major action is taken. An overbar above a predicate indicates the negation of the predicate. Blank entries correspond to impossible or invalid events.

identifiers, and for a LISTEN, the virtual circuit number is ultimately chosen by the network layer when the CALL REQUEST packet arrives.

The actions *A1* through *A12* are the major actions, such as sending packets and starting timers. Not all the minor actions, such as initializing the fields of a connection record, are listed. If an action involves waking up a sleeping process, the actions following the wakeup also count. For example, if a CALL REQUEST packet comes in and a process was asleep waiting for it, the transmission of the CALL ACCEPT packet following the wakeup counts as part of the action for CALL REQUEST. After each action is performed, the connection may move to a new state, as shown in Fig. 6-21.

The advantage of representing the protocol as a matrix is threefold. First, in this form it is much easier for the programmer to systematically check each combination of state and event to see if an action is required. In production implementations, some of the combinations would be used for error handling. In Fig. 6-21 no distinction is made between impossible situations and illegal ones. For example, if a connection is in *waiting* state, the DISCONNECT event is impossible because the user is blocked and cannot execute any primitives at all. On the other hand, in *sending* state, data packets are not expected because no credit has been issued. The arrival of a data packet is a protocol error.

The second advantage of the matrix representation of the protocol is in implementing it. One could envision a two-dimensional array in which element $a[i][j]$ was a pointer or index to the procedure that handled the occurrence of event i when in state j. One possible implementation is to write the transport entity as a short loop, waiting for an event at the top of the loop. When an event happens, the relevant connection is located and its state is extracted. With the event and state now known, the transport entity just indexes into the array a and calls the proper procedure. This approach gives a much more regular and systematic design than our transport entity.

The third advantage of the finite state machine approach is for protocol description. In some standards documents, the protocols are given as finite state machines of the type of Fig. 6-21. Going from this kind of description to a working transport entity is much easier if the transport entity is also driven by a finite state machine based on the one in the standard.

The primary disadvantage of the finite state machine approach is that it may be more difficult to understand than the straight programming example we used initially. However, this problem may be partially solved by drawing the finite state machine as a graph, as is done in Fig. 6-22.

6.4. THE INTERNET TRANSPORT PROTOCOLS (TCP AND UDP)

The Internet has two main protocols in the transport layer, a connection-oriented protocol and a connectionless one. In the following sections we will study both of them. The connection-oriented protocol is TCP. The connectionless protocol is UDP. Because UDP is basically just IP with a short header added, we will focus on TCP.

TCP (**Transmission Control Protocol**) was specifically designed to provide a reliable end-to-end byte stream over an unreliable internetwork. An

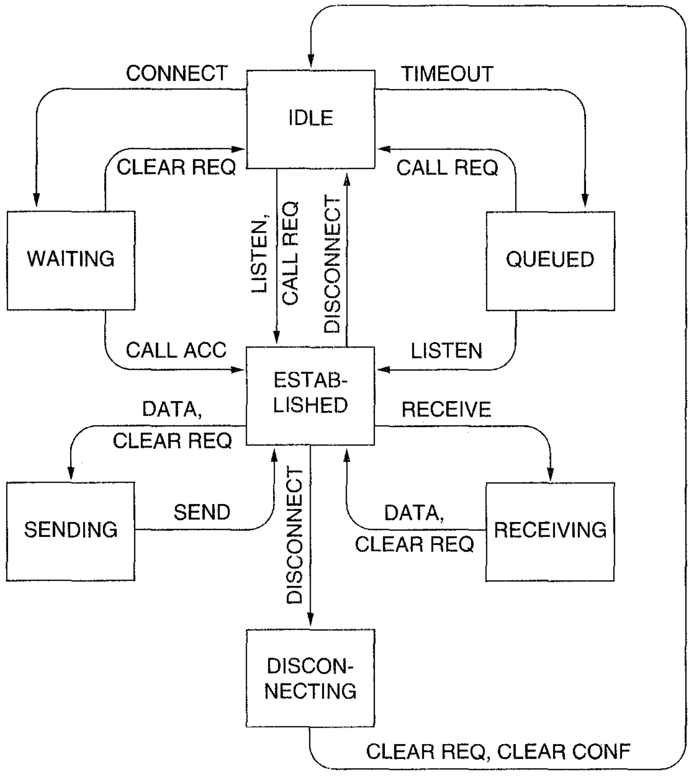


Fig. 6-22. The example protocol in graphical form. Transitions that leave the connection state unchanged have been omitted for simplicity.

internetwork differs from a single network because different parts may have wildly different topologies, bandwidths, delays, packet sizes, and other parameters. TCP was designed to dynamically adapt to properties of the internetwork and to be robust in the face of many kinds of failures.

TCP was formally defined in RFC 793. As time went on, various errors and inconsistencies were detected, and the requirements were changed in some areas. These clarifications and some bug fixes are detailed in RFC 1122. Extensions are given in RFC 1323.

Each machine supporting TCP has a TCP transport entity, either a user process or part of the kernel that manages TCP streams and interfaces to the IP layer. A TCP entity accepts user data streams from local processes, breaks them up into pieces not exceeding 64K bytes (in practice, usually about 1500 bytes), and sends each piece as a separate IP datagram. When IP datagrams containing TCP data arrive at a machine, they are given to the TCP entity, which reconstructs the original byte streams. For simplicity, we will sometimes use just "TCP" to mean the

TCP transport entity (a piece of software) or the TCP protocol (a set of rules). From the context it will be clear which is meant. For example, in "The user gives TCP the data," the TCP transport entity is clearly intended.

The IP layer gives no guarantee that datagrams will be delivered properly, so it is up to TCP to time out and retransmit them as need be. Datagrams that do arrive may well do so in the wrong order; it is also up to TCP to reassemble them into messages in the proper sequence. In short, TCP must furnish the reliability that most users want and that IP does not provide.

6.4.1. The TCP Service Model

TCP service is obtained by having both the sender and receiver create end points, called sockets, as discussed in Sec. 6.1.3. Each socket has a socket number (address) consisting of the IP address of the host and a 16-bit number local to that host, called a **port**. A port is the TCP name for a TSAP. To obtain TCP service, a connection must be explicitly established between a socket on the sending machine and a socket on the receiving machine. The socket calls are listed in Fig. 6-6.

A socket may be used for multiple connections at the same time. In other words, two or more connections may terminate at the same socket. Connections are identified by the socket identifiers at both ends, that is, (*socket1*, *socket2*). No virtual circuit numbers or other identifiers are used.

Port numbers below 256 are called **well-known ports** and are reserved for standard services. For example, any process wishing to establish a connection to a host to transfer a file using FTP can connect to the destination host's port 21 to contact its FTP daemon. Similarly, to establish a remote login session using TELNET, port 23 is used. The list of well-known ports is given in RFC 1700.

All TCP connections are full-duplex and point-to-point. Full duplex means that traffic can go in both directions at the same time. Point-to-point means that each connection has exactly two end points. TCP does not support multicasting or broadcasting.

A TCP connection is a byte stream, not a message stream. Message boundaries are not preserved end to end. For example, if the sending process does four 512-byte writes to a TCP stream, these data may be delivered to the receiving process as four 512-byte chunks, two 1024-byte chunks, one 2048-byte chunk (see Fig. 6-23), or some other way. There is no way for the receiver to detect the unit(s) in which the data were written.

Files in UNIX have this property too. The reader of a file cannot tell whether the file was written a block at a time, a byte at a time, or all in one blow. As with a UNIX file, the TCP software has no idea of what the bytes mean and no interest in finding out. A byte is just a byte.

When an application passes data to TCP, TCP may send it immediately or buffer it (in order to collect a larger amount to send at once), at its discretion.

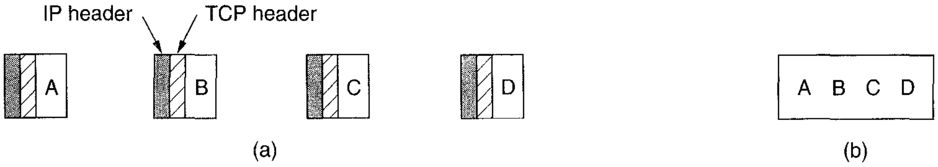


Fig. 6-23. (a) Four 512-byte segments sent as separate IP datagrams. (b) The 2048 bytes of data delivered to the application in a single READ call.

However, sometimes, the application really wants the data to be sent immediately. For example, suppose a user is logged into a remote machine. After a command line has been finished and the carriage return typed, it is essential that the line be shipped off to the remote machine immediately and not buffered until the next line comes in. To force data out, applications can use the PUSH flag, which tells TCP not to delay the transmission.

Some early applications used the PUSH flag as a kind of marker to delineate messages boundaries. While this trick sometimes works, it sometimes fails since not all implementations of TCP pass the PUSH flag to the application on the receiving side. Furthermore, if additional PUSHes come in before the first one has been transmitted (e.g., because the output line is busy), TCP is free to collect all the PUSHed data into a single IP datagram, with no separation between the various pieces.

One last feature of the TCP service that is worth mentioning here is **urgent data**. When an interactive user hits the DEL or CTRL-C key to break off a remote computation that has already begun, the sending application puts some control information in the data stream and gives it to TCP along with the URGENT flag. This event causes TCP to stop accumulating data and transmit everything it has for that connection immediately.

When the urgent data are received at the destination, the receiving application is interrupted (e.g., given a signal in UNIX terms), so it can stop whatever it was doing and read the data stream to find the urgent data. The end of the urgent data is marked, so the application knows when it is over. The start of the urgent data is not marked. It is up to the application to figure that out. This scheme basically provides a crude signaling mechanism and leaves everything else up to the application.

6.4.2. The TCP Protocol

In this section we will give a general overview of the TCP protocol. In the next one we will go over the protocol header, field by field. Every byte on a TCP connection has its own 32-bit sequence number. For a host blasting away at full speed on a 10-Mbps LAN, theoretically the sequence numbers could wrap around in an hour, but in practice it takes much longer. The sequence numbers are used both for acknowledgements and for the window mechanism, which use separate 32-bit header fields.

The sending and receiving TCP entities exchange data in the form of segments. A **segment** consists of a fixed 20-byte header (plus an optional part) followed by zero or more data bytes. The TCP software decides how big segments should be. It can accumulate data from several writes into one segment or split data from one write over multiple segments. Two limits restrict the segment size. First, each segment, including the TCP header, must fit in the 65,535 byte IP payload. Second, each network has a **maximum transfer unit** or **MTU**, and each segment must fit in the MTU. In practice, the MTU is generally a few thousand bytes and thus defines the upper bound on segment size. If a segment passes through a sequence of networks without being fragmented and then hits one whose MTU is smaller than the segment, the router at the boundary fragments the segment into two or more smaller segments.

A segment that is too large for a network that it must transit can be broken up into multiple segments by a router. Each new segment gets its own IP header, so fragmentation by routers increases the total overhead (because each additional segment adds 20 bytes of extra header information in the form of an IP header).

The basic protocol used by TCP entities is the sliding window protocol. When a sender transmits a segment, it also starts a timer. When the segment arrives at the destination, the receiving TCP entity sends back a segment (with data if any exists, otherwise without data) bearing an acknowledgement number equal to the next sequence number it expects to receive. If the sender's timer goes off before the acknowledgement is received, the sender transmits the segment again.

Although this protocol sounds simple, there are many ins and outs that we will cover below. For example, since segments can be fragmented, it is possible that part of a transmitted segment arrives and is acknowledged by the receiving TCP entity, but the rest is lost. Segments can also arrive out of order, so bytes 3072–4095 can arrive but cannot be acknowledged because bytes 2048–3071 have not turned up yet. Segments can also be delayed so long in transit that the sender times out and retransmits them. If a retransmitted segment takes a different route than the original, and is fragmented differently, bits and pieces of both the original and the duplicate can arrive sporadically, requiring a careful administration to achieve a reliable byte stream. Finally, with so many networks making up the Internet, it is possible that a segment may occasionally hit a congested (or broken) network along its path.

TCP must be prepared to deal with these problems and solve them in an efficient way. A considerable amount of effort has gone into optimizing the performance of TCP streams, even in the face of network problems. A number of the algorithms used by many TCP implementations will be discussed below.

6.4.3. The TCP Segment Header

Figure 6-24 shows the layout of a TCP segment. Every segment begins with a fixed-format 20-byte header. The fixed header may be followed by header options. After the options, if any, up to 65,535 − 20 − 20 = 65,515 data bytes may follow, where the first 20 refers to the IP header and the second to the TCP header. Segments without any data are legal and are commonly used for acknowledgements and control messages.

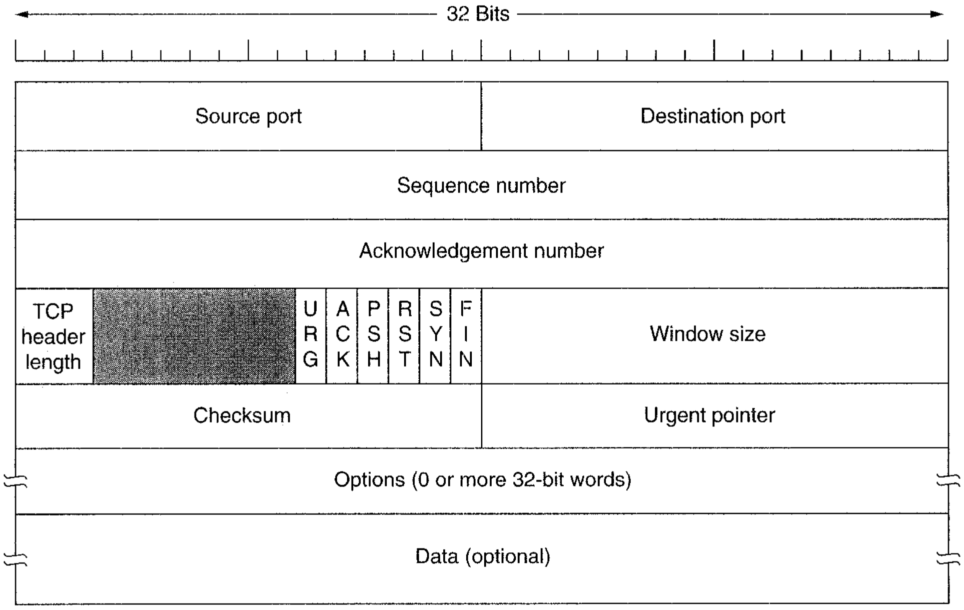


Fig. 6-24. The TCP header.

Let us dissect the TCP header field by field. The *Source port* and *Destination port* fields identify the local end points of the connection. Each host may decide for itself how to allocate its own ports starting at 256. A port plus its host's IP address forms a 48-bit unique TSAP. The source and destination socket numbers together identify the connection.

The *Sequence number* and *Acknowledgement number* fields perform their usual functions. Note that the latter specifies the next byte expected, not the last byte correctly received. Both are 32 bits long because every byte of data is numbered in a TCP stream.

The *TCP header length* tells how many 32-bit words are contained in the TCP header. This information is needed because the *Options* field is of variable length, so the header is too. Technically, this field really indicates the start of the data within the segment, measured in 32-bit words, but that number is just the header length in words, so the effect is the same.

Next comes a 6-bit field that is not used. The fact that this field has survived intact for over a decade is testimony to how well thought out TCP is. Lesser protocols would have needed it to fix bugs in the original design.

Now come six 1-bit flags. *URG* is set to 1 if the *Urgent pointer* is in use. The *Urgent pointer* is used to indicate a byte offset from the current sequence number at which urgent data are to be found. This facility is in lieu of interrupt messages. As we mentioned above, this facility is a bare bones way of allowing the sender to signal the receiver without getting TCP itself involved in the reason for the interrupt.

The *ACK* bit is set to 1 to indicate that the *Acknowledgement number* is valid. If *ACK* is 0, the segment does not contain an acknowledgement so the *Acknowledgement number* field is ignored.

The *PSH* bit indicates PUSHed data. The receiver is hereby kindly requested to deliver the data to the application upon arrival and not buffer it until a full buffer has been received (which it might otherwise do for efficiency reasons).

The *RST* bit is used to reset a connection that has become confused due to a host crash or some other reason. It is also used to reject an invalid segment or refuse an attempt to open a connection. In general, if you get a segment with the *RST* bit on, you have a problem on your hands.

The *SYN* bit is used to establish connections. The connection request has *SYN* = 1 and *ACK* = 0 to indicate that the piggyback acknowledgement field is not in use. The connection reply does bear an acknowledgement, so it has *SYN* = 1 and *ACK* = 1. In essence the *SYN* bit is used to denote CONNECTION REQUEST and CONNECTION ACCEPTED, with the *ACK* bit used to distinguish between those two possibilities.

The *FIN* bit is used to release a connection. It specifies that the sender has no more data to transmit. However, after closing a connection, a process may continue to receive data indefinitely. Both *SYN* and *FIN* segments have sequence numbers and are thus guaranteed to be processed in the correct order.

Flow control in TCP is handled using a variable-size sliding window. The *Window* field tells how many bytes may be sent starting at the byte acknowledged. A *Window* field of 0 is legal and says that the bytes up to and including *Acknowledgement number* − 1 have been received, but that the receiver is currently badly in need of a rest and would like no more data for the moment, thank you. Permission to send can be granted later by sending a segment with the same *Acknowledgement number* and a nonzero *Window* field.

A *Checksum* is also provided for extreme reliability. It checksums the header, the data, and the conceptual pseudoheader shown in Fig. 6-25. When performing this computation, the TCP *Checksum* field is set to zero, and the data field is padded out with an additional zero byte if its length is an odd number. The checksum algorithm is simply to add up all the 16-bit words in 1's complement and then to take the 1's complement of the sum. As a consequence, when the receiver performs the calculation on the entire segment, including the *Checksum* field, the result should be 0.

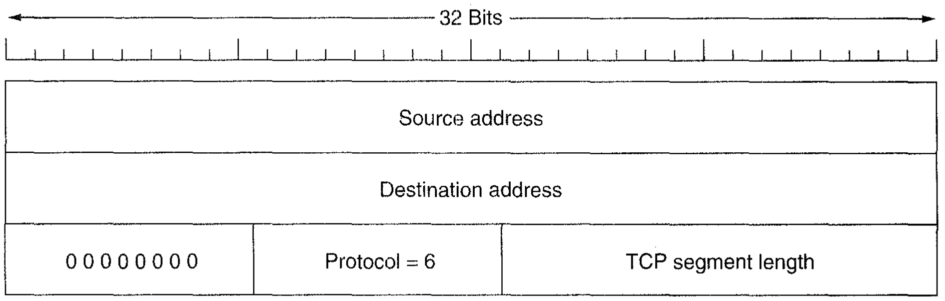


Fig. 6-25. The pseudoheader included in the TCP checksum.

The pseudoheader contains the 32-bit IP addresses of the source and destination machines, the protocol number for TCP (6), and the byte count for the TCP segment (including the header). Including the pseudoheader in the TCP checksum computation helps detect misdelivered packets, but doing so violates the protocol hierarchy since the IP addresses in it belong to the IP layer, not the TCP layer.

The *Options* field was designed to provide a way to add extra facilities not covered by the regular header. The most important option is the one that allows each host to specify the maximum TCP payload it is willing to accept. Using large segments is more efficient than using small ones because the 20-byte header can then be amortized over more data, but small hosts may not be able to handle very large segments. During connection setup, each side can announce its maximum and see its partner's. The smaller of the two numbers wins. If a host does not use this option, it defaults to a 536-byte payload. All Internet hosts are required to accept TCP segments of 536 + 20 = 556 bytes.

For lines with high bandwidth, high delay, or both, the 64 KB window is often a problem. On a T3 line (44.736 Mbps), it takes only 12 msec to output a full 64 KB window. If the round trip propagation delay is 50 msec (typical for a transcontinental fiber), the sender will be idle 3/4 of the time waiting for acknowledgements. On a satellite connection, the situation is even worse. A larger window size would allow the sender to keep pumping data out, but using the 16-bit *Window size* field, there is no way to express such a size. In RFC 1323, a *Window scale* option was proposed, allowing the sender and receiver to negotiate a window scale factor. This number allows both sides to shift the *Window size* field up to 16 bits to the left, thus allowing windows of up to 2^{32} bytes. Most TCP implementations now support this option.

Another option proposed by RFC 1106 and now widely implemented is the use of the selective repeat instead of go back n protocol. If the receiver gets one bad segment and then a large number of good ones, the normal TCP protocol will eventually time out and retransmit all the unacknowledged segments, including all those that were received correctly. RFC 1106 introduced NAKs, to allow the receiver to ask for a specific segment (or segments). After it gets these, it can acknowledge all the buffered data, thus reducing the amount of data retransmitted.

6.4.4. TCP Connection Management

Connections are established in TCP using the three-way handshake discussed in Sec. 6.2.2. To establish a connection, one side, say the server, passively waits for an incoming connection by executing the LISTEN and ACCEPT primitives, either specifying a specific source or nobody in particular.

The other side, say the client, executes a CONNECT primitive, specifying the IP address and port to which it wants to connect, the maximum TCP segment size it is willing to accept, and optionally some user data (e.g., a password). The CONNECT primitive sends a TCP segment with the *SYN* bit on and *ACK* bit off and waits for a response.

When this segment arrives at the destination, the TCP entity there checks to see if there is a process that has done a LISTEN on the port given in the *Destination port* field. If not, it sends a reply with the *RST* bit on to reject the connection.

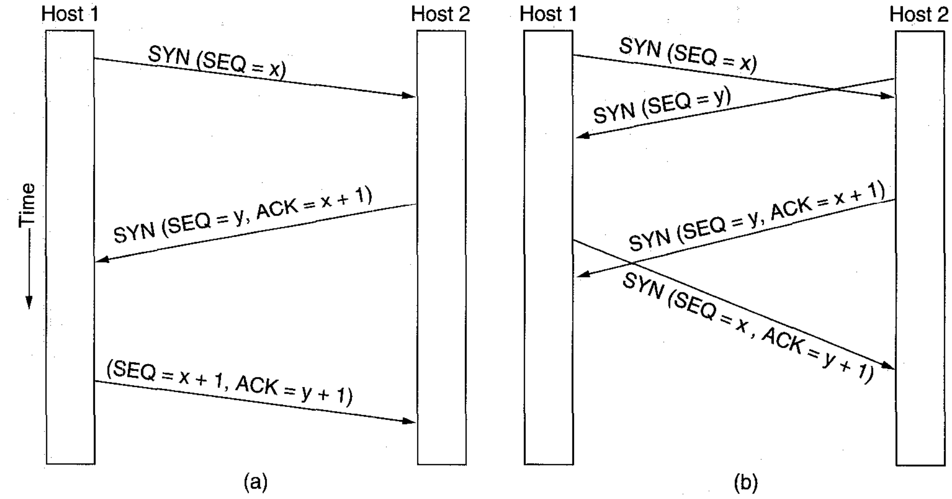

Host 1
Host 2
Host 1
Host 2
SYN (SEQ = x)
SYN (SEQ = y, ACK = x + 1)
(SEQ = x + 1, ACK = y + 1)
Time
SYN (SEQ = x)
SYN (SEQ = y)
SYN (SEQ = y, ACK = x + 1)
SYN (SEQ = x, ACK = y + 1)
(a)
(b)

Fig. 6-26. (a) TCP connection establishment in the normal case. (b) Call collision.

If some process is listening to the port, that process is given the incoming TCP segment. It can then either accept or reject the connection. If it accepts, an acknowledgement segment is sent back. The sequence of TCP segments sent in the normal case is shown in Fig. 6-26(a). Note that a *SYN* segment consumes 1 byte of sequence space so it can be acknowledged unambiguously.

In the event that two hosts simultaneously attempt to establish a connection between the same two sockets, the sequence of events is as illustrated in Fig. 6-26(b). The result of these events is that just one connection is established, not two because connections are identified by their end points. If the first setup results in a connection identified by (x, y) and the second one does too, only one table entry is made, namely, for (x, y).

The initial sequence number on a connection is not 0 for the reasons we discussed earlier. A clock-based scheme is used, with a clock tick every 4 μsec. For additional safety, when a host crashes, it may not reboot for the maximum packet lifetime (120 sec) to make sure that no packets from previous connections are still roaming around the Internet somewhere.

Although TCP connections are full duplex, to understand how connections are released it is best to think of them as a pair of simplex connections. Each simplex connection is released independently of its sibling. To release a connection, either party can send a TCP segment with the *FIN* bit set, which means that it has no more data to transmit. When the *FIN* is acknowledged, that direction is shut down for new data. Data may continue to flow indefinitely in the other direction, however. When both directions have been shut down, the connection is released. Normally, four TCP segments are needed to release a connection, one *FIN* and one *ACK* for each direction. However, it is possible for the first *ACK* and the second *FIN* to be contained in the same segment, reducing the total count to three.

Just as with telephone calls in which both people say goodbye and hang up the phone simultaneously, both ends of a TCP connection may send *FIN* segments at the same time. These are each acknowledged in the usual way, and the connection shut down. There is, in fact, no essential difference between the two hosts releasing sequentially or simultaneously.

To avoid the two-army problem, timers are used. If a response to a *FIN* is not forthcoming within two maximum packet lifetimes, the sender of the *FIN* releases the connection. The other side will eventually notice that nobody seems to be listening to it any more, and time out as well. While this solution is not perfect, given the fact that a perfect solution is theoretically impossible, it will have to do. In practice, problems rarely arise.

The steps required to establish and release connections can be represented in a finite state machine with the 11 states listed in Fig. 6-27. In each state, certain events are legal. When a legal event happens, some action may be taken. If some other event happens, an error is reported.

Each connection starts in the *CLOSED* state. It leaves that state when it does either a passive open (LISTEN), or an active open (CONNECT). If the other side does the opposite one, a connection is established and the state becomes *ESTABLISHED*. Connection release can be initiated by either side. When it is complete, the state returns to *CLOSED*.

The finite state machine itself is shown in Fig. 6-28. The common case of a client actively connecting to a passive server is shown with heavy lines—solid for

State	Description
CLOSED	No connection is active or pending
LISTEN	The server is waiting for an incoming call
SYN RCVD	A connection request has arrived; wait for ACK
SYN SENT	The application has started to open a connection
ESTABLISHED	The normal data transfer state
FIN WAIT 1	The application has said it is finished
FIN WAIT 2	The other side has agreed to release
TIMED WAIT	Wait for all packets to die off
CLOSING	Both sides have tried to close simultaneously
CLOSE WAIT	The other side has initiated a release
LAST ACK	Wait for all packets to die off

Fig. 6-27. The states used in the TCP connection management finite state machine.

the client, dotted for the server. The lightface lines are unusual event sequences. Each line in Fig. 6-28 is marked by an *event/action* pair. The event can either be a user-initiated system call (CONNECT, LISTEN, SEND, or CLOSE), a segment arrival (*SYN*, *FIN*, *ACK*, or *RST*), or in one case, a timeout of twice the maximum packet lifetime. The action is the sending of a control segment (*SYN*, *FIN*, or *RST*) or nothing, indicated by —. Comments are shown in parentheses.

The diagram can best be understood by first following the path of a client (the heavy solid line) then later the path of a server (the heavy dashed line). When an application on the client machine issues a CONNECT request, the local TCP entity creates a connection record, marks it as being in the *SYN SENT* state, and sends a *SYN* segment. Note that many connections may be open (or being opened) at the same time on behalf of multiple applications, so the state is per connection and recorded in the connection record. When the *SYN+ACK* arrives, TCP sends the final *ACK* of the three-way handshake and switches into the *ESTABLISHED* state. Data can now be sent and received.

When an application is finished, it executes a CLOSE primitive, which causes the local TCP entity to send a *FIN* segment and wait for the corresponding *ACK* (dashed box marked active close). When the *ACK* arrives, a transition is made to state *FIN WAIT 2* and one direction of the connection is now closed. When the other side closes, too, a *FIN* comes in, which is acknowledged. Now both sides are closed, but TCP waits a time equal to the maximum packet lifetime to guarantee that all packets from the connection have died off, just in case the acknowledgement was lost. When the timer goes off, TCP deletes the connection record.

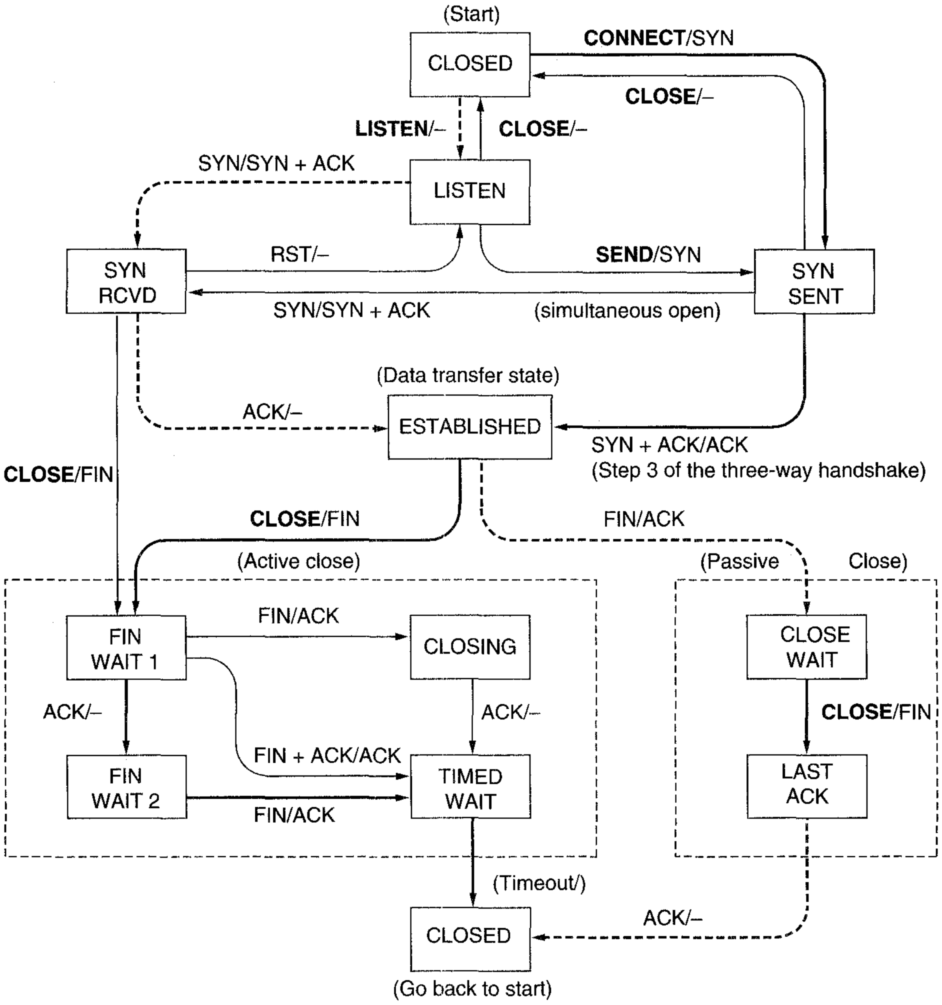


Fig. 6-28. TCP connection management finite state machine. The heavy solid line is the normal path for a client. The heavy dashed line is the normal path for a server. The light lines are unusual events.

Now let us examine connection management from the server's viewpoint. The server does a LISTEN and settles down to see who turns up. When a *SYN* comes in, it is acknowledged and the server goes to the *SYN RCVD* state. When the server's *SYN* is itself acknowledged, the three-way handshake is complete and the server goes to the *ESTABLISHED* state. Data transfer can now occur.

When the client has had enough, it does a CLOSE, which causes a *FIN* to arrive at the server (dashed box marked passive close). The server is then signaled. When it, too, does a CLOSE, a *FIN* is sent to the client. When the client's acknowledgement shows up, the server releases the connection and deletes the connection record.

6.4.5. TCP Transmission Policy

Window management in TCP is not directly tied to acknowledgements as it is in most data link protocols. For example, suppose the receiver has a 4096-byte buffer as shown in Fig. 6-29. If the sender transmits a 2048-byte segment that is correctly received, the receiver will acknowledge the segment. However, since it now has only 2048 of buffer space (until the application removes some data from the buffer), it will advertise a window of 2048 starting at the next byte expected.

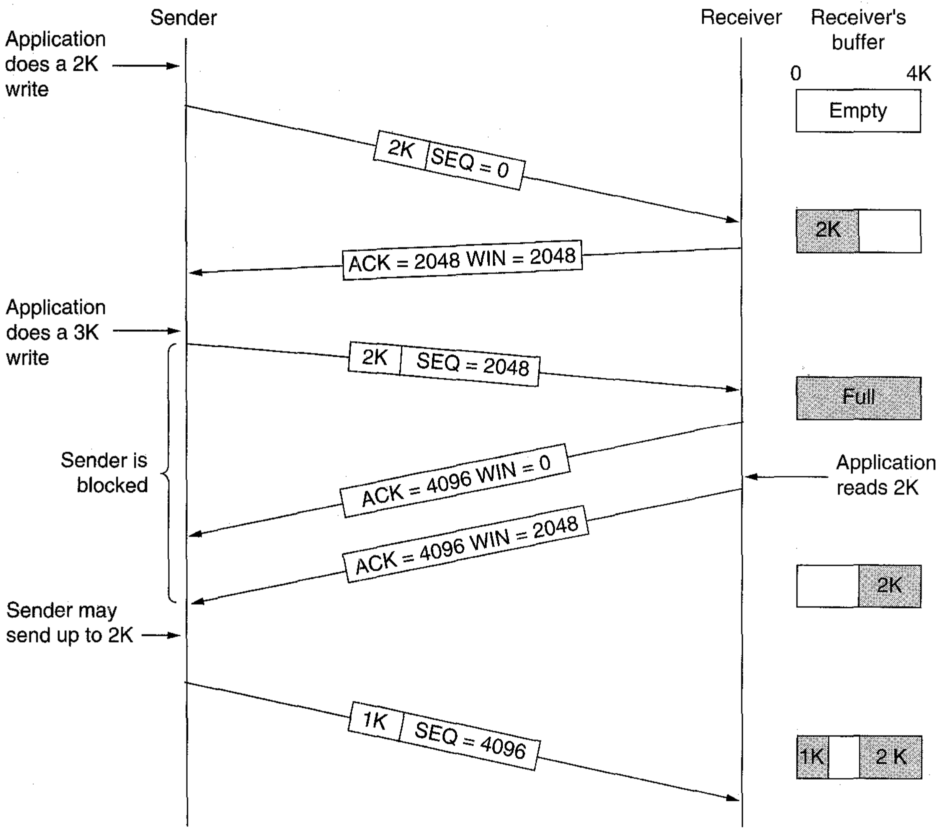


Fig. 6-29. Window management in TCP.

Now the sender transmits another 2048 bytes, which are acknowledged, but the advertised window is 0. The sender must stop until the application process on the receiving host has removed some data from the buffer, at which time TCP can advertise a larger window.

When the window is 0, the sender may not normally send segments, with two exceptions. First, urgent data may be sent, for example, to allow the user to kill the process running on the remote machine. Second, the sender may send a 1-byte segment to make the receiver reannounce the next byte expected and window size. The TCP standard explicitly provides this option to prevent deadlock if a window announcement ever gets lost.

Senders are not required to transmit data as soon as they come in from the application. Neither are receivers required to send acknowledgements as soon as possible. For example, in Fig. 6-29, When the first 2 KB of data came in, TCP, knowing that it had a 4-KB window available, would have been completely correct in just buffering the data until another 2 KB came in, to be able to transmit a segment with a 4-KB payload. This freedom can be exploited to improve performance.

Consider a TELNET connection to an interactive editor that reacts on every keystroke. In the worst case, when a character arrives at the sending TCP entity, TCP creates a 21-byte TCP segment, which it gives to IP to send as a 41-byte IP datagram. At the receiving side, TCP immediately sends a 40-byte acknowledgement (20 bytes of TCP header and 20 bytes of IP header). Later, when the editor has read the byte, TCP sends a window update, moving the window 1 byte to the right. This packet is also 40 bytes. Finally, when the editor has processed the character, it echoes it as a 41-byte packet. In all, 162 bytes of bandwidth are used and four segments are sent for each character typed. When bandwidth is scarce, this method of doing business is not desirable.

One approach that many TCP implementations use to optimize this situation is to delay acknowledgements and window updates for 500 msec in the hope of acquiring some data on which to hitch a free ride. Assuming the editor echoes within 500 msec, only one 41-byte packet now need be sent back to the remote user, cutting the packet count and bandwidth usage in half.

Although this rule reduces the load placed on the network by the receiver, the sender is still operating inefficiently by sending 41-byte packets containing 1 byte of data. A way to reduce this usage is known as **Nagle's algorithm** (Nagle, 1984). What Nagle suggested is simple: when data come into the sender one byte at a time, just send the first byte and buffer all the rest until the outstanding byte is acknowledged. Then send all the buffered characters in one TCP segment and start buffering again until they are all acknowledged. If the user is typing quickly and the network is slow, a substantial number of characters may go in each segment, greatly reducing the bandwidth used. The algorithm additionally allows a new packet to be sent if enough data have trickled in to fill half the window or a maximum segment.

Nagle's algorithm is widely used by TCP implementations, but there are times when it is better to disable it. In particular, when an X-Windows application is being run over the Internet, mouse movements have to be sent to the remote computer. Gathering them up to send in bursts makes the mouse cursor move erratically, which makes for unhappy users.

Another problem that can ruin TCP performance is the **silly window syndrome** (Clark, 1982). This problem occurs when data are passed to the sending TCP entity in large blocks, but an interactive application on the receiving side reads data 1 byte at a time. To see the problem, look at Fig. 6-30. Initially, the TCP buffer on the receiving side is full and the sender knows this (i.e., has a window of size 0). Then the interactive application reads one character from the TCP stream. This action makes the receiving TCP happy, so it sends a window update to the sender saying that it is all right to send 1 byte. The sender obliges and sends 1 byte. The buffer is now full, so the receiver acknowledges the 1-byte segment but sets the window to 0. This behavior can go on forever.

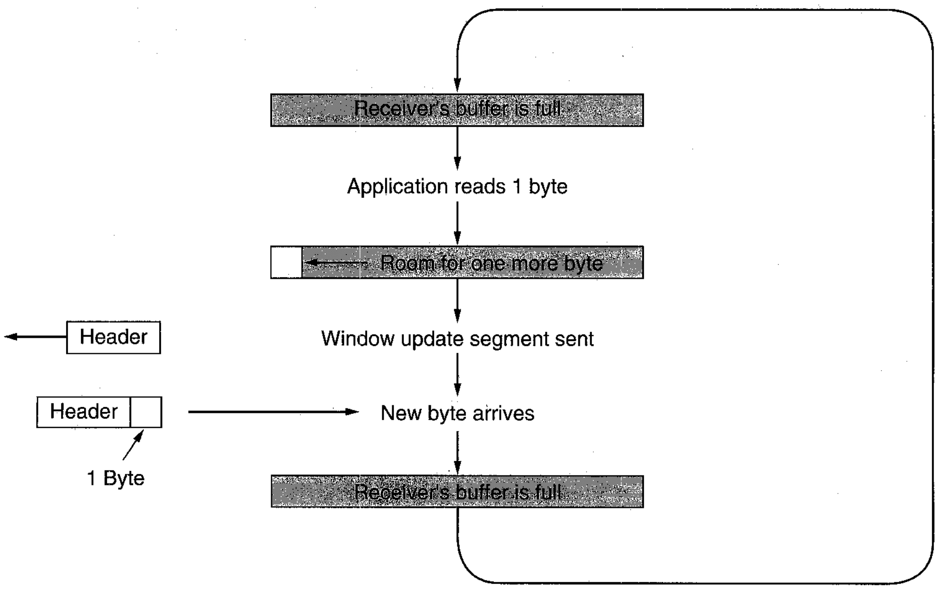


Fig. 6-30. Silly window syndrome.

Clark's solution is to prevent the receiver from sending a window update for 1 byte. Instead it is forced to wait until it has a decent amount of space available and advertise that instead. Specifically, the receiver should not send a window update until it can handle the maximum segment size it advertised when the connection was established, or its buffer is half empty, whichever is smaller.

Furthermore, the sender can also help by not sending tiny segments. Instead, it should try to wait until it has accumulated enough space in the window to send a full segment or at least one containing half of the receiver's buffer size (which it must estimate from the pattern of window updates it has received in the past).

Nagle's algorithm and Clark's solution to the silly window syndrome are complementary. Nagle was trying to solve the problem caused by the sending application delivering data to TCP a byte at a time. Clark was trying to solve the problem of the receiving application sucking the data up from TCP a byte at a time. Both solutions are valid and can work together. The goal is for the sender not to send small segments and the receiver not to ask for them.

The receiving TCP can go further in improving performance than just doing window updates in large units. Like the sending TCP, it also has the ability to buffer data, so it can block a READ request from the application until it has a large chunk of data to provide. Doing this reduces the number of calls to TCP, and hence the overhead. Of course, it also increases the response time, but for noninteractive applications like file transfer, efficiency may outweigh response time to individual requests.

Another receiver issue is what to do with out of order segments. They can be kept or discarded, at the receiver's discretion. Of course, acknowledgements can be sent only when all the data up to the byte acknowledged have been received. If the receiver gets segments 0, 1, 2, 4, 5, 6, and 7, it can acknowledge everything up to and including the last byte in segment 2. When the sender times out, it then retransmits segment 3. If the receiver has buffered segments 4 through 7, upon receipt of segment 3 it can acknowledge all bytes up to the end of segment 7.

6.4.6. TCP Congestion Control

When the load offered to any network is more than it can handle, congestion builds up. The Internet is no exception. In this section we will discuss algorithms that have been developed over the past decade to deal with congestion. Although the network layer also tries to manage congestion, most of the heavy lifting is done by TCP because the real solution to congestion is to slow down the data rate.

In theory, congestion can be dealt with by employing a principle borrowed from physics: the law of conservation of packets. The idea is not to inject a new packet into the network until an old one leaves (i.e., is delivered). TCP attempts to achieve this goal by dynamically manipulating the window size.

The first step in managing congestion is detecting it. In the old days, detecting congestion was difficult. A timeout caused by a lost packet could have been caused by either (1) noise on a transmission line or (2) packet discard at a congested router. Telling the difference was difficult.

Nowadays, packet loss due to transmission errors is relatively rare because most long-haul trunks are fiber (although wireless networks are a different story). Consequently, most transmission timeouts on the Internet are due to congestion. All the Internet TCP algorithms assume that timeouts are caused by congestion and monitor timeouts for signs of trouble the way miners watch their canaries.

Before discussing how TCP reacts to congestion, let us first describe what it does to try to prevent it from occurring in the first place. When a connection is established, a suitable window size has to be chosen. The receiver can specify a window based on its buffer size. If the sender sticks to this window size, problems will not occur due to buffer overflow at the receiving end, but they may still occur due to internal congestion within the network.

In Fig. 6-31, we see this problem illustrated hydraulically. In Fig. 6-31(a), we see a thick pipe leading to a small-capacity receiver. As long as the sender does not send more water than the bucket can contain, no water will be lost. In Fig. 6-31(b), the limiting factor is not the bucket capacity, but the internal carrying capacity of the network. If too much water comes in too fast, it will back up and some will be lost (in this case by overflowing the funnel).

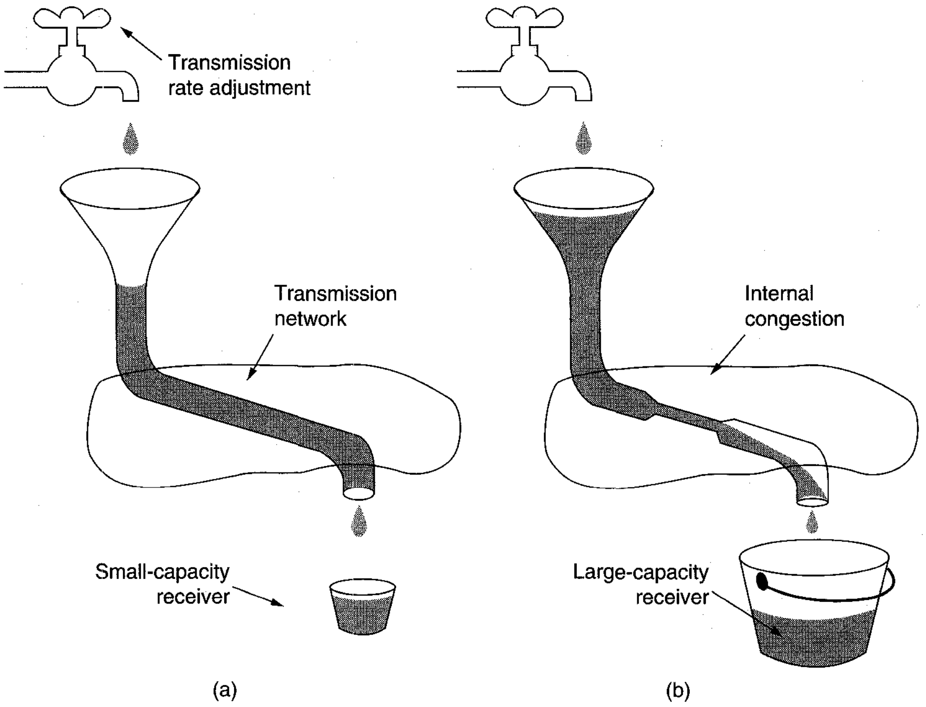


Fig. 6-31. (a) A fast network feeding a low-capacity receiver. (b) A slow network feeding a high-capacity receiver.

The Internet solution is to realize that two potential problems exist—network capacity and receiver capacity—and to deal with each of them separately. To do so, each sender maintains two windows: the window the receiver has granted and a second window, the **congestion window**. Each reflects the number of bytes the sender may transmit. The number of bytes that may be sent is the minimum of the two windows. Thus the effective window is the minimum of what the sender thinks is all right and what the receiver thinks is all right. If the receiver says "Send 8K" but the sender knows that bursts of more than 4K clog the network up, it sends 4K. On the other hand, if the receiver says "Send 8K" and the sender knows that bursts of up to 32K get through effortlessly, it sends the full 8K requested.

When a connection is established, the sender initializes the congestion window to the size of the maximum segment in use on the connection. It then sends one maximum segment. If this segment is acknowledged before the timer goes off, it adds one segment's worth of bytes to the congestion window to make it two maximum size segments and sends two segments. As each of these segments is acknowledged, the congestion window is increased by one maximum segment size. When the congestion window is *n* segments, if all *n* are acknowledged on time, the congestion window is increased by the byte count corresponding to *n* segments. In effect, each burst successfully acknowledged doubles the congestion window.

The congestion window keeps growing exponentially until either a timeout occurs or the receiver's window is reached. The idea is that if bursts of size, say, 1024, 2048, and 4096 bytes work fine, but a burst of 8192 bytes gives a timeout, the congestion window should be set to 4096 to avoid congestion. As long as the congestion window remains at 4096, no bursts longer than that will be sent, no matter how much window space the receiver grants. This algorithm is called **slow start**, but it is not slow at all (Jacobson, 1988). It is exponential. All TCP implementations are required to support it.

Now let us look at the Internet congestion control algorithm. It uses a third parameter, the **threshold**, initially 64K, in addition to the receiver and congestion windows. When a timeout occurs, the threshold is set to half of the current congestion window, and the congestion window is reset to one maximum segment. Slow start is then used to determine what the network can handle, except that exponential growth stops when the threshold is hit. From that point on, successful transmissions grow the congestion window linearly (by one maximum segment for each burst) instead of one per segment. In effect, this algorithm is guessing that it is probably acceptable to cut the congestion window in half, and then it gradually works its way up from there.

As an illustration of how the congestion algorithm works, see Fig. 6-32. The maximum segment size here is 1024 bytes. Initially the congestion window was 64K, but a timeout occurred, so the threshold is set to 32K and the congestion window to 1K for transmission 0 here. The congestion window then grows exponentially until it hits the threshold (32K). Starting then it grows linearly.

Transmission 13 is unlucky (it should have known) and a timeout occurs. The threshold is set to half the current window (by now 40K, so half is 20K) and slow start initiated all over again. When the acknowledgements from transmission 18 start coming in, the first four each increment the congestion window by one segment, but after that, growth becomes linear again.

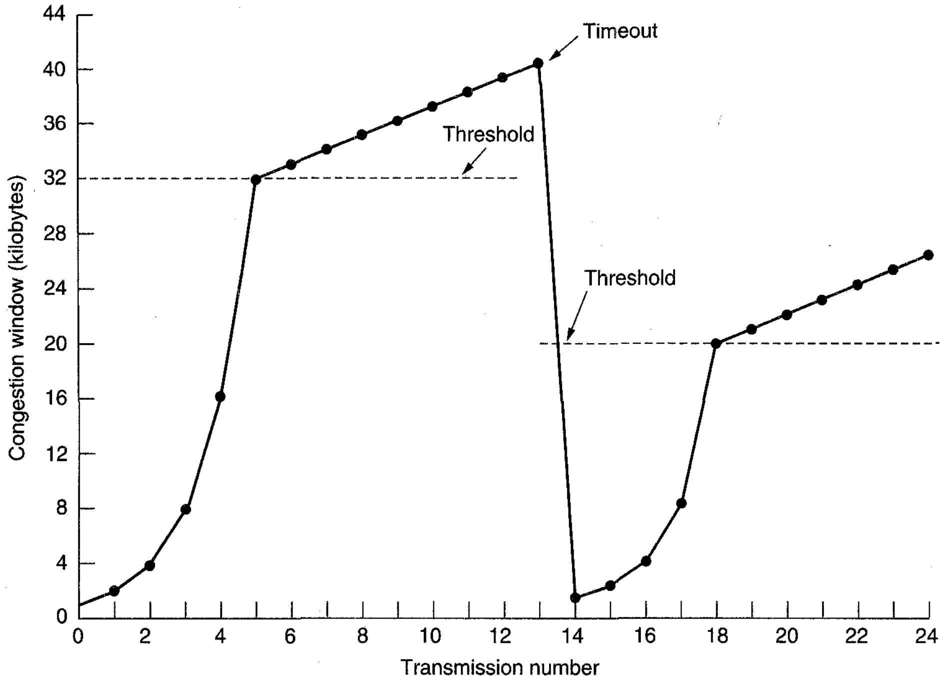


Fig. 6-32. An example of the Internet congestion algorithm.

If no more timeouts occur, the congestion window will continue to grow up to the size of the receiver's window. At that point, it will stop growing and remain constant as long as there are no more timeouts and the receiver's window does not change size. As an aside, if an ICMP SOURCE QUENCH packet comes in and is passed to TCP, this event is treated the same way as a timeout.

Work on improving the congestion control mechanism is continuing. For example, Brakmo et al. (1994) have reported improving TCP throughput by 40 percent to 70 percent by managing the clock more accurately, predicting congestion before timeouts occur, and using this early warning system to improve the slow start algorithm.

6.4.7. TCP Timer Management

TCP uses multiple timers (at least conceptually) to do its work. The most important of these is the **retransmission timer**. When a segment is sent, a retransmission timer is started. If the segment is acknowledged before the timer expires, the timer is stopped. If, on the other hand, the timer goes off before the acknowledgement comes in, the segment is retransmitted (and the timer started again). The question that arises is: How long should the timeout interval be?

This problem is much more difficult in the Internet transport layer than in the generic data link protocols of Chap. 3. In the latter case, the expected delay is highly predictable (i.e., has a low variance), so the timer can be set to go off just slightly after the acknowledgement is expected, as shown in Fig. 6-33(a). Since acknowledgements are rarely delayed in the data link layer, the absence of an acknowledgement at the expected time generally means the frame or the acknowledgement has been lost.

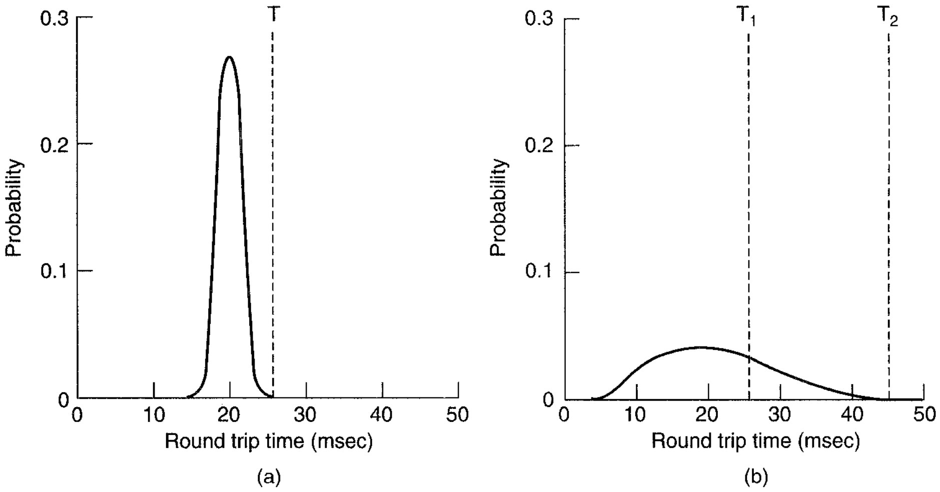


Fig. 6-33. (a) Probability density of acknowledgement arrival times in the data link layer. (b) Probability density of acknowledgement arrival times for TCP.

TCP is faced with a radically different environment. The probability density function for the time it takes for a TCP acknowledgement to come back looks more like Fig. 6-33(b) than Fig. 6-33(a). Determining the round-trip time to the destination is tricky. Even when it is known, deciding on the timeout interval is also difficult. If the timeout is set too short, say T_1 in Fig. 6-33(b), unnecessary retransmissions will occur, clogging the Internet with useless packets. If it is set too long, (T_2), performance will suffer due to the long retransmission delay whenever a packet is lost. Furthermore, the mean and variance of the acknowledgement arrival distribution can change rapidly within a few seconds as congestion builds up or is resolved.

The solution is to use a highly dynamic algorithm that constantly adjusts the timeout interval, based on continuous measurements of network performance. The algorithm generally used by TCP is due to Jacobson (1988) and works as follows. For each connection, TCP maintains a variable, *RTT*, that is the best current estimate of the round-trip time to the destination in question. When a segment is sent, a timer is started, both to see how long the acknowledgement takes and to trigger a retransmission if it takes too long. If the acknowledgement gets back before the timer expires, TCP measures how long the acknowledgement took, say, *M*. It then updates *RTT* according to the formula

$$RTT = \alpha RTT + (1 - \alpha)M$$

where α is a smoothing factor that determines how much weight is given to the old value. Typically $\alpha = 7/8$.

Even given a good value of *RTT*, choosing a suitable retransmission timeout is a nontrivial matter. Normally, TCP uses β*RTT*, but the trick is choosing β. In the initial implementations, β was always 2, but experience showed that a constant value was inflexible because it failed to respond when the variance went up.

In 1988, Jacobson proposed making β roughly proportional to the standard deviation of the acknowledgement arrival time probability density function so a large variance means a large β and vice versa. In particular, he suggested using the *mean deviation* as a cheap estimator of the *standard deviation*. His algorithm requires keeping track of another smoothed variable, *D*, the deviation. Whenever an acknowledgement comes in, the difference between the expected and observed values, $|RTT - M|$ is computed. A smoothed value of this is maintained in *D* by the formula

$$D = \alpha D + (1 - \alpha)\ |RTT - M|$$

where α may or may not be the same value used to smooth *RTT*. While *D* is not exactly the same as the standard deviation, it is good enough and Jacobson showed how it could be computed using only integer adds, subtracts, and shifts, a big plus. Most TCP implementations now use this algorithm and set the timeout interval to

$$\text{Timeout} = RTT + 4 * D$$

The choice of the factor 4 is somewhat arbitrary, but it has two advantages. First, multiplication by 4 can be done with a single shift. Second, it minimizes unnecessary timeouts and retransmissions because less than one percent of all packets come in more than four standard deviations late. (Actually, Jacobson initially said to use 2, but later work has shown that 4 gives better performance.)

One problem that occurs with the dynamic estimation of *RTT* is what to do when a segment times out and is sent again. When the acknowledgement comes in, it is unclear whether the acknowledgement refers to the first transmission or a later one. Guessing wrong can seriously contaminate the estimate of *RTT*. Phil Karn discovered this problem the hard way. He is an amateur radio enthusiast interested in transmitting TCP/IP packets by ham radio, a notoriously unreliable medium (on a good day, half the packets get through). He made a simple proposal: do not update *RTT* on any segments that have been retransmitted. Instead, the timeout is doubled on each failure until the segments get through the first time. This fix is called **Karn's algorithm**. Most TCP implementations use it.

The retransmission timer is not the only one TCP uses. A second timer is the **persistence timer**. It is designed to prevent the following deadlock. The receiver sends an acknowledgement with a window size of 0, telling the sender to wait. Later, the receiver updates the window, but the packet with the update is lost. Now both the sender and the receiver are waiting for each other to do something. When the persistence timer goes off, the sender transmits a probe to the receiver. The response to the probe gives the window size. If it is still zero, the persistence timer is set again and the cycle repeats. If it is nonzero, data can now be sent.

A third timer that some implementations use is the **keepalive timer**. When a connection has been idle for a long time, the keepalive timer may go off to cause one side to check if the other side is still there. If it fails to respond, the connection is terminated. This feature is controversial because it adds overhead and may terminate an otherwise healthy connection due to a transient network partition.

The last timer used on each TCP connection is the one used in the *TIMED WAIT* state while closing. It runs for twice the maximum packet lifetime to make sure that when a connection is closed, all packets created by it have died off.

6.4.8. UDP

The Internet protocol suite also supports a connectionless transport protocol, **UDP** (**User Data Protocol**). UDP provides a way for applications to send encapsulated raw IP datagrams and send them without having to establish a connection. Many client-server applications that have one request and one response use UDP rather than go to the trouble of establishing and later releasing a connection. UDP is described in RFC 768.

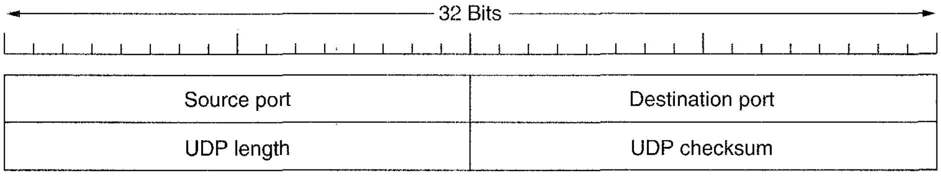


Fig. 6-34. The UDP header.

A UDP segment consists of an 8-byte header followed by the data. The header is shown in Fig. 6-34. The two ports serve the same function as they do in TCP: to identify the end points within the source and destination machines. The *UDP length* field includes the 8-byte header and the data. The *UDP checksum* includes the same format pseudoheader shown in Fig. 6-25, the UDP header, and the UDP data, padded out to an even number of bytes if need be. It is optional and stored as 0 if not computed (a true computed 0 is stored as all 1s, which is the same in 1's complement). Turning it off is foolish unless the quality of the data does not matter (e.g., digitized speech).

6.4.9. Wireless TCP and UDP

In theory, transport protocols should be independent of the technology of the underlying network layer. In particular, TCP should not care whether IP is running over fiber or over radio. In practice, it does matter because most TCP implementations have been carefully optimized based on assumptions that are true for wired networks but which fail for wireless networks. Ignoring the properties of wireless transmission can lead to a TCP implementation that is logically correct but has horrendous performance.

The principal problem is the congestion control algorithm. Nearly all TCP implementations nowadays assume that timeouts are caused by congestion, not by lost packets. Consequently, when a timer goes off, TCP slows down and sends less vigorously (e.g., Jacobson's slow start algorithm). The idea behind this approach is to reduce the network load and thus alleviate the congestion.

Unfortunately, wireless transmission links are highly unreliable. They lose packets all the time. The proper approach to dealing with lost packets is to send them again, and as quickly as possible. Slowing down just makes matters worse. If, say, 20 percent of all packets are lost, then when the sender transmits 100 packets/sec, the throughput is 80 packets/sec. If the sender slows down to 50 packets/sec, the throughput drops to 40 packets/sec.

In effect, when a packet is lost on a wired network, the sender should slow down. When one is lost on a wireless network, the sender should try harder. When the sender does not know what the network is, it is difficult to make the correct decision.

Frequently, the path from sender to receiver is inhomogeneous. The first 1000 km might be over a wired network, but the last 1 km might be wireless. Now making the correct decision on a timeout is even harder, since it matters where the problem occurred. A solution proposed by Bakne and Badrinath (1995), **indirect TCP**, is to split the TCP connection into two separate connections, as shown in Fig. 6-35. The first connection goes from the sender to the base station. The second one goes from the base station to the receiver. The base station simply copies packets between the connections in both directions.

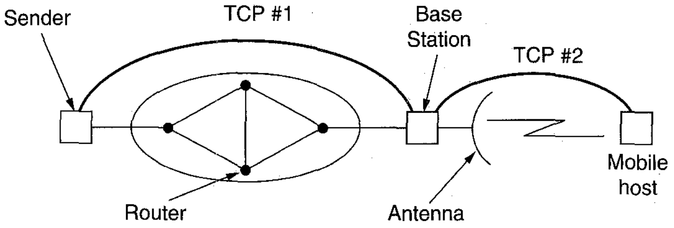


Fig. 6-35. Splitting a TCP connection into two connections.

The advantage of this scheme is that both connections are now homogeneous. Timeouts on the first connection can slow the sender down, whereas timeouts on the second one can speed it up. Other parameters can also be tuned separately for the two connections. The disadvantage is that it violates the semantics of TCP. Since each part of the connection is a full TCP connection, the base station acknowledges each TCP segment in the usual way. Only now, receipt of an acknowledgement by the sender does not mean that the receiver got the segment, only that the base station got it.

A different solution, due to Balakrishnan et al. (1995), does not break the semantics of TCP. It works by making several small modifications to the network layer code in the base station. One of the changes is the addition of a snooping agent that observes and caches TCP segments going out to the mobile host, and acknowledgements coming back from it. When the snooping agent sees a TCP segment going out to the mobile host but does not see an acknowledgement coming back before its (relatively short) timer goes off, it just retransmits that segment, without telling the source that it is doing so. It also generates a retransmission when it sees duplicate acknowledgements from the mobile host go by, invariably meaning that the mobile host has missed something. Duplicate acknowledgements are discarded on the spot, to avoid having the source misinterpret them as a sign of congestion.

One disadvantage of this transparency, however, is that if the wireless link is very lossy, the source may time out waiting for an acknowledgement and invoke the congestion control algorithm. With indirect TCP, the congestion control algorithm will never be started unless there really is congestion in the wired part of the network.

The Balakrishnan et al. paper also has a solution to the problem of lost segments originating at the mobile host. When the base station notices a gap in the inbound sequence numbers, it generates a request for a selective repeat of the missing bytes using a TCP option. Using these two fixes, the wireless link is made more reliable in both directions, without the source knowing about it, and without changing the semantics of TCP.

While UDP does not suffer from the same problems as TCP, wireless communication also introduces difficulties for it. The main trouble is that programs use UDP expecting it to be highly reliable. They know that no guarantees are given, but they still expect it to be near perfect. In a wireless environment, it will be far from perfect. For programs that are able to recover from lost UDP messages, but only at considerable cost, suddenly going from an environment where messages theoretically can be lost but rarely are, to one in which they are constantly being lost can result in a performance disaster.

Wireless communication also affects areas other than just performance. For example, how does a mobile host find a local printer to connect to, rather than use its home printer? Somewhat related to this is how to get the WWW page for the local cell, even if its name is not known. Also, WWW page designers tend to assume lots of bandwidth is available. Putting a large logo on every page becomes counterproductive if it is going to take 30 sec to transmit at 9600 bps every time the page is referenced, irritating the users no end.

6.5. THE ATM AAL LAYER PROTOCOLS

It is not really clear whether or not ATM has a transport layer. On the one hand, the ATM layer has the functionality of a network layer, and there is another layer on top of it (AAL), which sort of makes AAL a transport layer. Some experts agree with this view (e.g., De Prycker, 1993, page 112). One of the protocols used here (AAL 5) is functionally similar to UDP, which is unquestionably a transport protocol.

On the other hand, none of the AAL protocols provide a reliable end-to-end connection, as TCP does (although with only very minor changes they could). Also, in most applications another transport layer is used on top of AAL. Rather than split hairs, we will discuss the AAL layer and its protocols in this chapter without making a claim that it is a true transport layer.

The AAL layer in ATM networks is radically different than TCP, largely because the designers were primarily interested in transmitting voice and video streams, in which rapid delivery is more important than accurate delivery. Remember that the ATM layer just outputs 53-byte cells one after another. It has no error control, no flow control, and no other control. Consequently, it is not well matched to the requirements that most applications need.

To bridge this gap, in Recommendation I.363, ITU has defined an end-to-end layer on top of the ATM layer. This layer, called **AAL (ATM Adaptation Layer)** has a tortuous history, full of mistakes, revisions, and unfinished business. In the following sections we will look at it and its design.

The goal of AAL is to provide useful services to application programs and to shield them from the mechanics of chopping data up into cells at the source and reassembling them at the destination. When ITU began defining AAL, it realized that different applications had different requirements, so it organized the service space along three axes:

1. Real-time service versus nonreal-time service.
2. Constant bit rate service versus variable bit rate service.
3. Connection-oriented service versus connectionless service.

In principle, with three axes and two values on each axis, eight distinct services can be defined, as shown in Fig. 6-36. ITU felt that only four of these were of any use, and named them classes A, B, C, and D, as noted. The others were not supported. Starting with ATM 4.0, Fig. 6-36 is somewhat obsolete, so it has been presented here mostly as background information to help understand why the

AAL protocols have been designed as they have been. Instead of these service classes, the major distinction now is between the traffic classes we studied in Chap. 5 (ABR, CBR, NRT-VBR, RT-VBR, and UBR).

	A		B	C				D
Timing	Real time	None	Real time	None	Real time	None	Real time	None
Bit rate	Constant		Variable		Constant		Variable	
Mode	Connection orientated				Connectionless			

Fig. 6-36. Original service classes supported by AAL (now obsolete).

To handle these four classes of service, ITU defined four protocols, AAL 1 through AAL 4, respectively. However, later it discovered that the technical requirements for classes C and D were so similar that AAL 3 and AAL 4 were combined into AAL 3/4. Then the computer industry, which had been asleep at the switch, realized that none of them were any good. It solved this problem by the simple expedient of defining another protocol, AAL 5. We will look at all four of these shortly. We will also look at an interesting control protocol used on ATM systems.

6.5.1. Structure of the ATM Adaptation Layer

The ATM adaptation layer is divided into two major parts, one of which is often further subdivided, as illustrated in Fig. 6-37.

The upper part of the ATM adaptation layer is called the **convergence sublayer**. Its job is to provide the interface to the application. It consists of a subpart that is common to all applications (for a given AAL protocol) and an application specific subpart. The functions of each of these parts are protocol dependent but can include message framing and error detection.

In addition, at the source, the convergence sublayer is responsible for accepting bit streams or arbitrary length messages from the applications and breaking them up into units of 44 to 48 bytes for transmission. The exact size is protocol dependent, since some protocols use part of the 48-byte ATM payload for their own headers. At the destination, this sublayer reassembles the cells into the original messages. In general, message boundaries are preserved, when present. In other words, if the source sends four 512-byte messages, they will arrive as four 512-byte messages, not one 2048-byte message. For data streams, no message boundaries exist, so they are not preserved.

The lower part of the AAL is called the **SAR** (**Segmentation And Reassembly**) sublayer. It can add headers and trailers to the data units given to it by the

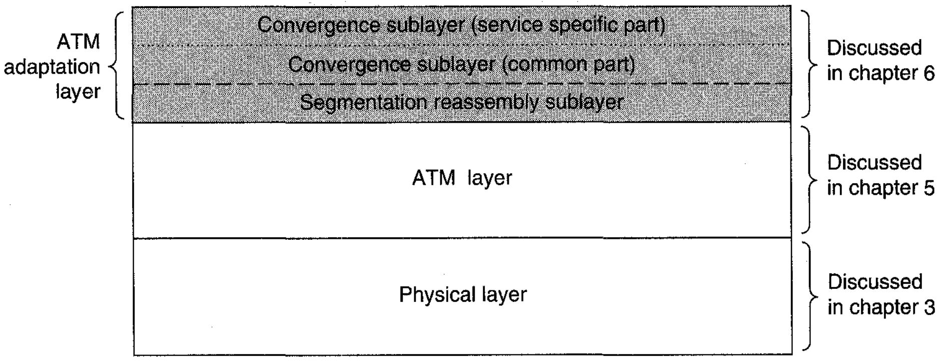

ATM adaptation layer
Convergence sublayer (service specific part)
Convergence sublayer (common part)
Segmentation reassembly sublayer
Discussed in chapter 6
ATM layer
Discussed in chapter 5
Physical layer
Discussed in chapter 3

Fig. 6-37. The ATM model showing the ATM adaptation layer and its sublayers.

convergence sublayer to form cell payloads. These payloads are then given to the ATM layer for transmission. At the destination, the SAR sublayer reassembles the cells into messages. The SAR sublayer is basically concerned with cells, whereas the convergence sublayer is concerned with messages.

The generic operation of the convergence and SAR sublayers is shown in Fig. 6-38. When a message comes in to the AAL from the application, the convergence sublayer may give it a header and/or trailer. The message is then broken up into 44- to 48-byte units, which are passed to the SAR sublayer. The SAR sublayer may add its own header and trailer to each piece and pass them down to the ATM layer for transmission as independent cells. Note that the figure shows the most general case because some of the AAL protocols have null headers and/or trailers.

The SAR sublayer also has some additional functions for some (but not all) service classes. In particular, it sometimes handles error detection and multiplexing. The SAR sublayer is present for all service classes but does more or less work, depending on the specific protocol.

The communication between the application and AAL layer uses the standard OSI *request* and *indication* primitives that we discussed in Chap. 1. The communication between the sublayers uses different primitives.

6.5.2. AAL 1

AAL 1 is the protocol used for transmitting class A traffic, that is, real-time, constant bit rate, connection-oriented traffic, such as uncompressed audio and video. Bits are fed in by the application at a constant rate and must be delivered at the far end at the same constant rate, with a minimum of delay, jitter, and overhead. The input is a stream of bits, with no message boundaries. For this traffic, error detecting protocols such as stop-and-wait are not used because the delays that are introduced by timeouts and retransmissions are unacceptable. However,

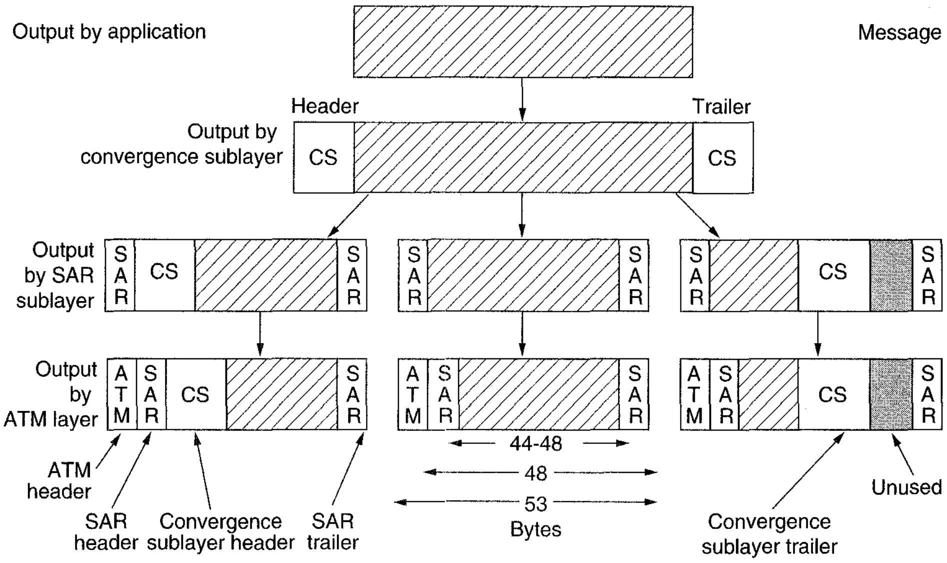


Fig. 6-38. The headers and trailers that can be added to a message in an ATM network.

missing cells are reported to the application, which must then take its own action (if any) to recover from them.

AAL 1 uses a convergence sublayer and a SAR sublayer. The convergence sublayer detects lost and misinserted cells. (A misinserted cell is one that is delivered to the wrong destination as a result of an undetected error in its virtual circuit or virtual path identifiers.) It also smoothes out incoming traffic to provide delivery of cells at a constant rate. Finally, the convergence sublayer breaks up the input messages or stream into 46- or 47-byte units that are given to the SAR sublayer for transmission. At the other end it extracts these and reconstructs the original input. The AAL 1 convergence sublayer does not have any protocol headers of its own.

In contrast, the AAL 1 SAR sublayer does have a protocol. The formats of its cells are given in Fig. 6-39. Both formats begin with a 1-byte header containing a 3-bit cell sequence number, *SN*, (to detect missing or misinserted cells). This field is followed by a 3-bit sequence number protection, *SNP*, (i.e., checksum) over the sequence number to allow correction of single errors and detection of double errors in the sequence field. It uses a cyclic redundancy check with the polynomial $x^3 + x + 1$. An even parity bit covering the header byte further reduces the likelihood of a bad sequence number sneaking in unnoticed. AAL 1 cells need not be filled with a full 47 bytes. For example, to transmit digitized voice arriving at a rate of 1 byte every 125 μsec, filling a cell with 47 bytes means collecting samples for 5.875 msec. If this delay before transmission is

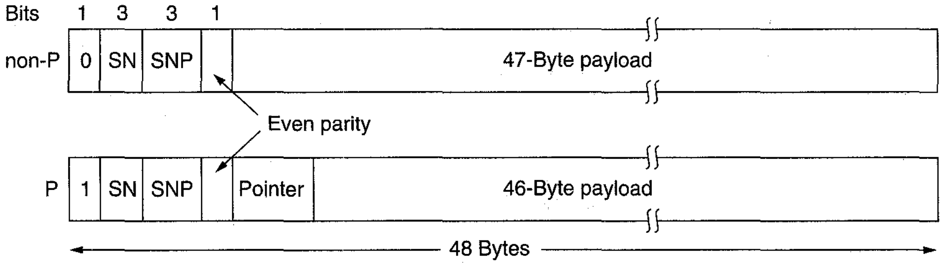


Fig. 6-39. The AAL 1 cell format.

unacceptable, partial cells can be sent. In this case, the number of actual data bytes per cell is the same for all cells and agreed on in advance.

The *P* cells are used when message boundaries must be preserved. The *Pointer* field is used to give the offset of the start of the next message. Only cells with an even sequence number may be *P* cells, so the pointer is in the range 0 to 92, to put it within the payload of either its own cell or the one following it. Note that this scheme allows messages to be an arbitrary number of bytes long, so messages can be run continuously and need not align on cell boundaries.

The high-order bit of the *Pointer* field is reserved for future use. The initial header bit of all the odd-numbered cells forms a data stream used for clock synchronization.

6.5.3. AAL 2

AAL 1 is designed for simple, connection-oriented, real-time data streams without error detection, except for missing and misinserted cells. For pure uncompressed audio or video, or any other data stream in which having a few garbled bits once in a while is not a problem, AAL 1 is adequate.

For compressed audio or video, the rate can vary strongly in time. For example, many compression schemes transmit a full video frame periodically and then send only the differences between subsequent frames and the last full frame for several frames. When the camera is stationary and nothing is moving, the difference frames are small, but when the camera is panning rapidly, they are large. Also, message boundaries must be preserved so that the start of the next full frame can be recognized, even in the presence of lost cells or bad data. For these reasons, a fancier protocol is needed. AAL 2 has been designed for this purpose.

As in AAL 1, the CS sublayer does not have a protocol but the SAR sublayer does. The SAR cell format is shown in Fig. 6-40. It has a 1-byte header and a 2-byte trailer, leaving room for up to 45 data bytes per cell.

The *SN* field (*Sequence Number*) is used for numbering cells in order to detect missing or misinserted cells. The *IT* field (*Information Type*) is used to indicate

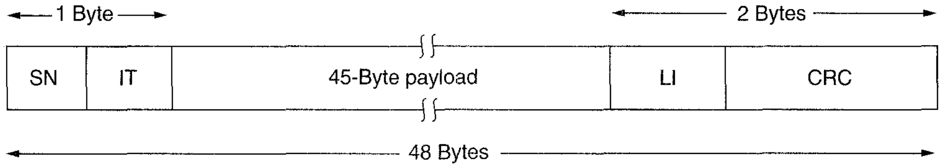


Fig. 6-40. The AAL 2 cell format.

that the cell is the start, middle, or end of a message. The *LI* (*Length indicator*) field tells how big the payload is, in bytes (it might be less than 45 bytes). Finally, the *CRC* field is a checksum over the entire cell, so errors can be detected.

Strange as it may sound, the field sizes are not included in the standard. According to one insider, at the very end of the standardization process the committee realized that AAL 2 had so many problems that it should not be used. Unfortunately, it was too late to stop the standardization process. They had a deadline to meet. In a last ditch effort, the committee removed all the field sizes so that the formal standard could be issued on time, but in such a way that nobody could actually use it. Such is life in the world of standardization.

6.5.4. AAL 3/4

Originally, ITU had different protocols for classes C and D, connection-oriented service and connectionless service for data transport that is sensitive to loss or errors but is not time dependent. Then ITU discovered that there was no real need for two protocols, so they were combined into a single protocol, AAL 3/4.

AAL 3/4 can operate in two modes: stream or message. In message mode, each call from the application to AAL 3/4 injects one message into the network. The message is delivered as such, that is, message boundaries are preserved. In stream mode the boundaries are not preserved. The discussion below will concentrate on message mode. Reliable and unreliable (i.e., no guarantee) transport are available in each mode.

A feature of AAL 3/4 not present in any of the other protocols is multiplexing. This aspect of AAL 3/4 allows multiple sessions (e.g., remote logins) from a single host to travel along the same virtual circuit and be separated at the destination, as illustrated in Fig. 6-41.

The reason that this facility is desirable is that carriers often charge for each connection setup and for each second that a connection is open. If a pair of hosts have several sessions open simultaneously, giving each one its own virtual circuit will be more expensive than multiplexing all of them onto the same virtual circuit. If one virtual circuit has sufficient bandwidth to handle the job, there is no need

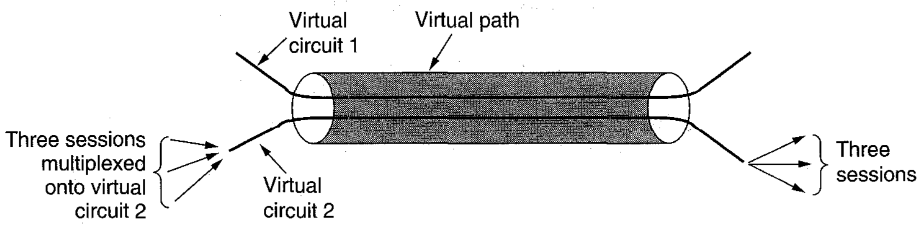
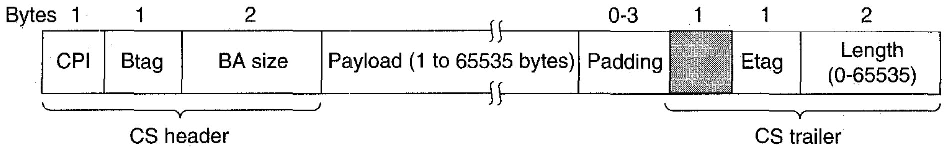


Fig. 6-41. Multiplexing of several sessions onto one virtual circuit.

for more than one. All sessions using a single virtual circuit get the same quality of service, since this is negotiated per virtual circuit.

This issue is the real reason that there were originally separate AAL 3 and AAL 4 formats: the Americans wanted multiplexing and the Europeans did not. So each group went off and made its own standard. Eventually, the Europeans decided that saving 10 bits in the header was not worth the price of having the United States and Europe not be able to communicate. For the same money, they could have stuck to their guns and we would have had four incompatible AAL standards (of which one is broken) instead of three.

Unlike AAL 1 and AAL 2, AAL 3/4 has both a convergence sublayer protocol and a SAR sublayer protocol. Messages as large as 65,535 bytes come into the convergence sublayer from the application. These are first padded out to a multiple of 4 bytes. Then a header and a trailer are attached, as shown in Fig. 6-42.



Bytes	1	1	2		0-3	1	1	2
	CPI	Btag	BA size	Payload (1 to 65535 bytes)	Padding	▨	Etag	Length (0-65535)

Fig. 6-42. AAL 3/4 convergence sublayer message format.

The *CPI* field (*Common Part Indicator*) gives the message type and the counting unit for the *BA size* and *Length* fields. The *Btag* and *Etag* fields are used to frame messages. The two bytes must be the same and are incremented by one on every new message sent. This mechanism checks for lost or misinserted cells. The *BA size* field is used for buffer allocation. It tells the receiver how much buffer space to allocate for the message in advance of its arrival. The *Length* field gives the payload length again. In message mode, it must be equal to *BA size*, but in stream mode it may be different. The trailer also contains 1 unused byte.

After the convergence sublayer has constructed and added a header and trailer to the message, as shown in Fig. 6-42, it passes the message to the SAR sublayer, which chops the message up into 44-byte chunks. Note that to support multiplexing, the convergence sublayer may have several messages constructed internally at once and may pass 44-byte chunks to the SAR sublayer first from one message, then from another, in any order.

The SAR sublayer inserts each 44-byte chunk into the payload of a cell whose format is shown in Fig. 6-43. These cells are then transmitted to the destination for reassembly, after which checksum verification is performed and action taken if need be.

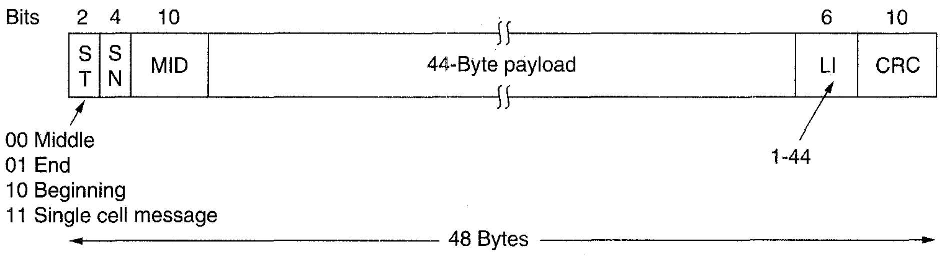


Fig. 6-43. The AAL 3/4 cell format.

The fields in the AAL 3/4 cell are as follows. The *ST* (*Segment Type*) field is used for message framing. It indicates whether the cell begins a message, is in the middle of a message, is the last cell of a message, or is a small (i.e., single cell) message. Next comes a 4-bit sequence number, *SN*, for detecting missing and misinserted cells. The *MID* (*Multiplexing ID*) field is used to keep track of which cell belongs to which session. Remember that the convergence sublayer may have several messages, belonging to different sessions, buffered at once, and it may send pieces of these messages in whatever order it wishes. All the pieces from messages belonging to session *i* carry *i* in the *MID* field, so they can be correctly reassembled at the destination. The trailer contains the payload length and cell checksum.

Notice that AAL 3/4 has two layers of protocol overhead: 8 bytes are added to every message and 4 bytes are added to every cell. All in all, it is a heavyweight mechanism, especially for short messages.

6.5.5. AAL 5

The AAL 1 through AAL 3/4 protocols were largely designed by the telecommunications industry and standardized by ITU without a lot of input from the computer industry. When the computer industry finally woke up and began to understand the implications of Fig. 6-43, a sense of panic set in. The complexity and inefficiency generated by two layers of protocol, coupled with the surprisingly short checksum (only 10 bits), caused some researchers to invent a new adaptation protocol. It was called **SEAL** (**Simple Efficient Adaptation Layer**), which suggests what the designers thought of the old ones. After some discussion, the ATM Forum accepted SEAL and assigned it the name AAL 5. For more information about AAL 5 and how it differs from AAL 3/4, see (Suzuki, 1994).

AAL 5 offers several kinds of service to its applications. One choice is reliable service (i.e., guaranteed delivery with flow control to prevent overruns). Another choice is unreliable service (i.e., no guaranteed delivery), with options to have cells with checksum errors either discarded or passed to the application anyway (but reported as bad). Both unicast and multicast are supported, but multicast does not provide guaranteed delivery.

Like AAL 3/4, AAL 5 supports both message mode and stream mode. In message mode, an application can pass a datagram of length 1 to 65,535 bytes to the AAL layer and have it delivered to the destination, either on a guaranteed or a best efforts basis. Upon arrival in the convergence sublayer, a message is padded out and a trailer added, as shown in Fig. 6-44. The amount of padding (0 to 47 bytes) is chosen to make the entire message, including the padding and trailer, be a multiple of 48 bytes. AAL 5 does not have a convergence sublayer header, just an 8-byte trailer.

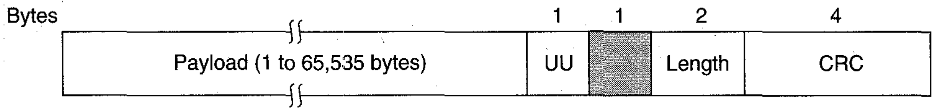


Fig. 6-44. AAL 5 convergence sublayer message format.

The *UU* (*User to User*) field is not used by the AAL layer itself. Instead, it is available for a higher layer for its own purposes, for example, sequencing or multiplexing. The higher layer in question may be the service-specific subpart of the convergence sublayer. The *Length* field tells how long the true payload is, in bytes, not counting the padding. A value of 0 is used to abort the current message in midstream. The *CRC* field is the standard 32-bit checksum over the entire message, including the padding and the trailer (with the *CRC* field set to 0). One 8-bit field in the trailer is reserved for future use.

The message is transmitted by passing it to the SAR sublayer, which does not add any headers or trailers. Instead, it breaks the message into 48-byte units and passes each of these to the ATM layer for transmission. It also tells the ATM layer to set a bit in the *PTI* field on the last cell, so message boundaries are preserved. A case can be made that this is an incorrect mixing of protocol layers because the AAL layer should not be using bits in the ATM layer's header. Doing so violates the most basic principle of protocol engineering, and suggests the layering should have perhaps been done differently.

The principal advantage of AAL 5 over AAL 3/4 is the much greater efficiency. While AAL 3/4 adds only 4 bytes per message, it also adds 4 bytes per cell, reducing the payload capacity to 44 bytes, a loss of 8 percent on long messages. AAL 5 has a slightly large trailer per message (8 bytes) but has no overhead in each cell. The lack of sequence numbers in the cells is compensated for by the longer checksum, which can detect lost, misinserted, or missing cells without using sequence numbers.

Within the Internet community, it is expected that the normal way of interfacing to ATM networks will be to transport IP packets with the AAL 5 payload field. Various issues relating to this approach are discussed in RFC 1483 and RFC 1577.

6.5.6. Comparison of AAL Protocols

The reader is hereby forgiven if he or she thinks that the various AAL protocols seem unnecessarily similar to one another and poorly thought out. The value of having distinct convergence and SAR sublayers is also questionable, especially since AAL 5 does not have anything in the SAR sublayer. A slightly enhanced ATM layer header could have provided for sequencing, multiplexing, and framing quite adequately.

Some of the differences between the various AAL protocols are summarized in Fig. 6-45. These relate to efficiency, error handling, multiplexing, and the relation between the AAL sublayers.

Item	AAL 1	AAL 2	AAL 3/4	AAL 5
Service class	A	B	C/D	C/D
Multiplexing	No	No	Yes	No
Message delimiting	None	None	Btag/Etag	Bit in PTI
Advance buffer allocation	No	No	Yes	No
User bytes available	0	0	0	1
CS padding	0	0	32-Bit word	0–47 bytes
CS protocol overhead (bytes)	0	0	8	8
CS checksum	None	None	None	32 Bits
SAR payload bytes	46–47	45	44	48
SAR protocol overhead (bytes)	1–2	3	4	0
SAR checksum	None	None	10 Bits	None

Fig. 6-45. Some differences between the various AAL protocols.

The overall impression that AAL gives is of too many variants with too many minor differences and a job half done. The original four service classes, A, B, C, D, have been effectively abandoned. AAL 1 is probably not really necessary;

AAL 2 is broken; AAL 3 and AAL 4 never saw the light of day; and AAL 3/4 is inefficient and has too short a checksum.

The future lies with AAL 5, but even here there is room for improvement. AAL 5 messages should have had a sequence number and a bit to distinguish data from control messages, so it could have been used as a reliable transport protocol. Unused space in the trailer was even available for them. As it stands, for reliable transport, the additional overhead of a transport layer is required on top of it, when it could have been avoided. If the full AAL committee had turned its work in as a class project, the professor would probably have given it back with instructions to fix it and turn it in again when it was finished. More criticism of ATM can be found in (Sterbenz et al., 1995).

6.5.7. SSCOP—Service Specific Connection-Oriented Protocol

Despite all these different AAL protocols, none of them provides for simple end-to-end reliable transport connections. For applications where that is required, another AAL protocol exists: **SSCOP** (**Service Specific Connection Oriented Protocol**). However, SSCOP is only used for control, not for data transmission.

SSCOP users send messages, each of which is assigned a 24-bit sequence number. Messages can be up to 64K bytes and are not fragmented. They must be delivered in order. Unlike some other reliable transport protocols, missing messages are always retransmitted using selective repeat rather than go back n.

SSCOP is fundamentally a dynamic sliding window protocol. For each connection, the receiver maintains a window of message sequence numbers that it is prepared to receive, and a bit map marking the ones it already has. This window can change size during protocol operation.

What makes SSCOP unusual is the way acknowledgements are handled: there is no piggybacking. Instead, periodically, the sender polls the receiver and asks it to send back the bit map giving the window status. Based on the result, the sender discards messages that have been accepted and updates its window. SSCOP is described in detail in (Henderson, 1995).

6.6. PERFORMANCE ISSUES

Performance issues are very important in computer networks. When hundreds or thousands of computers are connected together, complex interactions, with unforeseen consequences, are common. Frequently, this complexity leads to poor performance and no one knows why. In the following sections, we will examine many issues related to network performance to see what kinds of problems exist and what can be done about them.

Unfortunately, understanding network performance is more of an art than a science. There is little underlying theory that is actually of any use in practice.

The best we can do is give rules of thumb gained from hard experience and present examples taken from the real world. We have intentionally delayed this discussion until after studying the transport layer in TCP and ATM networks in order to be able to point out places where they have done things right or done things wrong.

The transport layer is not the only place performance issues arise. We saw some of them in the network layer in the previous chapter. Nevertheless, the network layer tends to be largely concerned with routing and congestion control. The broader, system-oriented issues tend to be transport related, so this chapter is an appropriate place to examine them.

In the next five sections, we will look at five aspects of network performance:

1. Performance problems.
2. Measuring network performance.
3. System design for better performance.
4. Fast TPDU processing.
5. Protocols for future high-performance networks.

As an aside, we need a name for the units exchanged by transport entities. The TCP term, segment, is confusing at best and is never used outside the TCP world in this context. The proper ATM terms, CS-PDU, SAR-PDU, and CPCS-PDU, are specific to ATM. Packets clearly refer to the network layer and messages belong to the application layer. For lack of a standard term, we will go back to calling the units exchanged by transport entities TPDUs. When we mean both TPDU and packet together, we will use packet as the collective term, as in "The CPU must be fast enough to process incoming packets in real time." By this we mean both the network layer packet and the TPDU encapsulated in it.

6.6.1. Performance Problems in Computer Networks

Some performance problems, such as congestion, are caused by temporary resource overloads. If more traffic suddenly arrives at a router than the router can handle, congestion will build up and performance will suffer. We studied congestion in detail in the previous chapter.

Performance also degrades when there is a structural resource imbalance. For example, if a gigabit communication line is attached to a low-end PC, the poor CPU will not be able to process the incoming packets fast enough, and some will be lost. These packets will eventually be retransmitted, adding delay, wasting bandwidth, and generally reducing performance.

Overloads can also be synchronously triggered. For example, if a TPDU contains a bad parameter (e.g., the port or process for which it is destined), in many cases the receiver will thoughtfully send back an error notification. Now consider what could happen if a bad TPDU is broadcast to 10,000 machines: each one might send back an error message. The resulting **broadcast storm** could cripple the network. UDP suffered from this problem until the protocol was changed to cause hosts to refrain from responding to errors in UDP TPDUs sent to broadcast addresses.

A second example of synchronous overload is what happens after an electrical power failure. When the power comes back on, all the machines simultaneously jump to their ROMs to start rebooting. A typical reboot sequence might require first going to some (RARP) server to learn one's true identity, and then to some file server to get a copy of the operating system. If hundreds of machines all do this at once, the server will probably collapse under the load.

Even in the absence of synchronous overloads and when there are sufficient resources available, poor performance can occur due to lack of system tuning. For example, if a machine has plenty of CPU power and memory, but not enough of the memory has been allocated for buffer space, overruns will occur and TPDUs will be lost. Similarly, if the scheduling algorithm does not give a high enough priority to processing incoming TPDUs, some of them may be lost.

Another tuning issue is setting timeouts correctly. When a TPDU is sent, a timer is typically set to guard against its loss. If the timeout is set too short, unnecessary retransmissions will occur, clogging the wires. If the timeout is set too long, unnecessary delays will occur after a TPDU is lost. Other tunable parameters include how long to wait for data to piggyback onto before sending a separate acknowledgement and the number of retransmissions before giving up.

Gigabit networks bring with them new performance problems. Consider, for example, sending data from San Diego to Boston when the receiver's buffer is 64K bytes. Suppose that the link is 1 Gbps and the one-way speed-of-light-in-fiber delay is 20 msec. Initially, at $t = 0$, the pipe is empty, as illustrated in Fig. 6-46(a). Only 500 μsec later, in Fig. 6-46(b), all the TPDUs are out on the fiber. The lead TPDU will now be somewhere in the vicinity of Brawley, still deep in Southern California. However, the transmitter must stop until it gets a window update.

After 20 msec, the lead TPDU hits Boston, as shown in Fig. 6-46(c) and is acknowledged. Finally, 40 msec after starting, the first acknowledgement gets back to the sender and the second burst can be transmitted. Since the transmission line was used for 0.5 msec out of 40, the efficiency is about 1.25 percent. This situation is typical of running older protocols over gigabit lines.

A useful quantity to keep in mind when analyzing network performance is the **bandwidth-delay product**. It is obtained by multiplying the bandwidth (in bits/sec) by the round-trip delay time (in sec). The product is the capacity of the pipe from the sender to the receiver and back (in bits).

For the example of Fig. 6-46 the bandwidth-delay product is 40 million bits. In other words, the sender would have to transmit a burst of 40 million bits to be

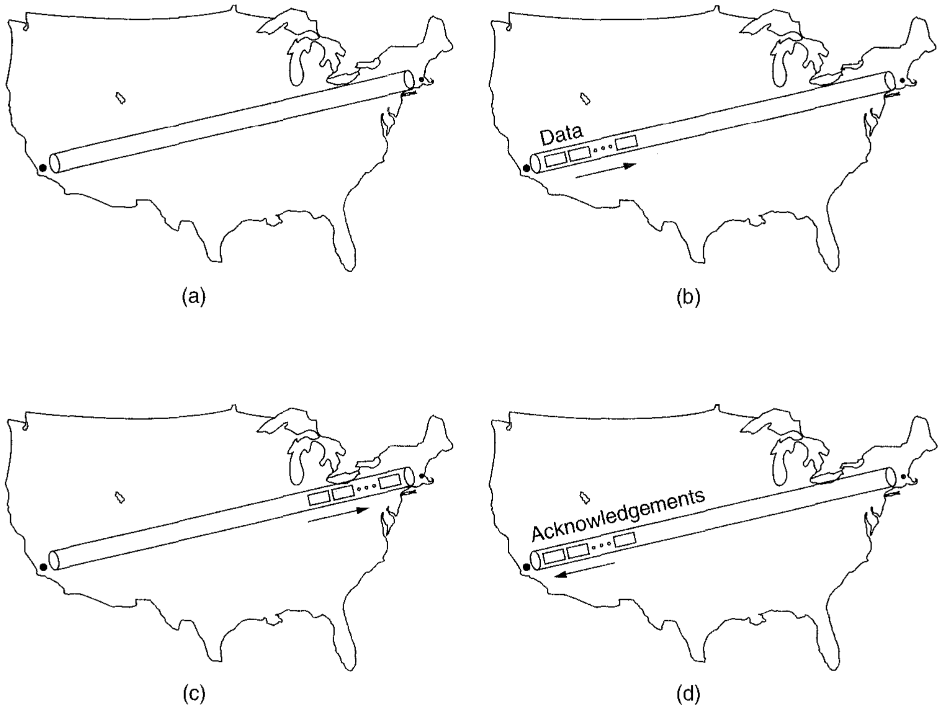


Fig. 6-46. The state of transmitting one megabit from San Diego to Boston. (a) At $t = 0$. (b) After 500 μsec. (c) After 20 msec. (d) After 40 msec.

able to keep going full speed until the first acknowledgement came back. It takes this many bits to fill the pipe (in both directions). This is why a burst of half a million bits only achieves a 1.25 percent efficiency: it is only 1.25 percent of the pipe capacity.

The conclusion to be drawn here is that to achieve good performance, the receiver's window must be at least as large as the bandwidth-delay product, preferably somewhat larger since the receiver may not respond instantly. For a transcontinental gigabit line, at least 5 megabytes are required for each connection.

If the efficiency is terrible for sending a megabit, imagine what it is like when sending a few hundred bytes for a remote procedure call. Unless some other use can be found for the line while the first client is waiting for its reply, a gigabit line is no better than a megabit line, just more expensive.

Another performance problem that occurs with time-critical applications like audio and video is jitter. Having a short mean transmission time is not enough. A small standard deviation is also required. Achieving a short mean transmission time along with a small standard deviation demands a serious engineering effort.

6.6.2. Measuring Network Performance

When a network performs poorly, its users often complain to the folks running it, demanding improvements. To improve the performance, the operators must first determine exactly what is going on. To find out what is really happening, the operators must make measurements. In this section we will look at network performance measurements. The discussion below is based on the work of Mogul (1993). For a more thorough discussion of the measurement process, see (Jain, 1991; and Villamizan and Song, 1995).

The basic loop used to improve network performance contains the following steps:

1. Measure the relevant network parameters and performance.
2. Try to understand what is going on.
3. Change one parameter.

These steps are repeated until the performance is good enough or it is clear that the last drop of improvement has been squeezed out.

Measurements can be made in many ways and at many locations (both physically and in the protocol stack). The most basic kind of measurement is to start a timer when beginning some activity and use it to see how long that activity takes. For example, knowing how long it takes for a TPDU to be acknowledged is a key measurement. Other measurements are made with counters that record how often some event has happened (e.g., number of lost TPDUs). Finally, one is often interested in knowing the amount of something, such as the number of bytes processed in a certain time interval.

Measuring network performance and parameters has many potential pitfalls. Below we list a few of them. Any systematic attempt to measure network performance should be careful to avoid these.

Make Sure that the Sample Size Is Large Enough

Do not measure the time to send one TPDU, but repeat the measurement, say, one million times and take the average. Having a large sample will reduce the uncertainty in the measured mean and standard deviation. This uncertainty can be computed using standard statistical formulas.

Make Sure that the Samples Are Representative

Ideally, the whole sequence of one million measurements should be repeated at different times of the day and the week to see the effect of different system loads on the measured quantity. Measurements of congestion, for example, are of little use if they are made at a moment when there is no congestion. Sometimes the results may be counterintuitive at first, such as heavy congestion at 10, 11, 1, and 2 o'clock, but no congestion at noon (when all the users are away at lunch).

Be Careful When Using a Coarse-Grained Clock

Computer clocks work by adding one to some counter at regular intervals. For example, a millisecond timer adds one to a counter every 1 msec. Using such a timer to measure an event that takes less than 1 msec is not impossible, but requires some care.

To measure the time to send a TPDU, for example, the system clock (say, in milliseconds) should be read out when the transport layer code is entered, and again when it is exited. If the true TPDU send time is 300 μsec, the difference between the two readings will be either 0 or 1, both wrong. However, if the measurement is repeated one million times and the total of all measurements added up and divided by one million, the mean time will be accurate to better than 1 μsec.

Be Sure that Nothing Unexpected Is Going On during Your Tests

Making measurements on a university system the day some major lab project has to be turned in may give different results than if made the next day. Likewise, if some researcher has decided to run a video conference over your network during your tests, you may get a biased result. It is best to run tests on an idle system and create the entire workload yourself. Even this approach has pitfalls though. While you might think nobody will be using the network at 3 A.M., that might be precisely when the automatic backup program begins copying all the disks to videotape. Furthermore, there might be heavy traffic for your wonderful World Wide Web pages from distant time zones.

Caching Can Wreak Havoc with Measurements

To measure file transfer times, the obvious way to do it is to open a large file, read the whole thing, close it, and see how long it takes. Then repeat the measurement many more times to get a good average. The trouble is, the system may cache the file, so that only the first measurement actually involves network traffic. The rest are just reads from the local cache. The results from such a measurement are essentially worthless (unless you want to measure cache performance).

Often you can get around caching by simply overflowing the cache. For example, if the cache is 10 MB, the test loop could open, read, and close two 10-MB files on each pass, in an attempt to force the cache hit rate to 0. Still, caution is advised unless you are absolutely sure you understand the caching algorithm.

Buffering can have a similar effect. One popular TCP/IP performance utility program has been known to report that UDP can achieve a performance substantially higher than the physical line allows. How does this occur? A call to UDP normally returns control as soon as the message has been accepted by the kernel and added to the transmission queue. If there is sufficient buffer space, timing 1000 UDP calls does not mean that all the data have been sent. Most of them may still be in the kernel, but the performance utility thinks they have all been transmitted.

Understand What You Are Measuring

When you measure the time to read a remote file, your measurements depend on the network, the operating systems on both the client and server, the particular hardware interface boards used, their drivers, and other factors. If done carefully, you will ultimately discover the file transfer time for the configuration you are using. If your goal is to tune this particular configuration, these measurements are fine.

However, if you are making similar measurements on three different systems in order to choose which network interface board to buy, your results could be thrown off completely by the fact that one of the network drivers is truly awful and is only getting 10 percent of the performance of the board.

Be Careful about Extrapolating the Results

Suppose that you make measurements of something with simulated network loads running from 0 (idle) to 0.4 (40 percent of capacity), as shown by the data points and solid line through them in Fig. 6-47. It may be tempting to extrapolate linearly, as shown by the dotted line. However, many queueing results involve a factor of $1/(1-\rho)$, wher ρ is the load, so the true values may look more like the dashed line.

6.6.3. System Design for Better Performance

Measuring and tinkering can often improve performance considerably, but they cannot substitute for good design in the first place. A poorly designed network can be improved only so much. Beyond that, it has to be redone from scratch.

In this section, we will present some rules of thumb based on experience with many networks. These rules relate to system design, not just network design, since the software and operating system are often more important than the routers and interface boards. Most of these ideas have been common knowledge to network designers for years and have been passed on from generation to generation by word of mouth. They were first stated explicitly by Mogul (1993); our treatment largely parallels his. Another relevant source is (Metcalfe, 1993).

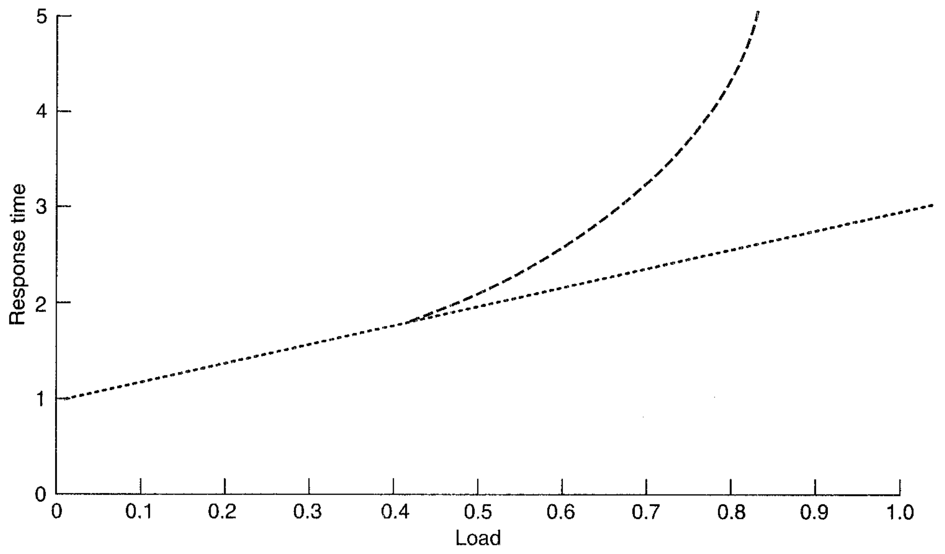


Fig. 6-47. Response as a function of load.

Rule #1: CPU Speed Is More Important than Network Speed

Long experience has shown that in nearly all networks, operating system and protocol overhead dominates actual time on the wire. For example, in theory, the minimum RPC time on an Ethernet is 102 μsec, corresponding to a minimum (64-byte) request followed by a minimum (64-byte) reply. In practice, getting the RPC time down to 1500 μsec is a considerable achievement (Van Renesse et al., 1988). Note that 1500 μsec is 15 times worse than the theoretical minimum. Nearly all the overhead is in the software.

Similarly, the biggest problem in running at 1 Gbps is getting the bits from the user's buffer out onto the fiber fast enough and having the receiving CPU process them as fast as they come in. In short, if you double the CPU speed, you often can come close to doubling the throughput. Doubling the network capacity often has no effect since the bottleneck is generally in the hosts.

Rule #2: Reduce Packet Count to Reduce Software Overhead

Processing a TPDU has a certain amount of overhead per TPDU (e.g., header processing) and a certain amount of processing per byte (e.g., doing the checksum). When sending 1 million bytes, the per-byte overhead is the same no matter what the TPDU size is. However, using 128-byte TPDUs means 32 times as much per-TPDU overhead as using 4K TPDUs. This overhead adds up fast.

In addition to the TPDU overhead, there is overhead in the lower layers to consider. Each arriving packet causes an interrupt. On a modern RISC processor, each interrupt breaks the CPU pipeline, interferes with the cache, requires a change to the memory management context, and forces a substantial number of CPU registers to be saved. An *n*-fold reduction in TPDUs sent thus reduces the interrupt and packet overhead by a factor of *n*.

This observation argues for collecting a substantial amount of data before transmission in order to reduce interrupts at the other side. Nagle's algorithm and Clark's solution to the silly window syndrome are attempts to do precisely this.

Rule #3: Minimize Context Switches

Context switches (e.g., from kernel mode to user mode) are deadly. They have the same bad properties as interrupts, the worst being a long series of initial cache misses. Context switches can be reduced by having the library procedure that sends data do internal buffering until it has a substantial amount of them. Similarly, on the receiving side, small incoming TPDUs should be collected together and passed to the user in one fell swoop instead of individually to minimize context switches.

In the best case, an incoming packet causes a context switch from the current user to the kernel, and then a switch to the receiving process to give it the newly-arrived data. Unfortunately, with many operating systems, additional context switches happen. For example, if the network manager runs as a special process in user space, a packet arrival is likely to cause a context switch from the current user to the kernel, then another one from the kernel to the network manager followed by another one back to the kernel, and finally one from the kernel to the receiving process. This sequence is shown in Fig. 6-48. All these context switches on each packet are very wasteful of CPU time and will have a devastating effect on network performance.

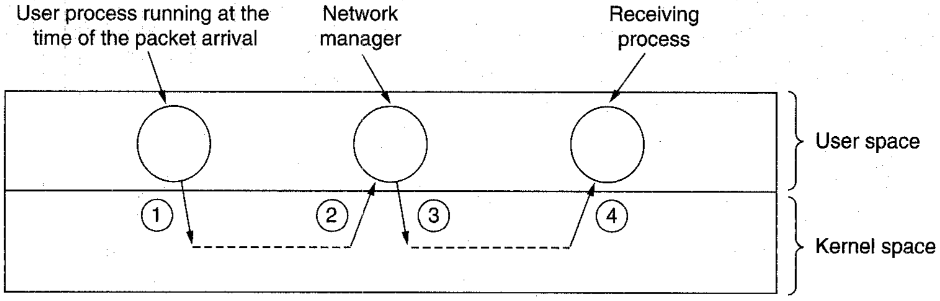


Fig. 6-48. Four context switches to handle one packet with a user-space network manager.

Rule #4: Minimize Copying

Even worse than multiple context switches is making multiple copies. It is not unusual for an incoming packet to be copied three or four times before the TPDU enclosed in it is delivered. After a packet is received by the network interface in a special on-board hardware buffer, it is typically copied to a kernel buffer. From there it is copied to a network layer buffer, then to a transport layer buffer, and finally to the receiving application process.

A clever operating system will copy a word at a time, but it is not unusual to require about five instructions per word (a load, a store, incrementing an index register, a test for end-of-data, and a conditional branch). On a 50-MIPS machine, making three copies of each packet at five instructions per 32-bit word copied requires 75 nsec per incoming byte. Such a machine can thus accept data at a maximum rate of about 107 Mbps. When overhead for header processing, interrupt handling, and context switches is factored in, 50 Mbps might be achievable, and we have not even considered the actual processing of the data. Clearly, handling a 1-Gbps line is out of the question.

In fact, probably a 50-Mbps line is out of the question, too. In the computation above, we have assumed that a 50-MIPS machine can execute any 50 million instructions/sec. In reality, machines can only run at such speeds if they are not referencing memory. Memory operations are often a factor of three slower than register-register instructions, so actually getting 16 Mbps out of the 1 Gbps line might be considered pretty good. Note that hardware assistance will not help here. The problem is too much copying by the operating system.

Rule #5: You Can Buy More Bandwidth but Not Lower Delay

The next three rules deal with communication, rather than protocol processing. The first rule states that if you want more bandwidth, you can just buy it. Putting a second fiber next to the first one doubles the bandwidth but does nothing to reduce the delay. Making the delay shorter requires improving the protocol software, the operating system, or the network interface. Even if all of these are done, the delay will not be reduced if the bottleneck is the transmission time.

Rule #6: Avoiding Congestion Is Better than Recovering from It

The old maxim that an ounce of prevention is worth a pound of cure certainly holds for network congestion. When a network is congested, packets are lost, bandwidth is wasted, useless delays are introduced, and more. Recovering from it takes time and patience. Not having it occur in the first place is better. Congestion avoidance is like getting your DTP vaccination: it hurts a little at the time you get it, but it prevents something that would hurt a lot more.

Rule #7: Avoid Timeouts

Timers are necessary in networks, but they should be used sparingly and timeouts should be minimized. When a timer goes off, some action is generally repeated. If it is truly necessary to repeat the action, so be it, but repeating it unnecessarily is wasteful.

The way to avoid extra work is to be careful that timers are set a little bit on the conservative side. A timer that takes too long to expire adds a small amount of extra delay to one connection in the (unlikely) event of a TPDU being lost. A timer that goes off when it should not have uses up scarce CPU time, wastes bandwidth, and puts extra load on perhaps dozens of routers for no good reason.

6.6.4. Fast TPDU Processing

The moral of the story above is that the main obstacle to fast networking is protocol software. In this section we will look at some ways to speed up this software. For more information, see (Clark et al., 1989; Edwards and Muir, 1995; and Chandranmenon and Varghese, 1995).

TPDU processing overhead has two components: overhead per TPDU and overhead per byte. Both must be attacked. The key to fast TPDU processing is to separate out the normal case (one-way data transfer) and handle it specially. Although a sequence of special TPDUs are needed to get into the *ESTABLISHED* state, once there, TPDU processing is straightforward until one side starts to close the connection.

Let us begin by examining the sending side in the *ESTABLISHED* state when there are data to be transmitted. For the sake of clarity, we assume here that the transport entity is in the kernel, although the same ideas apply if it is a user-space process or a library inside the sending process. In Fig. 6-49, the sending process traps into the kernel to do the SEND. The first thing the transport entity does is make a test to see if this is the normal case: the state is *ESTABLISHED*, neither side is trying to close the connection, a regular (i.e., not an out-of-band) full TPDU is being sent, and there is enough window space available at the receiver. If all conditions are met, no further tests are needed and the fast path through the sending transport entity can be taken.

In the normal case, the headers of consecutive data TPDUs are almost the same. To take advantage of this fact, a prototype header is stored within the transport entity. At the start of the fast path, it is copied as fast as possible to a scratch buffer, word by word. Those fields that change from TPDU to TPDU are then overwritten in the buffer. Frequently, these fields are easily derived from state variables, such as the next sequence number. A pointer to the full TPDU header plus a pointer to the user data are then passed to the network layer. Here the same strategy can be followed (not shown in Fig. 6-49). Finally, the network layer gives the resulting packet to the data link layer for transmission.

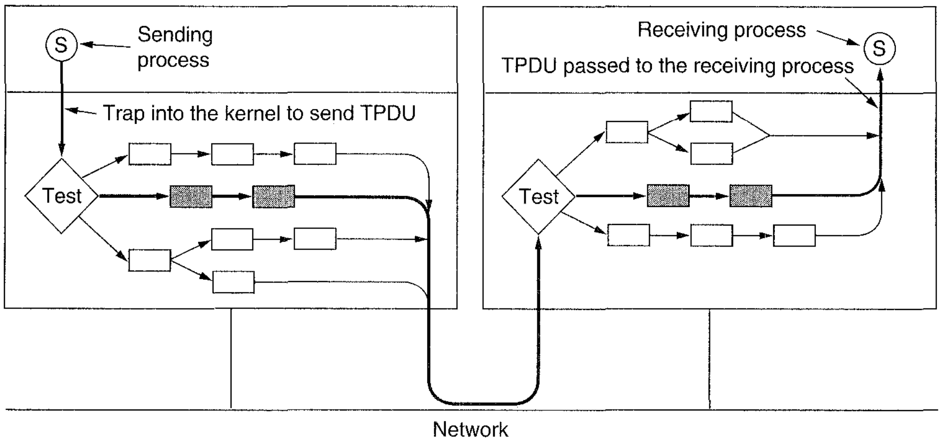

S
Sending process
Trap into the kernel to send TPDU
Test
Receiving process
S
TPDU passed to the receiving process
Test
Network

Fig. 6-49. The fast path from sender to receiver is shown with a heavy line. The processing steps on this path are shaded.

As an example of how this principle works in practice, let us consider TCP/IP. Fig. 6-50(a) shows the TCP header. The fields that are the same between consecutive TPDUs on a one-way flow are shaded. All the sending transport entity has to do is copy the five words from the prototype header into the output buffer, fill in the next sequence number (by copying it from a word in memory), compute the checksum, and increment the sequence number in memory. It can then hand the header and data to a special IP procedure for sending a regular, maximum TPDU. IP then copies its five-word prototype header [see Fig. 6-50(b)] into the buffer, fills in the *Identification* field, and computes its checksum. The packet is now ready for transmission.

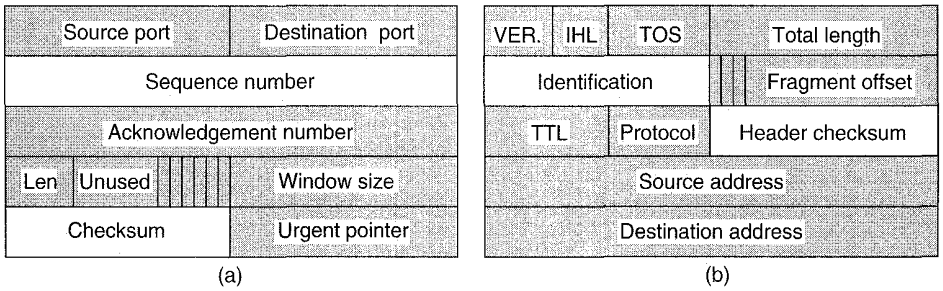


Fig. 6-50. (a) TCP header. (b) IP header. In both cases, the shaded fields are taken from the prototype without change.

Now let us look at fast path processing on the receiving side of Fig. 6-49. Step 1 is locating the connection record for the incoming TPDU. For ATM, finding the connection record is easy: the *VPI* field can be used as an index into the path table to find the virtual circuit table for that path and the *VCI* can be used as an index to find the connection record. For TCP, the connection record can be stored in a hash table for which some simple function of the two IP addresses and two ports is the key. Once the connection record has been located, both addresses and both ports must be compared to verify that the correct record has been found.

An optimization that often speeds up connection record lookup even more is just to maintain a pointer to the last one used and try that one first. Clark et al. (1989) tried this and observed a hit rate exceeding 90 percent. Other lookup heuristics are described in (McKenney and Dove, 1992).

The TPDU is then checked to see if it is a normal one: the state is *ESTABLISHED*, neither side is trying to close the connection, the TPDU is a full one, no special flags are set, and the sequence number is the one expected. These tests take just a handful of instructions. If all conditions are met, a special fast path TCP procedure is called.

The fast path updates the connection record and copies the data to the user. While it is copying, it also computes the checksum, eliminating an extra pass over the data. If the checksum is correct, the connection record is updated and an acknowledgement is sent back. The general scheme of first making a quick check to see if the header is what is expected, and having a special procedure to handle that case, is called **header prediction**. Many TCP implementations use it. When this optimization and all the other ones discussed in this chapter are used together, it is possible to get TCP to run at 90 percent of the speed of a local memory-to-memory copy, assuming the network itself is fast enough.

Two other areas where major performance gains are possible are buffer management and timer management. The issue in buffer management is avoiding unnecessary copying, as we mentioned above. Timer management is important because nearly all timers set do not expire. They are set to guard against TPDU loss, but most TPDUs arrive correctly and their acknowledgements also arrive correctly. Hence it is important to optimize timer management for the case of timers rarely expiring.

A common scheme is to use a linked list of timer events sorted by expiry time. The head entry contains a counter telling how many ticks away from expiry it is. Each successive entry contains a counter telling how many ticks after the previous entry it is. Thus if timers expire in 3, 10, and 12 ticks, respectively, the three counters are 3, 7, and 2, respectively.

At every clock tick, the counter in the head entry is decremented. When it hits zero, its event is processed and the next item on the list becomes the head. Its counter does not have to be changed. In this scheme, inserting and deleting timers are expensive operations, with execution times proportional to the length of the list.

A more efficient approach can be used if the maximum timer interval is bounded and known in advance. Here an array, called a **timing wheel**, can be used, as shown in Fig. 6-51. Each slot corresponds to one clock tick. The current time shown is $T = 4$. Timers are scheduled to expire at 3, 10, and 12 ticks from now. If a new timer suddenly is set to expire in seven ticks, an entry is just made in slot 11. Similarly, if the timer set for $T + 10$ has to be canceled, the list starting in slot 14 has to be searched and the required entry removed. Note that the array of Fig. 6-51 cannot accommodate timers beyond $T + 15$.

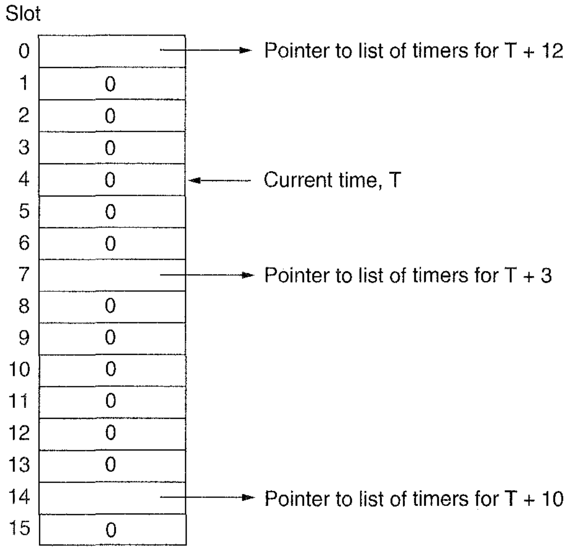

Slot
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
0
0
0
0
0
0
0
0
0
0
0
0
0
Pointer to list of timers for T + 12
Current time, T
Pointer to list of timers for T + 3
Pointer to list of timers for T + 10

Fig. 6-51. A timing wheel.

When the clock ticks, the current time pointer is advanced by one slot (circularly). If the entry now pointed to is nonzero, all of its timers are processed. Many variations on the basic idea are discussed in (Varghese and Lauck, 1987).

6.6.5. Protocols for Gigabit Networks

At the start of the 1990s, gigabit networks began to appear. People's first reaction was to use the old protocols on them, but various problems quickly arose. In this section we will discuss some of these problems and the directions new protocols are taking to solve them. Other information can be found in (Baransel et al., 1995; and Partridge, 1994).

The first problem is that many protocols use 16-bit or 32-bit sequence numbers. In the old days, 2^{32} was a pretty good approximation to infinity. It no longer is. At a data rate of 1 Gbps, it takes about 32 sec to send 2^{32} bytes. If sequence numbers refer to bytes, as they do in TCP, then a sender can start transmitting byte 0, blast away, and 32 sec later be back at byte 0. Even assuming that all bytes have been acknowledged, the sender cannot safely transmit new data labeled starting at 0 because the old packets may still be floating around somewhere. In the Internet, for example, packets can live for 120 sec. If packets are numbered instead of bytes, the problem is less severe, unless the sequence numbers are 16 bits, in which case the problem is even worse.

The problem is that many protocol designers simply assumed, without stating it, that the time to use up the entire sequence space would greatly exceed the maximum packet lifetime. Consequently there was no need to even worry about the problem of old duplicates still existing when the sequence numbers wrapped around. At gigabit speeds, that unstated assumption fails.

A second problem is that communication speeds have improved much faster than computing speeds. (Note to computer engineers: Go out and beat those communication engineers! We are counting on you.) In the 1970s, the ARPANET ran at 56 kbps and had computers that ran at about 1 MIPS. Packets were 1008 bits, so the ARPANET was capable of delivering about 56 packets/sec. With almost 18 msec available per packet, a host could afford to spend 18,000 instructions processing a packet. Of course, doing so would soak up the entire CPU, but it could devote 9000 instructions per packet and still have half the CPU left over to do real work.

Compare these numbers to modern 100-MIPS computers exchanging 4-KB packets over a gigabit line. Packets can flow in at a rate of over 30,000 per second, so packet processing must be completed in 15 μsec if we want to reserve half the CPU for applications. In 15 μsec, a 100-MIPS computer can execute 1500 instructions, only 1/6 of what the ARPANET hosts had available. Furthermore, modern RISC instructions do less per instruction than the old CISC instructions did, so the problem is even worse than it appears. The conclusion is: there is less time available for protocol processing than there used to be, so protocols must become simpler.

A third problem is that the go back n protocol performs poorly on lines with a large bandwidth-delay product. Consider, for example, a 4000-km line operating at 1 Gbps. The round-trip transmission time is 40 msec, in which time a sender can transmit 5 megabytes. If an error is detected, it will be 40 msec before the sender is told about it. If go back n is used, the sender will have to retransmit not just the bad packet, but also the 5 megabytes worth of packets that came afterward. Clearly, this is a massive waste of resources.

A fourth problem is that gigabit lines are fundamentally different from megabit lines in that long ones are delay limited rather than bandwidth limited. In Fig. 6-52 we show the time it takes to transfer a 1-megabit file 4000 km at various transmission speeds. At speeds up to 1 Mbps, the transmission time is dominated by the rate at which the bits can be sent. By 1 Gbps, the 40-msec round-trip delay dominates the 1 msec it takes to put the bits on the fiber. Further increases in bandwidth have hardly any effect at all.

Figure 6-52 has unfortunate implications for network protocols. It says that stop-and-wait protocols, such as RPC, have an inherent upper bound on their

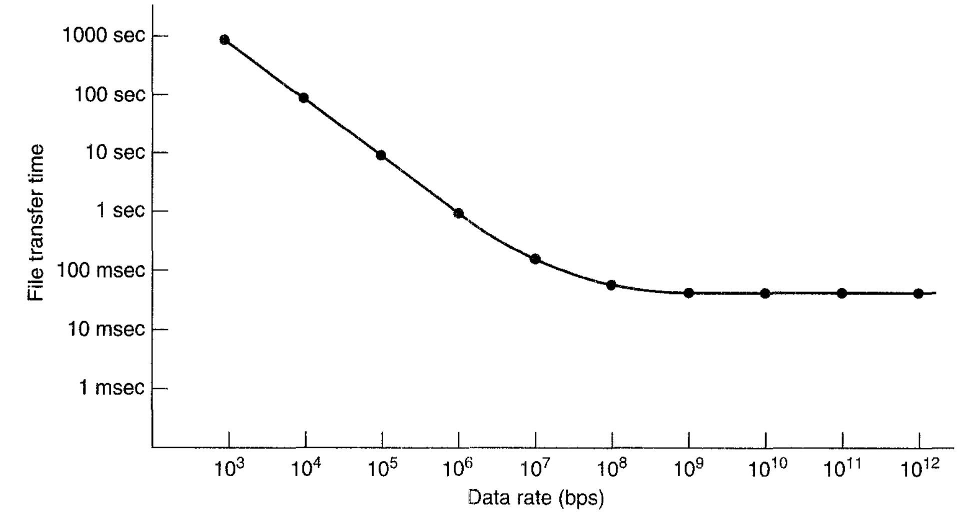


Fig. 6-52. Time to transfer and acknowledge a 1-megabit file over a 4000-km line.

performance. This limit is dictated by the speed of light. No amount of technological progress in optics will improve matters (new laws of physics would help, though).

A fifth problem that is worth mentioning is not a technological or protocol one like the others, but a result of new applications. Simply stated, it is that for many gigabit applications, such as multimedia, the variance in the packet arrival times is as important as the mean delay itself. A slow-but-uniform delivery rate, is often preferable to a fast-but-jumpy one.

Let us now turn from the problems to ways of dealing with them. We will first make some general remarks, then look at protocol mechanisms, packet layout, and protocol software.

The basic principle that all gigabit network designers should learn by heart is:

Design for speed, not for bandwidth optimization.

Old protocols were often designed to minimize the number of bits on the wire, frequently by using small fields and packing them together into bytes and words. Nowadays, there is plenty of bandwidth. Protocol processing is the problem, so protocols should be designed to minimize it.

A tempting way to go fast is to build fast network interfaces in hardware. The difficulty with this strategy is that unless the protocol is exceedingly simple, hardware just means a plug-in board with a second CPU and its own program. To avoid having the network coprocessor be as expensive as the main CPU, it is often a slower chip. The consequence of this design is that much of the time the main (fast) CPU is idle waiting for the second (slow) CPU to do the critical work. It is a myth to think that the main CPU has other work to do while waiting. Furthermore, when two general-purpose CPUs communicate, race conditions can occur, so elaborate protocols are needed between the two processors to synchronize them correctly. Usually, the best approach is to make the protocols simple and have the main CPU do the work.

Let us now look at the issue of feedback in high-speed protocols. Due to the (relatively) long delay loop, feedback should be avoided: it takes too long for the receiver to signal the sender. One example of feedback is governing the transmission rate using a sliding window protocol. To avoid the (long) delays inherent in the receiver sending window updates to the sender, it is better to use a rate-based protocol. In such a protocol, the sender can send all it wants to, provided it does not send faster than some rate the sender and receiver have agreed upon in advance.

A second example of feedback is Jacobson's slow start algorithm. This algorithm makes multiple probes to see how much the network can handle. With high-speed networks, making half a dozen or so small probes to see how the network responds wastes a huge amount of bandwidth. A more efficient scheme is to have the sender, receiver, and network all reserve the necessary resources at connection setup time. Reserving resources in advance also has the advantage of making it easier to reduce jitter. In short, going to high speeds inexorably pushes the design toward connection-oriented operation, or something fairly close to it.

Packet layout is an important consideration in gigabit networks. The header should contain as few fields as possible, to reduce processing time, and these fields should be big enough to do the job and be word aligned for ease of processing. In this context, "big enough" means that problems such as sequence numbers wrapping around while old packets still exist, receivers being unable to advertise enough window space because the window field is too small, and so on, do not occur.

The header and data should be separately checksummed, for two reasons. First, to make it possible to checksum the header but not the data. Second, to verify that the header is correct before starting to copy the data into user space. It is desirable to do the data checksum at the time the data are copied to user space, but if the header is incorrect, the copy may be to the wrong process. To avoid an incorrect copy but to allow the data checksum to be done during copying, it is essential that the two checksums be separate.

The maximum data size should be large, to permit efficient operation even in the face of long delays. Also, the larger the data block, the smaller the fraction of the total bandwidth devoted to headers.

Another valuable feature is the ability to send a normal amount of data along with the connection request. In this way, one round-trip time can be saved.

Finally, a few words about the protocol software are appropriate. A key thought is concentrating on the successful case. Many older protocols tend to emphasize what to do when something goes wrong (e.g., a packet getting lost). To make the protocols run fast, the designer should aim for minimizing processing time when everything goes right. Minimizing processing time when an error occurs is secondary.

A second software issue is minimizing copying time. As we saw earlier, copying data is often the main source of overhead. Ideally, the hardware should dump each incoming packet into memory as a contiguous block of data. The software should then copy this packet to the user buffer with a single block copy. Depending on how the cache works, it may even be desirable to avoid a copy loop. In other words, to copy 1024 words, the fastest way may be to have 1024 back-to-back MOVE instructions (or 1024 load-store pairs). The copy routine is so critical it should be carefully handcrafted in assembly code, unless there is a way to trick the compiler into producing precisely the optimal code.

In the late 1980s, there was a brief flurry of interest in fast special-purpose protocols such as NETBLT (Clark et al., 1987), VTMP (Cheriton and Williamson, 1989), and XTP (Chesson, 1989). A survey is given in (Doeringer et al., 1990). However, the trend now is toward simplifying general-purpose protocols to make them fast, too. ATM exhibits many of the features discussed above, and IPv6 does too.

6.7. SUMMARY

The transport layer is the key to understanding layered protocols. It provides various services, the most important of which is an end-to-end, reliable, connection-oriented byte stream from sender to receiver. It is accessed through service primitives that permit the establishment, use and release of connections.

Transport protocols must be able to do connection management over unreliable networks. Connection establishment is complicated by the existence of delayed duplicate packets that can reappear at inopportune moments. To deal with them, three-way handshakes are needed to establish connections. Releasing a connection is easier than establishing one but is still far from trivial due to the two-army problem.

Even when the network layer is completely reliable, the transport layer has plenty of work to do, as we saw in our example. It must handle all the service primitives, manage connections and timers, and allocate and utilize credits.

The main Internet transport protocol is TCP. It uses a 20-byte header on all segments. Segments can be fragmented by routers within the Internet, so hosts must be prepared to do reassembly. A great deal of work has gone into optimizing TCP performance, using algorithms from Nagle, Clark, Jacobson, Karn, and others.

ATM has four protocols in the AAL layer. All of them break messages into cells at the source and reassemble the cells into messages at the destination. The

CS and SAR sublayers add their own headers and trailers in various ways, leaving from 44 to 48 bytes of cell payload.

Network performance is typically dominated by protocol and TPDU processing overhead, and this situation gets worse at higher speeds. Protocols should be designed to minimize the number of TPDUs, context switches, and times each TPDU is copied. For gigabit networks, simple protocols using rate, rather than credit, flow control are called for.

PROBLEMS

1. In our example transport primitives of Fig. 6-3, LISTEN is a blocking call. Is this strictly necessary? If not, explain how a nonblocking primitive could be used. What advantage would this have over the scheme described in the text?

2. In the model underlying Fig. 6-5, it is assumed that packets may be lost by the network layer and thus must be individually acknowledged. Suppose that the network layer is 100 percent reliable and never loses packets. What changes, if any, are needed to Fig. 6-5?

3. Imagine a generalized n-army problem, in which the agreement of any two of the armies is sufficient for victory. Does a protocol exist that allows blue to win?

4. Suppose that the clock-driven scheme for generating initial sequence numbers is used with a 15-bit wide clock counter. The clock ticks once every 100 msec, and the maximum packet lifetime is 60 sec. How often need resynchronization take place
(a) in the worst case?
(b) when the data consumes 240 sequence numbers/min?

5. Why does the maximum packet lifetime, T, have to be large enough to ensure that not only the packet, but also its acknowledgements, have vanished?

6. Imagine that a two-way handshake rather than a three-way handshake were used to set up connections. In other words, the third message was not required. Are deadlocks now possible? Give an example or show that none exist.

7. Consider the problem of recovering from host crashes (i.e., Fig. 6-18). If the interval between writing and sending an acknowledgement, or vice versa, can be made relatively small, what are the two best sender-receiver strategies for minimizing the chance of a protocol failure?

8. Are deadlocks possible with the transport entity described in the text?

9. Out of curiosity, the implementer of the transport entity of Fig. 6-20 has decided to put counters inside the *sleep* procedure to collect statistics about the *conn* array. Among these are the number of connections in each of the seven possible states, n_i ($i = 1, \ldots, 7$). After writing a massive FORTRAN program to analyze the data, our implementer discovered that the relation $\sum n_i = MAX\_CONN$ appears to always be true. Are there any other invariants involving only these seven variables?

10. What happens when the user of the transport entity given in Fig. 6-20 sends a zero length message? Discuss the significance of your answer.

11. For each event that can potentially occur in the transport entity of Fig. 6-20, tell whether it is legal or not when the user is sleeping in *sending* state.

12. Discuss the advantages and disadvantages of credits versus sliding window protocols.

13. Datagram fragmentation and reassembly are handled by IP and are invisible to TCP. Does this mean that TCP does not have to worry about data arriving in the wrong order?

14. A process on host 1 has been assigned port p and a process on host 2 has been assigned port q. Is it possible for there to be two or more TCP connections between these two ports at the same time?

15. The maximum payload of a TCP segment is 65,515 bytes. Why was such a strange number chosen?

16. Describe two ways to get into the *SYN RCVD* state of Fig. 6-28.

17. Give a potential disadvantage when Nagle's algorithm is used on a badly congested network.

18. Consider the effect of using slow start on a line with a 10-msec round-trip time and no congestion. The receive window is 24 KB and the maximum segment size is 2 KB. How long does it take before the first full window can be sent?

19. Suppose that the TCP congestion window is set to 18K bytes and a timeout occurs. How big will the window be if the next four transmission bursts are all successful? Assume that the maximum segment size is 1 KB.

20. If the TCP round-trip time, *RTT*, is currently 30 msec and the following acknowledgements come in after 26, 32, and 24 msec, respectively, what is the new *RTT* estimate? Use $\alpha = 0.9$.

21. A TCP machine is sending windows of 65,535 bytes over a 1-Gbps channel that has a 10-msec one-way delay. What is the maximum throughput achievable? What is the line efficiency?

22. In a network that has a maximum TPDU size of 128 bytes, a maximum TPDU lifetime of 30 sec, and an 8-bit sequence number, what is the maximum data rate per connection?

23. Why does UDP exist? Would it not have been enough to just let user processes send raw IP packets?

24. A group of N users located in the same building are all using the same remote computer via an ATM network. The average user generates L lines of traffic (input + output) per hour, on the average, with the mean line length being P bytes, excluding the ATM headers. The packet carrier charges C cents per byte of user data transported, plus X cents per hour for each ATM virtual circuit open. Under what conditions is it cost effective to multiplex all N transport connections onto the same ATM virtual circuit, if such multiplexing adds 2 bytes of data to each packet? Assume that even one ATM virtual circuit has enough bandwidth for all the users.

25. Can AAL 1 handle messages shorter than 40 bytes using the scheme with the *Pointer* field? Explain your answer.

26. Make a guess at what the field sizes for AAL 2 were before they were pulled from the standard.

27. AAL 3/4 allows multiple sessions to be multiplexed onto a single virtual circuit. Give an example of a situation in which that has no value. Assume that one virtual circuit has sufficient bandwidth to carry all the traffic. *Hint*: Think about virtual paths.

28. What is the payload size of the maximum length message that fits in a single AAL 3/4 cell?

29. When a 1024-byte message is sent with AAL 3/4, what is the efficiency obtained? In other words, what fraction of the bits transmitted are useful data bits? Repeat the problem for AAL 5.

30. An ATM device is transmitting single-cell messages at 600 Mbps. One cell in 100 is totally scrambled due to random noise. How many undetected errors per week can be expected with the 32-bit AAL 5 checksum?

31. A client sends a 128-byte request to a server located 100 km away over a 1-gigabit optical fiber. What is the efficiency of the line during the remote procedure call?

32. Consider the situation of the previous problem again. Compute the minimum possible response time both for the given 1-Gbps line and for a 1-Mbps line. What conclusion can you draw?

33. Suppose that you are measuring the time to receive a TPDU. When an interrupt occurs, you read out the system clock in milliseconds. When the TPDU is fully processed, you read out the clock again. You measure 0 msec 270,000 times and 1 msec 730,000 times. How long does it take to receive a TPDU?

34. A CPU executes instructions at the rate of 100 MIPS. Data can be copied 64 bits at a time, with each word copied costing six instructions. If an coming packet has to be copied twice, can this system handle a 1-Gbps line? For simplicity, assume that all instructions, even those instructions that read or write memory, run at the full 100-MIPS rate.

35. To get around the problem of sequence numbers wrapping around while old packets still exist, one could use 64-bit sequence numbers. However, theoretically, an optical fiber can run at 75 Tbps. What maximum packet lifetime is required to make sure that future 75 Tbps networks do not have wraparound problems even with 64-bit sequence numbers? Assume that each byte has its own sequence number, as TCP does.

36. In the text we calculated that a gigabit line dumps 30,000 packets/sec on the host, giving it only 1500 instructions to process it and leaving half the CPU time for applications. This calculation assumed a 4-KB packet. Redo the calculation for an ARPANET-sized packet (128 bytes).

37. For a 1-Gbps network operating over 4000 km, the delay is the limiting factor, not the bandwidth. Consider a MAN with the average source and destination 20 km apart. At what data rate does the round-trip delay due to the speed of light equal the transmission delay for a 1-KB packet?

38. Modify the program of Fig. 6-20 to do error recovery. Add a new packet type, *reset*, that can arrive after a connection has been opened by both sides but closed by neither. This event, which happens simultaneously on both ends of the connection, means that any packets that were in transit have either been delivered or destroyed, but in either case are no longer in the subnet.

39. Write a program that simulates buffer management in a transport entity using a sliding window for flow control rather than the credit system of Fig. 6-20. Let higher-layer processes randomly open connections, send data, and close connections. To keep it simple, have all the data travel from machine *A* to machine *B*, and none the other way. Experiment with different buffer allocation strategies at *B*, such as dedicating buffers to specific connections versus a common buffer pool, and measure the total throughput achieved by each one.

7

THE APPLICATION LAYER

Having finished all the preliminaries, we now come to the application layer, where all the interesting applications can be found. The layers below the application layer are there to provide reliable transport, but they do not do any real work for users. In this chapter we will study some real applications.

However, even in the application layer there is a need for support protocols to allow the real applications to function. Accordingly, we will look at three of these before starting with the applications themselves. The first area is security, which is not a single protocol, but a large number of concepts and protocols that can be used to ensure privacy where needed. The second is DNS, which handles naming within the Internet. The third support protocol is for network management. After that, we will examine four real applications: electronic mail, USENET (net news), the World Wide Web, and finally, multimedia.

7.1. NETWORK SECURITY

For the first few decades of their existence, computer networks were primarily used by university researchers for sending email, and by corporate employees for sharing printers. Under these conditions, security did not get a lot of attention. But now, as millions of ordinary citizens are using networks for banking, shopping, and filing their tax returns, network security is looming on the horizon as a potentially massive problem. In the following sections, we will study network security from several angles, point out numerous pitfalls, and discuss many algorithms and protocols for making networks more secure.

Security is a broad topic and covers a multitude of sins. In its simplest form, it is concerned with making sure that nosy people cannot read, or worse yet, modify messages intended for other recipients. It is concerned with people trying to access remote services that they are not authorized to use. It also deals with how to tell whether that message purportedly from the IRS saying: "Pay by Friday or else" is really from the IRS or from the Mafia. Security also deals with the problems of legitimate messages being captured and replayed, and with people trying to deny that they sent certain messages.

Most security problems are intentionally caused by malicious people trying to gain some benefit or harm someone. A few of the most common perpetrators are listed in Fig. 7-1. It should be clear from this list that making a network secure involves a lot more than just keeping it free of programming errors. It involves outsmarting often intelligent, dedicated, and sometimes well-funded adversaries. It should also be clear that measures that will stop casual adversaries will have little impact on the serious ones.

Adversary	Goal
Student	To have fun snooping on people's email
Hacker	To test out someone's security system; steal data
Sales rep	To claim to represent all of Europe, not just Andorra
Businessman	To discover a competitor's strategic marketing plan
Ex-employee	To get revenge for being fired
Accountant	To embezzle money from a company
Stockbroker	To deny a promise made to a customer by email
Con man	To steal credit card numbers for sale
Spy	To learn an enemy's military strength
Terrorist	To steal germ warfare secrets

Fig. 7-1. Some people who cause security problems and why.

Network security problems can be divided roughly into four intertwined areas: secrecy, authentication, nonrepudiation, and integrity control. Secrecy has to do with keeping information out of the hands of unauthorized users. This is what usually comes to mind when people think about network security. Authentication deals with determining whom you are talking to before revealing sensitive information or entering into a business deal. Nonrepudiation deals with signatures:

How do you prove that your customer really placed an electronic order for ten million left-handed doohickeys at 89 cents each when he later claims the price was 69 cents? Finally, how can you be sure that a message you received was really the one sent and not something that a malicious adversary modified in transit or concocted?

All these issues (secrecy, authentication, nonrepudiation, and integrity control) occur in traditional systems, too, but with some significant differences. Secrecy and integrity are achieved by using registered mail and locking documents up. Robbing the mail train is harder than it was in Jesse James' day.

Also, people can usually tell the difference between an original paper document and a photocopy, and it often matters to them. As a test, make a photocopy of a valid check. Try cashing the original check at your bank on Monday. Now try cashing the photocopy of the check on Tuesday. Observe the difference in the bank's behavior. With electronic checks, the original and the copy are indistinguishable. It may take a while for banks to get used to this.

People authenticate other people by recognizing their faces, voices, and handwriting. Proof of signing is handled by signatures on letterhead paper, raised seals, and so on. Tampering can usually be detected by handwriting, paper, and ink experts. None of these options are available electronically. Clearly, other solutions are needed.

Before getting into the solutions themselves, it is worth spending a few moments considering where in the protocol stack network security belongs. There is probably no one single place. Every layer has something to contribute. In the physical layer, wiretapping can be foiled by enclosing transmission lines in sealed tubes containing argon gas at high pressure. Any attempt to drill into a tube will release some gas, reducing the pressure and triggering an alarm. Some military systems use this technique.

In the data link layer, packets on a point-to-point line can be encoded as they leave one machine and decoded as they enter another. All the details can be handled in the data link layer, with higher layers oblivious to what is going on. This solution breaks down when packets have to traverse multiple routers, however, because packets have to be decrypted at each router, leaving them vulnerable to attacks from within the router. Also, it does not allow some sessions to be protected (e.g., those involving on-line purchases by credit card) and others not. Nevertheless, **link encryption**, as this method is called, can be added to any network easily and is often useful.

In the network layer, firewalls can be installed to keep packets in or keep packets out. We looked at firewalls in Chap. 5. In the transport layer, entire connections can be encrypted, end to end, that is, process to process. Although these solutions help with secrecy issues and many people are working hard to improve them, none of them solve the authentication or nonrepudiation problem in a sufficiently general way. To tackle these problems, the solutions must be in the application layer, which is why they are being studied in this chapter.

7.1.1. Traditional Cryptography

Cryptography has a long and colorful history. In this section we will just sketch some of the highlights, as background information for what follows. For a complete history, Kahn's (1967) book is still recommended reading. For a comprehensive treatment of the current state-of-the-art, see (Kaufman et al., 1995; Schneier, 1996; and Stinson, 1995).

Historically, four groups of people have used and contributed to the art of cryptography: the military, the diplomatic corps, diarists, and lovers. Of these, the military has had the most important role and has shaped the field. Within military organizations, the messages to be encrypted have traditionally been given to poorly paid code clerks for encryption and transmission. The sheer volume of messages prevented this work from being done by a few elite specialists.

Until the advent of computers, one of the main constraints on cryptography had been the ability of the code clerk to perform the necessary transformations, often on a battlefield with little equipment. An additional constraint has been the difficulty in switching over quickly from one cryptographic method to another one, since this entails retraining a large number of people. However, the danger of a code clerk being captured by the enemy has made it essential to be able to change the cryptographic method instantly, if need be. These conflicting requirements have given rise to the model of Fig. 7-2.

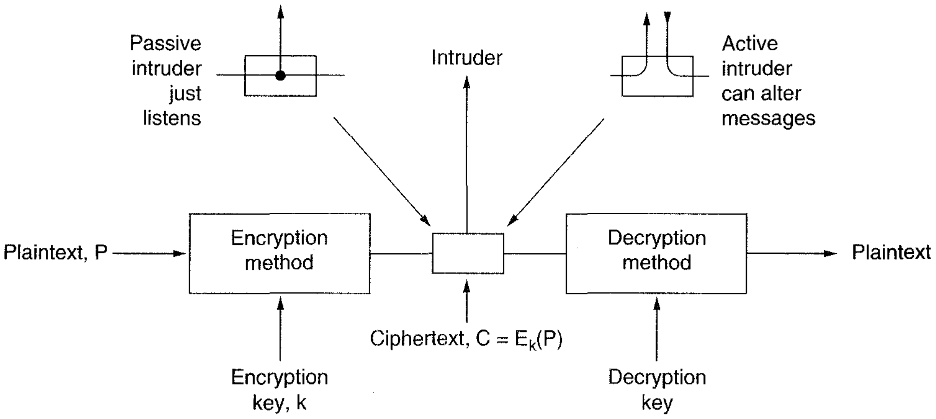


Fig. 7-2. The encryption model.

The messages to be encrypted, known as the **plaintext**, are transformed by a function that is parametrized by a **key**. The output of the encryption process, known as the **ciphertext**, is then transmitted, often by messenger or radio. We assume that the enemy, or **intruder**, hears and accurately copies down the complete ciphertext. However, unlike the intended recipient, he does not know what the decryption key is and so cannot decrypt the ciphertext easily. Sometimes the intruder can not only listen to the communication channel (passive intruder) but can also record messages and play them back later, inject his own messages, or modify legitimate messages before they get to the receiver (active intruder). The art of breaking ciphers is called **cryptanalysis**. The art of devising ciphers (cryptography) and breaking them (cryptanalysis) is collectively known as **cryptology**.

It will often be useful to have a notation for relating plaintext, ciphertext, and keys. We will use $C = E_K(P)$ to mean that the encryption of the plaintext P using key K gives the ciphertext C. Similarly, $P = D_K(C)$ represents of decryption of C to get the plaintext again. It then follows that

$$D_K(E_K(P)) = P$$

This notation suggests that E and D are just mathematical functions, which they are. The only tricky part is that both are functions of two parameters, and we have written one of the parameters (the key) as a subscript, rather than as an argument, to distinguish it from the message.

A fundamental rule of cryptography is that one must assume that the cryptanalyst knows the general method of encryption used. In other words, the cryptanalyst knows how the encryption method, E, of Fig. 7-2 works. The amount of effort necessary to invent, test, and install a new method every time the old method is compromised or thought to be compromised has always made it impractical to keep this secret, and thinking it is secret when it is not does more harm than good.

This is where the key enters. The key consists of a (relatively) short string that selects one of many potential encryptions. In contrast to the general method, which may only be changed every few years, the key can be changed as often as required. Thus our basic model is a stable and publicly known general method parametrized by a secret and easily changed key.

The nonsecrecy of the algorithm cannot be emphasized enough. By publicizing the algorithm, the cryptographer gets free consulting from a large number of academic cryptologists eager to break the system so they can publish papers demonstrating how smart they are. If many experts have tried to break the algorithm for 5 years after its publication and no one has succeeded, it is probably pretty solid.

The real secrecy is in the key, and its length is a major design issue. Consider a simple combination lock. The general principle is that you enter digits in sequence. Everyone knows this, but the key is secret. A key length of two digits means that there are 100 possibilities. A key length of three digits means 1000 possibilities, and a key length of six digits means a million. The longer the key, the higher the **work factor** the cryptanalyst has to deal with. The work factor for breaking the system by exhaustive search of the key space is exponential in the key length. Secrecy comes from having a strong (but public) algorithm and a long key. To prevent your kid brother from reading your email, 64-bit keys will do. To keep major governments at bay, keys of at least 256 bits are needed.

From the cryptanalyst's point of view, the cryptanalysis problem has three principal variations. When he has a quantity of ciphertext and no plaintext, he is confronted with the **ciphertext only** problem. The cryptograms that appear in the puzzle section of newspapers pose this kind of problem. When he has some matched ciphertext and plaintext, the problem becomes known as the **known plaintext** problem. Finally, when the cryptanalyst has the ability to encrypt pieces of plaintext of his own choosing, we have the **chosen plaintext** problem. Newspaper cryptograms could be broken trivially if the cryptanalyst were allowed to ask such questions as: What is the encryption of ABCDE?

Novices in the cryptography business often assume that if a cipher can withstand a ciphertext only attack, it is secure. This assumption is very naive. In many cases the cryptanalyst can make a good guess at parts of the plaintext. For example, the first thing many timesharing systems say when you call them up is "PLEASE LOGIN." Equipped with some matched plaintext-ciphertext pairs, the cryptanalyst's job becomes much easier. To achieve security, the cryptographer should be conservative and make sure that the system is unbreakable even if his opponent can encrypt arbitrary amounts of chosen plaintext.

Encryption methods have historically been divided into two categories: substitution ciphers and transposition ciphers. We will now deal with each of these briefly as background information for modern cryptography.

Substitution Ciphers

In a **substitution cipher** each letter or group of letters is replaced by another letter or group of letters to disguise it. One of the oldest known ciphers is the **Caesar cipher**, attributed to Julius Caesar. In this method, *a* becomes *D*, *b* becomes *E*, *c* becomes *F*, . . . , and *z* becomes *C*. For example, *attack* becomes *DWWDFN*. In examples, plaintext will be given in lowercase letters, and ciphertext in uppercase letters.

A slight generalization of the Caesar cipher allows the ciphertext alphabet to be shifted by *k* letters, instead of always 3. In this case *k* becomes a key to the general method of circularly shifted alphabets. The Caesar cipher may have fooled the Carthaginians, but it has not fooled anyone since.

The next improvement is to have each of the symbols in the plaintext, say the 26 letters for simplicity, map onto some other letter. For example,

plaintext: a b c d e f g h i j k l m n o p q r s t u v w x y z

ciphertext: Q W E R T Y U I O P A S D F G H J K L Z X C V B N M

This general system is called a **monoalphabetic substitution**, with the key being the 26-letter string corresponding to the full alphabet. For the key above, the plaintext *attack* would be transformed into the ciphertext *QZZQEA*.

At first glance this might appear to be a safe system because although the cryptanalyst knows the general system (letter for letter substitution), he does not know which of the $26! \simeq 4 \times 10^{26}$ possible keys is in use. In contrast with the Caesar cipher, trying all of them is not a promising approach. Even at 1 μsec per solution, a computer would take 10^{13} years to try all the keys.

Nevertheless, given a surprisingly small amount of ciphertext, the cipher can be broken easily. The basic attack takes advantage of the statistical properties of natural languages. In English, for example, *e* is the most common letter, followed by *t*, *o*, *a*, *n*, *i*, etc. The most common two letter combinations, or **digrams**, are *th*, *in*, *er*, *re*, and *an*. The most common three letter combinations, or **trigrams**, are *the*, *ing*, *and*, and *ion*.

A cryptanalyst trying to break a monoalphabetic cipher would start out by counting the relative frequencies of all letters in the ciphertext. Then he might tentatively assign the most common one to *e* and the next most common one to *t*. He would then look at trigrams to find a common one of the form *tXe*, which strongly suggests that *X* is *h*. Similarly, if the pattern *thYt* occurs frequently, the *Y* probably stands for *a*. With this information, he can look for a frequently occurring trigram of the form *aZW*, which is most likely *and*. By making guesses at common letters, digrams, and trigrams, and knowing about likely patterns of vowels and consonants, the cryptanalyst builds up a tentative plaintext, letter by letter.

Another approach is to guess a probable word or phrase. For example, consider the following ciphertext from an accounting firm (blocked into groups of five characters):

CTBMN BYCTC BTJDS QXBNS GSTJC BTSWX CTQTZ CQVUJ
QJSGS TJQZZ MNQJS VLNSX VSZJU JDSTS JQUUS JUBXJ
DSKSU JSNTK BGAQJ ZBGYQ TLCTZ BNYBN QJSW

A likely word in a message from an accounting firm is *financial*. Using our knowledge that *financial* has a repeated letter (*i*), with four other letters between their occurrences, we look for repeated letters in the ciphertext at this spacing. We find 12 hits, at positions 6, 15, 27, 31, 42, 48, 56, 66, 70, 71, 76, and 82. However, only two of these, 31 and 42, have the next letter (corresponding to *n* in the plaintext) repeated in the proper place. Of these two, only 31 also has the *a* correctly positioned, so we know that *financial* begins at position 30. From this point on, deducing the key is easy by using the frequency statistics for English text.

Transposition Ciphers

Substitution ciphers preserve the order of the plaintext symbols but disguise them. **Transposition ciphers**, in contrast, reorder the letters but do not disguise them. Figure 7-3 depicts a common transposition cipher, the columnar transposition. The cipher is keyed by a word or phrase not containing any repeated letters. In this example, MEGABUCK is the key. The purpose of the key is to number the columns, column 1 being under the key letter closest to the start of the alphabet, and so on. The plaintext is written horizontally, in rows. The ciphertext is read out by columns, starting with the column whose key letter is the lowest.

M	E	G	A	B	U	C	K
7	4	5	1	2	8	3	6
p	l	e	a	s	e	t	r
a	n	s	f	e	r	o	n
e	m	i	l	l	i	o	n
d	o	l	l	a	r	s	t
o	m	y	s	w	i	s	s
b	a	n	k	a	c	c	o
u	n	t	s	i	x	t	w
o	t	w	o	a	b	c	d

Plaintext

pleasetransferonemilliondollarsto
myswissbankaccountsixtwotwo

Ciphertext

AFLLSKSOSELAWAIATOOSSCTCLNMOMANT
ESILYNTWRNNTSOWDPAEDOBUOERIRICXB

Fig. 7-3. A transposition cipher.

To break a transposition cipher, the cryptanalyst must first be aware that he is dealing with a transposition cipher. By looking at the frequency of *E*, *T*, *A*, *O*, *I*, *N*, etc., it is easy to see if they fit the normal pattern for plaintext. If so, the cipher is clearly a transposition cipher, because in such a cipher every letter represents itself.

The next step is to make a guess at the number of columns. In many cases a probable word or phrase may be guessed at from the context of the message. For example, suppose that our cryptanalyst suspected the plaintext phrase *milliondollars* to occur somewhere in the message. Observe that digrams *MO*, *IL*, *LL*, *LA*, *IR* and *OS* occur in the ciphertext as a result of this phrase wrapping around. The ciphertext letter *O* follows the ciphertext letter *M* (i.e., they are vertically adjacent in column 4) because they are separated in the probable phrase by a distance equal to the key length. If a key of length seven had been used, the digrams *MD*, *IO*, *LL*, *LL*, *IA*, *OR*, and *NS* would have occurred instead. In fact, for each key length, a different set of digrams is produced in the ciphertext. By hunting for the various possibilities, the cryptanalyst can often easily determine the key length.

The remaining step is to order the columns. When the number of columns, k, is small, each of the $k(k-1)$ column pairs can be examined to see if its digram frequencies match those for English plaintext. The pair with the best match is assumed to be correctly positioned. Now each remaining column is tentatively tried as the successor to this pair. The column whose digram and trigram frequencies give the best match is tentatively assumed to be correct. The predecessor column is found in the same way. The entire process is continued until a potential ordering is found. Chances are that the plaintext will be recognizable at this point (e.g., if *milloin* occurs, it is clear what the error is).

Some transposition ciphers accept a fixed-length block of input and produce a fixed-length block of output. These ciphers can be completely described by just giving a list telling the order in which the characters are to be output. For example, the cipher of Fig. 7-3 can be seen as a 64 character block cipher. Its output is 4, 12, 20, 28, 36, 44, 52, 60, 5, 13 , . . . , 62. In other words, the fourth input character, *a*, is the first to be output, followed by the twelfth, *f*, and so on.

One-Time Pads

Constructing an unbreakable cipher is actually quite easy; the technique has been known for decades. First choose a random bit string as the key. Then convert the plaintext into a bit string, for example by using its ASCII representation. Finally, compute the EXCLUSIVE OR of these two strings, bit by bit. The resulting ciphertext cannot be broken, because every possible plaintext is an equally probable candidate. The ciphertext gives the cryptanalyst no information at all. In a sufficiently large sample of ciphertext, each letter will occur equally often, as will every digram and every trigram.

This method, known as the **one-time pad**, has a number of practical disadvantages, unfortunately. To start with, the key cannot be memorized, so both sender and receiver must carry a written copy with them. If either one is subject to capture, written keys are clearly undesirable. Additionally, the total amount of data that can be transmitted is limited by the amount of key available. If the spy strikes it rich and discovers a wealth of data, he may find himself unable to transmit it back to headquarters because the key has been used up. Another problem is the sensitivity of the method to lost or inserted characters. If the sender and receiver get out of synchronization, all data from then on will appear garbled.

With the advent of computers, the one-time pad might potentially become practical for some applications. The source of the key could be a special CD that contains several gigabits of information, and if transported in a music CD box and prefixed by a few songs, would not even be suspicious. Of course, at gigabit network speeds, having to insert a new CD every 5 sec could become tedious. For this reason, we will now start looking at modern encryption algorithms that can process arbitrarily large amounts of plaintext.

7.1.2. Two Fundamental Cryptographic Principles

Although we will study many different cryptographic systems in the pages ahead, there are two principles underlying all of them that are important to understand. The first principle is that all encrypted messages must contain some redundancy, that is, information not needed to understand the message. An example may make it clear why this is needed. Consider a mail-order company, The Couch Potato (TCP), with 60,000 products. Thinking they are being very efficient, TCP's programmers decide that ordering messages should consist of a 16-byte customer name followed by a 3-byte data field (1 byte for the quantity and 2 bytes for the product number). The last 3 bytes are to be encrypted using a very long key known only by the customer and TCP.

At first this might seem secure, and in a sense it is because passive intruders cannot decrypt the messages. Unfortunately, it also has a fatal flaw that renders it useless. Suppose that a recently-fired employee wants to punish TCP for firing her. Just before leaving, she takes (part of) the customer list with her. She works through the night writing a program to generate fictitious orders using real customer names. Since she does not have the list of keys, she just puts random numbers in the last 3 bytes, and sends hundreds of orders off to TCP.

When these messages arrive, TCP's computer uses the customer's name to locate the key and decrypt the message. Unfortunately for TCP, almost every 3-byte message is valid, so the computer begins printing out shipping instructions. While it might seem odd for a customer to order 137 sets of children's swings, or 240 sandboxes, for all the computer knows, the customer might be planning to open a chain of franchised playgrounds. In this way an active intruder (the ex-employee) can cause a massive amount of trouble, even though she cannot understand the messages her computer is generating.

This problem can be solved by adding redundancy to all messages. For example, if order messages are extended to 12 bytes, the first 9 of which must be zeros, then this attack no longer works because the ex-employee no longer can generate a large stream of valid messages. The moral of the story is that all messages must contain considerable redundancy so that active intruders cannot send random junk and have it be interpreted as a valid message.

However, adding redundancy also makes it much easier for cryptanalysts to break messages. Suppose that the mail order business is highly competitive, and The Couch Potato's main competitor, The Sofa Tuber, would dearly love to know how many sandboxes TCP is selling. Consequently, they have tapped TCP's telephone line. In the original scheme with 3-byte messages, cryptanalysis was nearly impossible, because after guessing a key, the cryptanalyst had no way of telling whether the guess was right. After all, almost every message is technically legal. With the new 12-byte scheme, it is easy for the cryptanalyst to tell a valid message from an invalid one.

Thus cryptographic principle number one is that all messages must contain redundancy to prevent active intruders from tricking the receiver into acting on a false message. However, this same redundancy makes it much easier for passive intruders to break the system, so there is some tension here. Furthermore, the redundancy should never be in the form of *n* zeros at the start or end of a message, since running such messages through some cryptographic algorithms gives more predictable results, making the cryptanalysts' job easier. A random string of English words would be a much better choice for the redundancy.

The second cryptographic principle is that some measures must be taken to prevent active intruders from playing back old messages. If no such measures were taken, our ex-employee could tap TCP's phone line and just keep repeating previously sent valid messages. One such measure is including in every message a timestamp valid only for, say, 5 minutes. The receiver can then just keep messages around for 5 minutes, to compare newly arrived messages to previous ones to filter out duplicates. Messages older than 5 minutes can be thrown out, since any replays sent more than 5 minutes later will be rejected as too old. Measures other than timestamps will be discussed later.

7.1.3. Secret-Key Algorithms

Modern cryptography uses the same basic ideas as traditional cryptography, transposition and substitution, but its emphasis is different. Traditionally, cryptographers have used simple algorithms and relied on very long keys for their security. Nowadays the reverse is true: the object is to make the encryption algorithm so complex and involuted that even if the cryptanalyst acquires vast mounds of enciphered text of his own choosing, he will not be able to make any sense of it at all.

Transpositions and substitutions can be implemented with simple circuits. Figure 7-4(a) shows a device, known as a **P-box** (P stands for permutation), used to effect a transposition on an 8-bit input. If the 8 bits are designated from top to bottom as 01234567, the output of this particular P-box is 36071245. By appropriate internal wiring, a P-box can be made to perform any transposition, and do it at practically the speed of light.

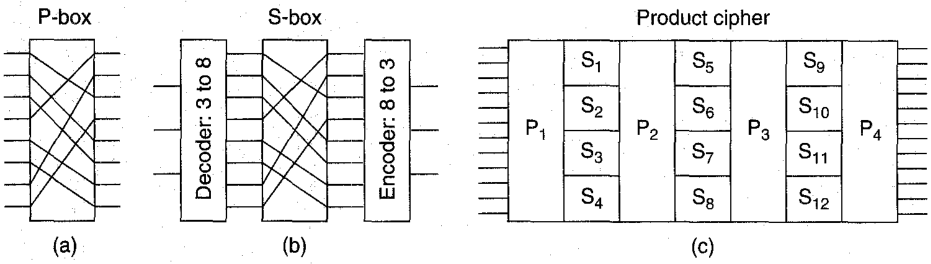


Fig. 7-4. Basic elements of product ciphers. (a) P-box. (b) S-box. (c) Product.

Substitutions are performed by **S-boxes**, as shown in Fig. 7-4(b). In this example a 3-bit plaintext is entered and a 3-bit ciphertext is output. The 3-bit input selects one of the eight lines exiting from the first stage and sets it to 1; all the other lines are 0. The second stage is a P-box. The third stage encodes the selected input line in binary again. With the wiring shown, if the eight octal numbers 01234567 were input one after another, the output sequence would be 24506713. In other words, 0 has been replaced by 2, 1 has been replaced by 4, etc. Again, by appropriate wiring of the P-box inside the S-box, any substitution can be accomplished.

The real power of these basic elements only becomes apparent when we cascade a whole series of boxes to form a **product cipher**, as shown in Fig. 7-4(c). In this example, 12 input lines are transposed by the first stage. Theoretically, it would be possible to have the second stage be an S-box that mapped a 12-bit number onto another 12-bit number. However, such a device would need $2^{12} = 4096$ crossed wires in its middle stage. Instead, the input is broken up into four groups of 3 bits, each of which is substituted independently of the others. Although this method is less general, it is still powerful. By including a sufficiently large number of stages in the product cipher, the output can be made to be an exceedingly complicated function of the input.

DES

In January 1977, the U.S. government adopted a product cipher developed by IBM as its official standard for unclassified information. This cipher, **DES (Data Encryption Standard)**, was widely adopted by the industry for use in security products. It is no longer secure in its original form (Wayner, 1995), but in a modified form it is still useful. We will now explain how DES works.

An outline of DES is shown in Fig. 7-5(a). Plaintext is encrypted in blocks of 64 bits, yielding 64 bits of ciphertext. The algorithm, which is parametrized by a 56-bit key, has 19 distinct stages. The first stage is a key independent transposition on the 64-bit plaintext. The last stage is the exact inverse of this transposition. The stage prior to the last one exchanges the leftmost 32 bits with the rightmost 32 bits. The remaining 16 stages are functionally identical but are parametrized by different functions of the key. The algorithm has been designed to allow decryption to be done with the same key as encryption. The steps are just run in the reverse order.

The operation of one of these intermediate stages is illustrated in Fig. 7-5(b). Each stage takes two 32-bit inputs and produces two 32-bit outputs. The left output is simply a copy of the right input. The right output is the bitwise EXCLUSIVE OR of the left input and a function of the right input and the key for this stage, K_i. All the complexity lies in this function.

The function consists of four steps, carried out in sequence. First, a 48-bit number, E, is constructed by expanding the 32-bit R_{i-1} according to a fixed transposition and duplication rule. Second, E and K_i are EXCLUSIVE ORed together. This output is then partitioned into eight groups of 6 bits each, each of which is fed into a different S-box. Each of the 64 possible inputs to an S-box is mapped onto a 4-bit output. Finally, these 8 × 4 bits are passed through a P-box.

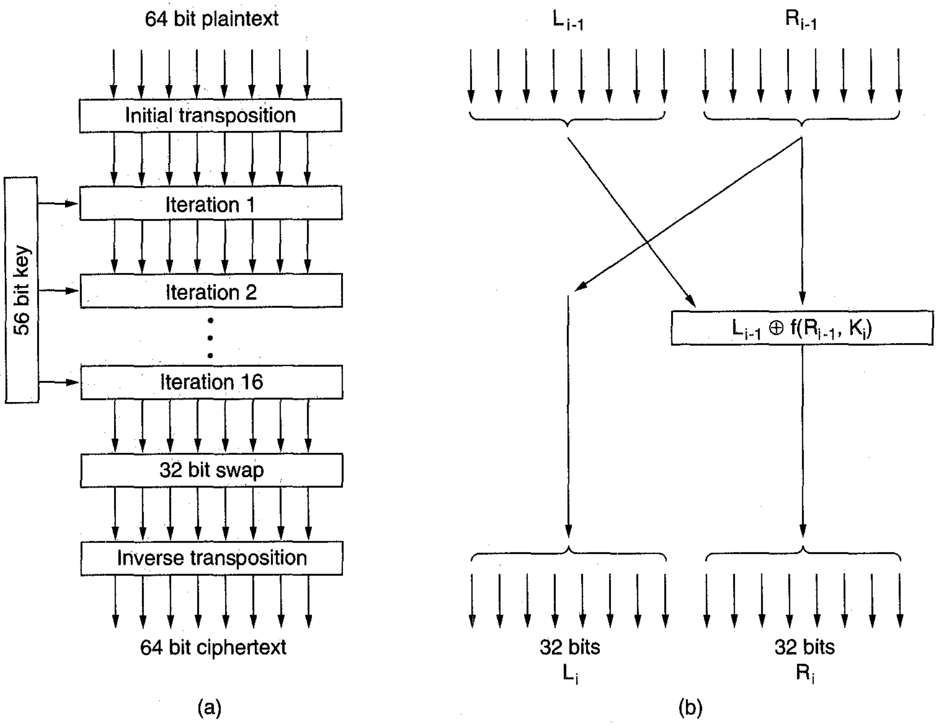


Fig. 7-5. The data encryption standard. (a) General outline. (b) Detail of one iteration.

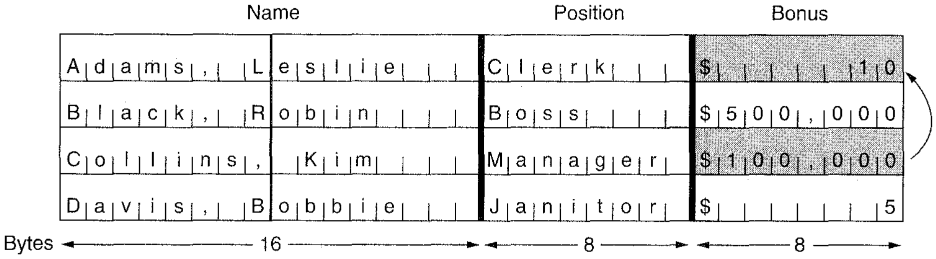
In each of the 16 iterations, a different key is used. Before the algorithm starts, a 56-bit transposition is applied to the key. Just before each iteration, the key is partitioned into two 28-bit units, each of which is rotated left by a number of bits dependent on the iteration number. K_i is derived from this rotated key by applying yet another 56-bit transposition to it. A different 48-bit subset of the 56 bits is extracted and permuted on each round.

DES Chaining

Despite all this complexity, DES is basically a monoalphabetic substitution cipher using a 64-bit character. Whenever the same 64-bit plaintext block goes in the front end, the same 64-bit ciphertext block comes out the back end. A cryptanalyst can exploit this property to help break DES.

To see how this monoalphabetic substitution cipher property can be used to subvert DES, let us consider encrypting a long message the obvious way: by breaking it up into consecutive 8-byte (64-bit) blocks and encrypting them one after another with the same key. The last block is padded out to 64 bits, if need be. This technique is known as **electronic code book mode**.

In Fig. 7-6 we have the start of a computer file listing the annual bonuses a company has decided to award to its employees. This file consists of consecutive 32-byte records, one per employee, in the format shown: 16 bytes for the name, 8 bytes for the position, and 8 bytes for the bonus. Each of the sixteen 8-byte blocks (numbered from 0 to 15) is encrypted by DES.



Name		Position	Bonus
Adams, L	eslie	Clerk	$ 10
Black, R	obin	Boss	$500,000
Collins,	Kim	Manager	$100,000
Davis, B	obbie	Janitor	$ 5

Fig. 7-6. The plaintext of a file encrypted as 16 DES blocks.

Leslie just had a fight with the boss and is not expecting much of a bonus. Kim, in contrast is the boss' favorite, and everyone knows this. Leslie can get access to the file after it is encrypted but before it is sent to the bank. Can Leslie rectify this unfair situation, given only the encrypted file?

No problem at all. All Leslie has to do is make a copy of ciphertext block 11 (which contains Kim's bonus) and use it to replace ciphertext block 3 (which contains Leslie's bonus). Even without knowing what block 11 says, Leslie can expect to have a much merrier Christmas this year. (Copying ciphertext block 7 is also a possibility, but is more likely to be detected; besides, Leslie is not a greedy person.)

To thwart this type of attack, DES (and all block ciphers) can be chained in various ways so that replacing a block the way Leslie did will cause the plaintext decrypted starting at the replaced block to be garbage. One way of chaining is **cipher block chaining**. In this method, shown in Fig. 7-7, each plaintext block is EXCLUSIVE ORed (#) with the previous ciphertext block before being encrypted. Consequently, the same plaintext block no longer maps onto the same ciphertext block, and the encryption is no longer a big monoalphabetic substitution cipher. The first block is EXCLUSIVE ORed with a randomly chosen **initialization vector, IV**, that is transmitted along with the ciphertext.

We can see how cipher block chaining works by examining the example of Fig. 7-7. We start out by computing $C_0 = E(P_0 \text{ XOR } IV)$. Then we compute $C_1 = E(P_1 \text{ XOR } C_0)$, and so on. Decryption works the other way, with $P_0 = IV \text{ XOR } D(C_0)$, and so on. Note that the encryption of block i is a function

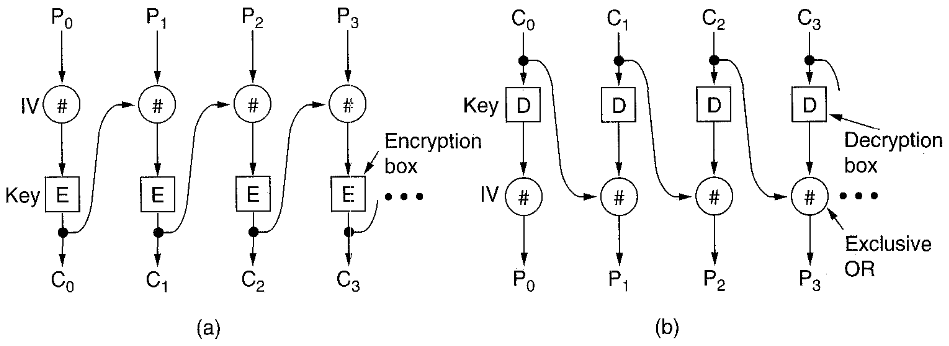


Fig. 7-7. Cipher block chaining

of all the plaintext in blocks 0 through $i - 1$, so the same plaintext generates different ciphertext depending on where it occurs. A transformation of the type Leslie made will result in nonsense for two blocks starting at Leslie's bonus field. To an astute security officer, this peculiarity might suggest where to start the ensuing investigation.

Cipher block chaining also has the advantage that the same plaintext block will not result in the same ciphertext block, making cryptanalysis more difficult. In fact, this is the main reason it is used.

However, cipher block chaining has the disadvantage of requiring an entire 64-bit block to arrive before decryption can begin. For use with interactive terminals, where people can type lines shorter than eight characters and then stop, waiting for a response, this mode is unsuitable. For byte-by-byte encryption, **cipher feedback mode**, shown in Fig. 7-8, can be used. In this figure, the state of the encryption machine is shown after bytes 0 through 9 have been encrypted and sent. When plaintext byte 10 arrives, as illustrated in Fig. 7-8(a), the DES algorithm operates on the 64-bit shift register to generate a 64-bit ciphertext. The leftmost byte of that ciphertext is extracted and EXCLUSIVE ORed with P_{10}. That byte is transmitted on the transmission line. In addition, the shift register is shifted left 8 bits, causing C_2 to fall off the left end, and C_{10} is inserted in the position just vacated at the right end by C_9. Note that the contents of the shift register depend on the entire previous history of the plaintext, so a pattern that repeats multiple times in the plaintext will be encrypted differently each time in the ciphertext. As with cipher block chaining, an initialization vector is needed to start the ball rolling.

Decryption with cipher feedback mode just does the same thing as encryption. In particular, the contents of the shift register is *encrypted*, not *decrypted*, so the selected byte that is EXCLUSIVE ORed with C_{10} to get P_{10} is the same one that was EXCLUSIVE ORed with P_{10} to generate C_{10} in the first place. As long as the two shift registers remain identical, decryption works correctly.

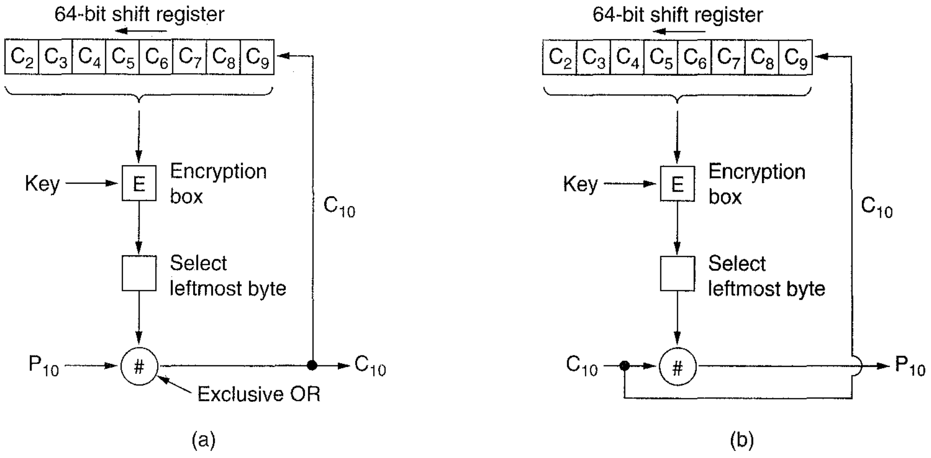


Fig. 7-8. Cipher feedback mode.

As an aside, it should be noted that if one bit of the ciphertext is accidentally inverted during transmission, the 8 bytes that are decrypted while the bad byte is in the shift register will be corrupted. Once the bad byte is pushed out of the shift register, correct plaintext will once again be generated. Thus the effects of a single inverted bit are relatively localized and do not ruin the rest of the message.

Nevertheless, there exist applications in which having a 1-bit transmission error mess up 64 bits of plaintext is too large an effect. For these applications, a fourth option, **output feedback mode**, exists. It is identical to cipher feedback mode, except that the byte fed back into the right end of the shift register is taken from just before the EXCLUSIVE OR box, not just after it.

Output feedback mode has the property that a 1-bit error in the ciphertext causes only a 1-bit error in the resulting plaintext. On the other hand, it is less secure than the other modes, and should be avoided for general-purpose use. Electronic code book mode should also be avoided except under special circumstances (e.g., encrypting a single random number, such as a session key). For normal operation, cipher block chaining should be used when the input arrives in 8-byte units (e.g., for encrypting disk files) and cipher feedback mode should be used for irregular input streams, such as keyboard input.

Breaking DES

DES has been enveloped in controversy from the day it was launched. It was based on a cipher developed and patented by IBM, called Lucifer, except that IBM's cipher used a 128-bit key instead of a 56-bit key. When the U.S. federal government wanted to standardize on one cipher for unclassified use, it "invited"

IBM to "discuss" the matter with NSA, the government's code-breaking arm, which is the world's largest employer of mathematicians and cryptologists. NSA is so secret that an industry joke goes:

Q: What does NSA stand for?
A: No Such Agency.

Actually, NSA stands for National Security Agency.

After these discussions took place, IBM reduced the key from 128 bits to 56 bits and decided to keep secret the process by which DES was designed. Many people suspected that the key length was reduced to make sure that NSA could just break DES, but no organization with a smaller budget could. The point of the secret design was supposedly to hide a trapdoor that could make it even easier for NSA to break DES. When an NSA employee discreetly told IEEE to cancel a planned conference on cryptography, that did not make people any more comfortable.

In 1977, two Stanford cryptography researchers, Diffie and Hellman (1977), designed a machine to break DES and estimated that it could be built for 20 million dollars. Given a small piece of plaintext and matched ciphertext, this machine could find the key by exhaustive search of the 2^{56}-entry key space in under 1 day. Nowadays, such a machine would cost perhaps 1 million dollars. A detailed design for a machine that can break DES by exhaustive search in about four hours is presented in (Wiener, 1994).

Here is another strategy. Although software encryption is 1000 times slower than hardware encryption, a high-end home computer can still do about 250,000 encryptions/sec in software and is probably idle 2 million seconds/month. This idle time could be put to use breaking DES. If someone posted a message to one of the popular Internet newsgroups, it should not be hard to sign up the necessary 140,000 people to check all 7×10^{16} keys in a month.

Probably the most innovative idea for breaking DES is the **Chinese Lottery** (Quisquater and Girault, 1991). In this design, every radio and television has to be equipped with a cheap DES chip capable of performing 1 million encryptions/sec in hardware. Assuming that every one of the 1.2 billion people in China owns a radio or television, whenever the Chinese government wants to decrypt a message encrypted by DES, it just broadcasts the plaintext/ciphertext pair, and each of the 1.2 billion chips begins searching its preassigned section of the key space. Within 60 seconds, one (or more) hits will be found. To ensure that they are reported, the chips could be programmed to display or announce the message:

CONGRATULATIONS! YOU HAVE JUST WON THE CHINESE LOTTERY.
TO COLLECT, PLEASE CALL 1-800-BIG-PRIZE

The conclusion that one can draw from these arguments is that DES should no longer be used for anything important. However, although 2^{56} is a paltry

7×10^{16}, 2^{112} is a magnificent 5×10^{33}. Even with a billion DES chips doing a billion operations per second, it would take 100 million years to exhaustively search a 112-bit key space. Thus the thought arises of just running DES twice, with two different 56-bit keys.

Unfortunately, Merkle and Hellman (1981) have developed a method that makes double encryption suspect. It is called the **meet-in-the-middle** attack and works like this (Hellman, 1980). Suppose that someone has doubly encrypted a series of plaintext blocks, using electronic code book mode. For a few values of *i*, the cryptanalyst has matched pairs $(P_i, C_{i)}$ where

$$C_i = E_{K2}(E_{K1}(P_i))$$

If we now apply the decryption function, D_{K2} to each side of this equation, we get

$$D_{K2}(C_i) = E_{K1}(P_i) \qquad (7\text{-}1)$$

because encrypting x and then decrypting it with the same key gives back x.

The meet-in-the-middle attack uses this equation to find the DES keys, $K1$ and $K2$, as follows:

1. Compute $R_i = E_i(P_1)$ for all 2^{56} values of i, where E is the DES encryption function. Sort this table in ascending order of R_i.
2. Compute $S_j = D_j(C_1)$ for all 2^{56} values of j, where D is the DES decryption function. Sort this table in ascending order of S_j.
3. Scan the first table looking for an R_i that matches some S_j in the second table. When a match is found, we then have a key pair (i,j) such that $D_j(C_1) = E_i(P_1)$. Potentially, i is $K1$ and j is $K2$.
4. Check to see if $E_j(E_i(P_2))$ is equal to C_2. If it is, try all the other (plaintext, ciphertext) pairs. If it is not, continue searching the two tables looking for matches.

Many false alarms will certainly occur before the real keys are located, but eventually they will be found. This attack requires only 2^{57} encryption or decryption operations (to construct the two tables), far less than 2^{112}. However it also requires a total of 2^{60} bytes of storage for the two tables, so it is not currently feasible in this basic form, but Merkle and Hellman have shown various optimizations and trade-offs that permit less storage at the expense of more computing. All in all, double encryption using DES is probably not much more secure than single encryption.

Triple encryption is another matter. As early as 1979, IBM realized that the DES key length was too short and devised a way to effectively increase it using triple encryption (Tuchman, 1979). The method chosen, which has since been incorporated in International Standard 8732, is illustrated in Fig. 7-9. Here two keys and three stages are used. In the first stage, the plaintext is encrypted with K_1. In the second stage, DES is run in decryption mode, using K_2 as the key. Finally, another encryption is done with K_1.

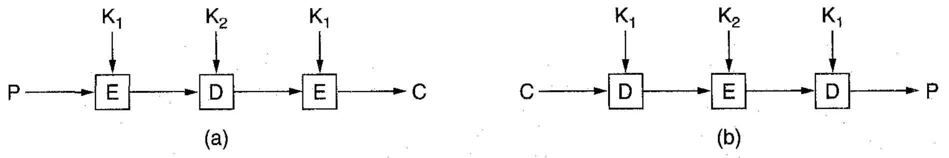


Fig. 7-9. Triple encryption using DES.

This design immediately gives rise to two questions. First, why are only two keys used, instead of three? Second, why is EDE used, instead of EEE? The reason that two keys are used is that even the most paranoid cryptographers concede that 112 bits is sufficient for commercial applications for the time being. Going to 168 bits would just add the unnecessary overhead of managing and transporting another key.

The reason for encrypting, decrypting, and then encrypting again is backward compatibility with existing single-key DES systems. Both the encryption and decryption functions are mappings between sets of 64-bit numbers. From a cryptographic point of view, the two mappings are equally strong. By using EDE, however, instead of EEE, a computer using triple encryption can speak to one using single encryption by just setting $K_1 = K_2$. This property allows triple encryption to be phased in gradually, something of no concern to academic cryptographers, but of considerable import to IBM and its customers.

No method is known for breaking triple DES in EDE mode. Van Oorschot and Wiener (1988) have presented a method to speed up the search of EDE by a factor of 16, but even with their attack, EDE is highly secure. For anyone wishing nothing less than the very best, EEE with three distinct 56-bit keys (168 bits in all) is recommended.

Before leaving the subject of DES, it is worth at least mentioning two recent developments in cryptanalysis. The first development is **differential cryptanalysis** (Biham and Shamir, 1993). This technique can be used to attack any block cipher. It works by beginning with a pair of plaintext blocks that differ in only a small number of bits and watching carefully what happens on each internal iteration as the encryption proceeds. In many cases, some patterns are much more common than other patterns, and this observation leads to a probabilistic attack.

The other development worth noting is **linear cryptanalysis** (Matsui, 1994). It can break DES with only 2^{43} known plaintexts. It works by EXCLUSIVE ORing certain plaintext and ciphertext bits together to generate 1 bit. When done repeatedly, half the bits should be 0s and half should be 1s. Often, however, ciphers introduce a bias in one direction or the other, and this bias, however small, can be exploited to reduce the work factor. For the details, see Matsui's paper.

IDEA

Perhaps all this hammering on why DES is insecure is like beating a dead horse, but the reality is that singly-encrypted DES is still widely used for secure applications, such as banking using automated teller machines. While this choice was probably appropriate when it was made, a decade or more ago, it is no longer adequate.

At this point, the reader is probably legitimately wondering: "If DES is so weak, why hasn't anyone invented a better block cipher?" The fact is, many other block ciphers have been proposed, including BLOWFISH (Schneier, 1994), Crab (Kaliski and Robshaw, 1994), FEAL (Shimizu and Miyaguchi, 1988), KHAFRE (Merkle, 1991), LOKI91 (Brown et al., 1991), NEWDES (Scott, 1985), REDOC-II (Cusick and Wood, 1991), and SAFER K64 (Massey, 1994). Schneier (1996) discusses all of these and innumerable others. Probably the most interesting and important of the post-DES block ciphers is **IDEA** the (**International Data Encryption Algorithm**) (Lai and Massey, 1990; and Lai, 1992). Let us now study IDEA in more detail.

IDEA was designed by two researchers in Switzerland, so it is probably free of any NSA "guidance" that might have introduced a secret trapdoor. It uses a 128-bit key, which will make it immune to brute force, Chinese lottery, and meet-in-the-middle attacks for decades to come. It was also designed to withstand differential cryptanalysis. No currently known technique or machine is thought to be able to break IDEA.

The basic structure of the algorithm resembles DES in that 64-bit plaintext input blocks are mangled in a sequence of parameterized iterations to produce 64-bit ciphertext output blocks, as shown in Fig. 7-10(a). Given the extensive bit mangling (for every iteration, every output bit depends on every input bit), eight iterations are sufficient. As with all block ciphers, IDEA can also be used in cipher feedback mode and the other DES modes.

The details of one iteration are depicted in Fig. 7-10(b). Three operations are used, all on unsigned 16-bit numbers. These operations are EXCLUSIVE OR, addition modulo 2^{16}, and multiplication modulo $2^{16} + 1$. All three of these can easily be done on a 16-bit microcomputer by ignoring the high-order parts of results. The operations have the property that no two pairs obey the associative law or distributive law, making cryptanalysis more difficult. The 128-bit key is used to generate 52 subkeys of 16 bits each, 6 for each of eight iterations and 4 for the final transformation. Decryption uses the same algorithm as encryption, only with different subkeys.

Both software and hardware implementations of IDEA have been constructed. The first software implementation ran on a 33-MHz 386 and achieved an encryption rate of 0.88 Mbps. On a modern machine running ten times as fast, 9 Mbps should be achievable in software. An experimental 25-MHz VLSI chip was built at ETH Zurich and encrypted at a rate of 177 Mbps.

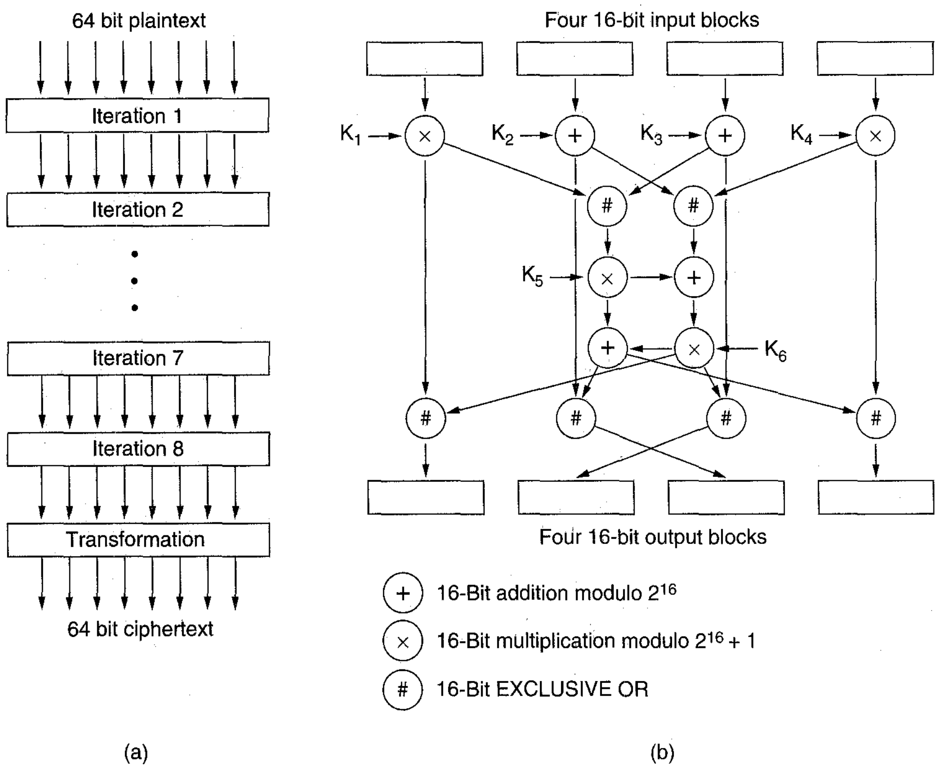


Fig. 7-10. (a) IDEA. (b) Detail of one iteration.

7.1.4. Public-Key Algorithms

Historically the key distribution problem has always been the weak link in most cryptosystems. No matter how strong a cryptosystem was, if an intruder could steal the key, the system was worthless. Since all cryptologists always took for granted that the encryption key and decryption key were the same (or easily derived from one another) and the key had to be distributed to all users of the system, it seemed as if there was an inherent built-in problem: keys had to protected from theft, but they also had to be distributed, so they could not just be locked up in a bank vault.

In 1976, two researchers at Stanford University, Diffie and Hellman (1976), proposed a radically new kind of cryptosystem, one in which the encryption and decryption keys were different, and the decryption key could not be derived from the encryption key. In their proposal, the (keyed) encryption algorithm, *E*, and the (keyed) decryption algorithm, D, had to meet the following three requirements. These requirements can be stated simply as follows:

1. $D(E(P)) = P$.
2. It is exceedingly difficult to deduce D from E.
3. E cannot be broken by a chosen plaintext attack.

The first requirement says that if we apply D to an encrypted message, $E(P)$, we get the original plaintext message, P, back. The second requirement speaks for itself. The third requirement is needed because, as we shall see in a moment, intruders may experiment with the algorithm to their hearts' content. Under these conditions, there is no reason that the encryption key cannot be made public.

The method works like this. A person, say, Alice, wanting to receive secret messages, first devises two algorithms, E_A and D_A, meeting the above requirements. The encryption algorithm and key, E_A, is then made public, hence the name **public-key cryptography** (to contrast it with traditional secret-key cryptography). This might be done by putting it in a file that anyone who wanted to could read. Alice publishes the decryption algorithm (to get the free consulting), but keeps the decryption key secret. Thus, E_A is public, but D_A is private.

Now let us see if we can solve the problem of establishing a secure channel between Alice and Bob, who have never had any previous contact. Both Alice's encryption key, E_A, and Bob's encryption key, E_B, are assumed to be in a publicly readable file. (Basically, all users of the network are expected to publish their encryption keys as soon as they become network users.) Now Alice takes her first message, P, computes $E_B(P)$, and sends it to Bob. Bob then decrypts it by applying his secret key D_B [i.e., he computes $D_B(E_B(P)) = P$]. No one else can read the encrypted message, $E_B(P)$, because the encryption system is assumed strong and because it is too difficult to derive D_B from the publicly known E_B. Alice and Bob can now communicate securely.

A note on terminology is perhaps useful here. Public-key cryptography requires each user to have two keys: a public key, used by the entire world for encrypting messages to be sent to that user, and a private key, which the user needs for decrypting messages. We will consistently refer to these keys as the *public* and *private* keys, respectively, and distinguish them from the *secret* keys used for both encryption and decryption in conventional (also called **symmetric key**) cryptography.

The RSA Algorithm

The only catch is that we need to find algorithms that indeed satisfy all three requirements. Due to the potential advantages of public-key cryptography, many researchers are hard at work, and some algorithms have already been published. One good method was discovered by a group at M.I.T. (Rivest et al., 1978). It is known by the initials of the three discoverers (Rivest, Shamir, Adleman): **RSA**. Their method is based on some principles from number theory. We will now summarize how to use the method below; for details, consult the paper.

1. Choose two large primes, p and q, (typically greater than 10^{100}).
2. Compute $n = p \times q$ and $z = (p - 1) \times (q - 1)$.
3. Choose a number relatively prime to z and call it d.
4. Find e such that $e \times d = 1$ *mod* z.

With these parameters computed in advance, we are ready to begin encryption. Divide the plaintext (regarded as a bit string) into blocks, so that each plaintext message, P, falls in the interval $0 \le P < n$. This can be done by grouping the plaintext into blocks of k bits, where k is the largest integer for which $2^k < n$ is true.

To encrypt a message, P, compute $C = P^e$ (mod n). To decrypt C, compute $P = C^d$ (mod n). It can be proven that for all P in the specified range, the encryption and decryption functions are inverses. To perform the encryption, you need e and n. To perform the decryption, you need d and n. Therefore, the public key consists of the pair (e, n) and the private key consists of (d, n).

The security of the method is based on the difficulty of factoring large numbers. If the cryptanalyst could factor the (publicly known) n, he could then find p and q, and from these z. Equipped with knowledge of z and e, d can be found using Euclid's algorithm. Fortunately, mathematicians have been trying to factor large numbers for at least 300 years, and the accumulated evidence suggests that it is an exceedingly difficult problem.

According to Rivest and colleagues, factoring a 200-digit number requires 4 billion years of computer time; factoring a 500-digit number requires 10^{25} years. In both cases, they assume the best known algorithm and a computer with a 1−μsec instruction time. Even if computers continue to get faster by an order of magnitude per decade, it will be centuries before factoring a 500-digit number becomes feasible, at which time our descendants can simply choose p and q still larger.

A trivial pedagogical example of the RSA algorithm is given in Fig. 7-11. For this example we have chosen $p = 3$ and $q = 11$, giving $n = 33$ and $z = 20$. A suitable value for d is $d = 7$, since 7 and 20 have no common factors. With these choices, e can be found by solving the equation $7e = 1$ (mod 20), which yields $e = 3$. The ciphertext, C, for a plaintext message, P, is given by $C = P^3$ (mod 33). The ciphertext is decrypted by the receiver according to the rule $P = C^7$ (mod 33). The figure shows the encryption of the plaintext "SUZANNE" as an example.

Because the primes chosen for this example are so small, P must be less than 33, so each plaintext block can contain only a single character. The result is a

Plaintext (P)			Ciphertext (C)		After decryption	
Symbolic	Numeric	P^3	P^3 (mod 33)	C^7	C^7 (mod 33)	Symbolic
S	19	6859	28	13492928512	19	S
U	21	9261	21	1801088541	21	U
Z	26	17576	20	1280000000	26	Z
A	01	1	1	1	1	A
N	14	2744	5	78125	14	N
N	14	2744	5	78125	14	N
E	05	125	26	8031810176	5	E

Sender's computation

Receiver's computation

Fig. 7-11. An example of the RSA algorithm.

monoalphabetic substitution cipher, not very impressive. If instead we had chosen p and $q \approx 10^{100}$, we would have $n \approx 10^{200}$, so each block could be up to 664 bits ($2^{664} \approx 10^{200}$) or 83 8-bit characters, versus 8 characters for DES.

It should be pointed out that using RSA as we have described is similar to using DES in ECB mode—the same input block gives the same output block. Therefore some form of chaining is needed for data encryption. However, in practice, most RSA-based systems use public-key cryptography primarily for distributing one-time session keys for use with DES, IDEA, or similar algorithms. RSA is too slow for actually encrypting large volumes of data.

Other Public-Key Algorithms

Although RSA is widely used, it is by no means the only public-key algorithm known. The first public-key algorithm was the knapsack algorithm (Merkle and Hellman, 1978). The idea here is that someone owns a large number of objects, each with a different weight. The owner encodes the message by secretly selecting a subset of the objects and placing them in the knapsack. The total weight of the objects in the knapsack is made public, as is the list of all possible objects. The list of objects in the knapsack is kept secret. With certain additional restrictions, the problem of figuring out a possible list of objects with the given weight was thought to be computationally infeasible, and formed the basis of the public-key algorithm.

The algorithm's inventor, Ralph Merkle, was quite sure that this algorithm could not be broken, so he offered a 100-dollar reward to anyone who could break it. Adi Shamir (the "S" in RSA) promptly broke it and collected the reward. Undeterred, Merkle strengthened the algorithm and offered a 1000-dollar reward to anyone who could break the new one. Ron Rivest (the "R" in RSA) promptly broke the new one and collected the reward. Merkle did not dare offer 10,000 dollars for the next version, so "A" (Leonard Adleman) was out of luck. Although it has been patched up again, the knapsack algorithm is not considered secure and is rarely used.

Other public-key schemes are based on the difficulty of computing discrete logarithms (Rabin, 1979). Algorithms that use this principle have been invented by El Gamal (1985) and Schnorr (1991).

A few other schemes exist, such as those based on elliptic curves (Menezes and Vanstone, 1993), but the three major categories are those based on the difficulty of factoring large numbers, computing discrete logarithms, and determining the contents of a knapsack from its weight. These problems are thought to be genuinely difficult to solve because mathematicians have been working on them for many years without any great breakthroughs.

7.1.5. Authentication Protocols

Authentication is the technique by which a process verifies that its communication partner is who it is supposed to be and not an imposter. Verifying the identity of a remote process in the face of a malicious, active intruder is surprisingly difficult and requires complex protocols based on cryptography. In this section, we will study some of the many authentication protocols that are used on insecure computer networks.

As an aside, some people confuse authorization with authentication. Authentication deals with the question of whether or not you are actually communicating with a specific process. Authorization is concerned with what that process is permitted to do. For example, a client process contacts a file server and says: "I am Scott's process and I want to delete the file *cookbook.old*." From the file server's point of view, two questions must be answered:

1. Is this actually Scott's process (authentication)?
2. Is Scott allowed to delete *cookbook.old* (authorization)?

Only after both questions have been unambiguously answered in the affirmative can the requested action take place. The former question is really the key one. Once the file server knows whom it is talking to, checking authorization is just a matter of looking up entries in local tables. For this reason, we will concentrate on authentication in this section.

The general model that all authentication protocols use is this. An initiating user (really a process), say, Alice, wants to establish a secure connection with a second user, Bob. Alice and Bob are sometimes called **principals**, the main characters in our story. Bob is a banker with whom Alice would like to do business. Alice starts out by sending a message either to Bob, or to a trusted **key distribution center (KDC)**, which is always honest. Several other message exchanges follow in various directions. As these message are being sent, a nasty intruder, Trudy,[†] may intercept, modify, or replay them in order to trick Alice and Bob or just to gum up the works.

Nevertheless, when the protocol has been completed, Alice is sure she is talking to Bob and Bob is sure he is talking to Alice. Furthermore, in most of the protocols, the two of them will also have established a secret **session key** for use in the upcoming conversation. In practice, for performance reasons, all data traffic is encrypted using secret-key cryptography, although public-key cryptography is widely used for the authentication protocols themselves and for establishing the session key.

The point of using a new, randomly-chosen session key for each new connection is to minimize the amount of traffic that gets sent with the users' secret keys or public keys, to reduce the amount of ciphertext an intruder can obtain, and to minimize the damage done if a process crashes and its core dump falls into the wrong hands. Hopefully, the only key present then will be the session key. All the permanent keys should have been carefully zeroed out after the session was established.

Authentication Based on a Shared Secret Key

For our first authentication protocol, we will assume that Alice and Bob already share a secret key, K_{AB} (In the formal protocols, we will abbreviate Alice as A and Bob as B, respectively.) This shared key might have been agreed upon on the telephone, or in person, but, in any event, not on the (insecure) network.

This protocol is based on a principle found in many authentication protocols: one party sends a random number to the other, who then transforms it in a special way and then returns the result. Such protocols are called **challenge-response** protocols. In this and subsequent authentication protocols, the following notation will be used:

A, B are the identities of Alice and Bob
R_i's are the challenges, where the subscript identifies the challenger
K_i are keys, where i indicates the owner; K_S is the session key

The message sequence for our first shared-key authentication protocol is shown in Fig. 7-12. In message 1, Alice sends her identity, A, to Bob in a way that Bob understands. Bob, of course, has no way of knowing whether this message came from Alice or from Trudy, so he chooses a challenge, a large random number, R_B, and sends it back to "Alice" as message 2, in plaintext. Alice then encrypts the message with the key she shares with Bob and sends the ciphertext, $K_{AB}(R_B)$, back in message 3. When Bob sees this message, he immediately knows that it came from Alice because Trudy does not know K_{AB} and thus could

[†] I thank Kaufman$_1$ et al.$_{23}$ (1995) for revealing her name.

not have generated it. Furthermore, since R_B was chosen randomly from a large space (say, 128-bit random numbers), it is very unlikely that Trudy would have seen R_B and its response from an earlier session.

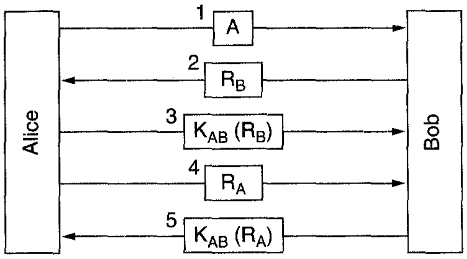


Fig. 7-12. Two-way authentication using a challenge-response protocol.

At this point, Bob is sure he is talking to Alice, but Alice is not sure of anything. For all Alice knows, Trudy might have intercepted message 1 and sent back R_B in response. Maybe Bob died last night. To find out whom she is talking to, Alice picks a random number, R_A and sends it to Bob as plaintext, in message 4. When Bob responds with $K_{AB}(R_A)$, Alice knows she is talking to Bob. If they wish to establish a session key now, Alice can pick one, K_S, and send it to Bob encrypted with K_{AB}.

Although the protocol of Fig. 7-12 works, it contains extra messages. These can be eliminated by combining information, as illustrated in Fig. 7-13. Here Alice initiates the challenge-response protocol instead of waiting for Bob to do it. Similarly, while he is responding to Alice's challenge, Bob sends his own. The entire protocol can be reduced to three messages instead of five.

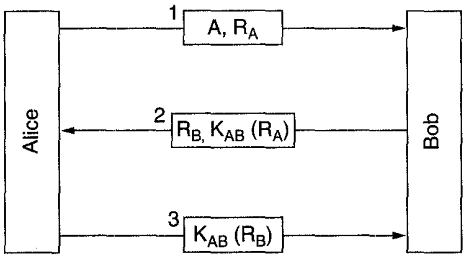


Fig. 7-13. A shortened two-way authentication protocol.

Is this new protocol an improvement over the original one? In one sense it is: it is shorter. Unfortunately, it is also wrong. Under certain circumstances, Trudy can defeat this protocol by using what is known as a **reflection attack**. In particular, Trudy can break it if it is possible to open multiple sessions with Bob at once. This situation would be true, for example, if Bob is a bank and is prepared to accept many simultaneous connections from teller machines at once.

Trudy's reflection attack is shown in Fig. 7-14. It starts out with Trudy claiming she is Alice and sending R_T. Bob responds, as usual, with his own challenge, R_B. Now Trudy is stuck. What can she do? She does not know $K_{AB}(R_B)$.

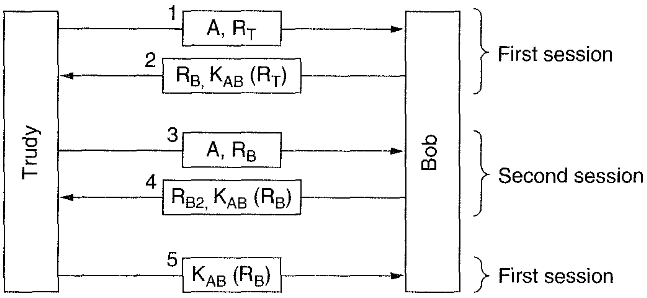


Fig. 7-14. The reflection attack.

She can open a second session with message 3, supplying the R_B taken from message 2 as her challenge. Bob calmly encrypts it and sends back $K_{AB}(R_B)$ in message 4. Now Trudy has the missing information, so she can complete the first session and abort the second one. Bob is now convinced that Trudy is Alice, so when she asks for her bank account balance, he gives it to her without question. Then when she asks him to transfer it all to a secret bank account in Switzerland, he does so without a moment's hesitation.

The moral of this story is:

Designing a correct authentication protocol is harder than it looks.

Three general rules that often help are as follows:

1. Have the initiator prove who she is before the responder has to. In this case, Bob gives away valuable information before Trudy has to give any evidence of who she is.

2. Have the initiator and responder use different keys for proof, even if this means having two shared keys, K_{AB} and K'_{AB}.

3. Have the initiator and responder draw their challenges from different sets. For example, the initiator must use even numbers and the responder must use odd numbers.

All three rules were violated here, with disastrous results. Note that our first (five-message) authentication protocol requires Alice to prove her identity first, so that protocol is not subject to the reflection attack.

Establishing a Shared Key: The Diffie-Hellman Key Exchange

So far we have assumed that Alice and Bob share a secret key. Suppose that they do not? How can they establish one? One way would be for Alice to call Bob and give him her key on the phone, but he would probably start out by saying: "How do I know you are Alice and not Trudy?" They could try to arrange a meeting, with each one bringing a passport, a drivers' license, and three major credit cards, but being busy people, they might not be able to find a mutually acceptable date for months. Fortunately, incredible as it may sound, there is a way for total strangers to establish a shared secret key in broad daylight, even with Trudy carefully recording every message.

The protocol that allows strangers to establish a shared secret key is called the **Diffie-Hellman key exchange** (Diffie and Hellman, 1976) and works as follows. Alice and Bob have to agree on two large prime numbers, n, and g, where $(n - 1)/2$ is also a prime and certain conditions apply to g. These numbers may be public, so either one of them can just pick n and g and tell the other openly. Now Alice picks a large (say, 512-bit) number, x, and keeps it secret. Similarly, Bob picks a large secret number, y.

Alice initiates the key exchange protocol by sending Bob a message containing $(n, g, g^x \bmod n)$, as shown in Fig. 7-15. Bob responds by sending Alice a message containing $g^y \bmod n$. Now Alice takes the number Bob sent her and raises it to the xth power to get $(g^y \bmod n)^x$. Bob performs a similar operation to get $(g^x \bmod n)^y$. By the laws of modular arithmetic, both calculations yield $g^{xy} \bmod n$. Lo and behold, Alice and Bob now share a secret key, $g^{xy} \bmod n$.

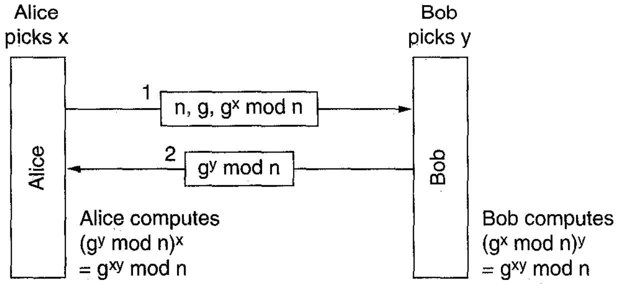


Fig. 7-15. The Diffie-Hellman key exchange.

Trudy, of course, has seen both messages. She knows g and n from message 1. If she could compute x and y, she could figure out the secret key. The trouble is, given only $g^x \bmod n$, she cannot find x. No practical algorithm for computing discrete logarithms modulo a very large prime number is known.

To make the above example more concrete, we will use the (completely unrealistic) values of $n = 47$ and $g = 3$. Alice picks $x = 8$ and Bob picks $y = 10$.

Both of these are kept secret. Alice's message to Bob is (47, 3, 28) because $3^8 \bmod 47$ is 28. Bob's message to Alice is (17). Alice computes $17^8 \bmod 47$, which is 4. Bob computes $28^{10} \bmod 47$, which is 4. Alice and Bob have independently determined that the secret key is now 4. Trudy has to solve the equation $3^x \bmod 47 = 28$, which can be done by exhaustive search for small numbers like this, but not when all the numbers are hundreds of bits long. All currently-known algorithms simply take too long, even using a massively parallel supercomputer.

Despite the elegance of the Diffie-Hellman algorithm, there is a problem: when Bob gets the triple (47, 3, 28), how does he know it is from Alice and not from Trudy? There is no way he can know. Unfortunately, Trudy can exploit this fact to deceive both Alice and Bob, as illustrated in Fig. 7-16. Here, while Alice and Bob are choosing x and y, respectively, Trudy picks her own random number, z. Alice sends message 1 intended for Bob. Trudy intercepts it and sends message 2 to Bob, using the correct g and n (which are public anyway) but with her own z instead of x. She also sends message 3 back to Alice. Later Bob sends message 4 to Alice, which Trudy again intercepts and keeps.

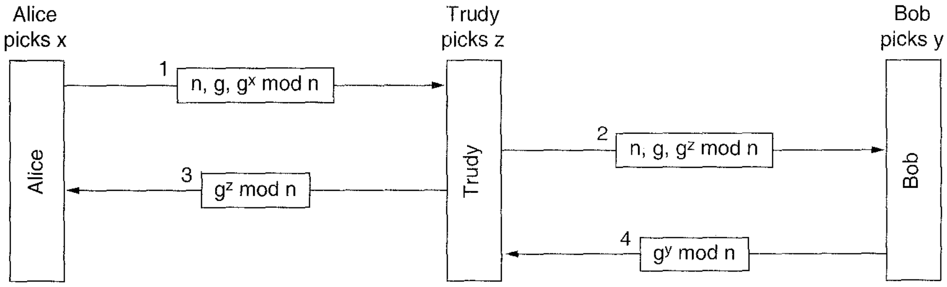


Fig. 7-16. The bucket brigade attack.

Now everybody does the modular arithmetic. Alice computes the secret key as $g^{xz} \bmod n$, and so does Trudy (for messages to Alice). Bob computes $g^{yz} \bmod n$ and so does Trudy (for messages to Bob). Alice thinks she is talking to Bob so she establishes a session key (with Trudy). So does Bob. Every message that Alice sends on the encrypted session is captured by Trudy, stored, modified if desired, and then (optionally) passed on to Bob. Similarly in the other direction. Trudy sees everything and can modify all messages at will, while both Alice and Bob are under the illusion that they have a secure channel to one another. This attack is known as the **bucket brigade attack**, because it vaguely resembles an old-time volunteer fire department passing buckets along the line from the fire truck to the fire. It is also called the **(wo)man-in-the-middle attack**, which should not be confused with the meet-in-the-middle attack on block ciphers. Fortunately, more complex algorithms can defeat this attack.

Authentication Using a Key Distribution Center

Setting up a shared secret with a stranger almost worked, but not quite. On the other hand, it probably was not worth doing in the first place (sour grapes attack). To talk to *n* people this way, you would need *n* keys. For popular people, key management would become a real burden, especially if each key had to be stored on a separate plastic chip card.

A different approach is to introduce a trusted key distribution center (KDC). In this model, each user has a single key shared with the KDC. Authentication and session key management now goes through the KDC. The simplest known KDC authentication protocol involving two parties and a trusted KDC is depicted in Fig. 7-17.

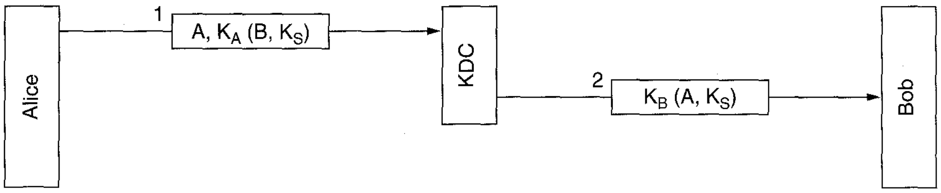


Fig. 7-17. A first attempt at an authentication protocol using a KDC.

The idea behind this protocol is simple: Alice picks a session key, K_S, and tells the KDC that she wants to talk to Bob using K_S. This message is encrypted with the secret key Alice shares (only) with the KDC, K_A. The KDC decrypts this message, extracting Bob's identity and the session key. It then constructs a new message containing Alice's identity and the session key and sends this message to Bob. This encryption is done with K_B, the secret key Bob shares with the KDC. When Bob decrypts the message, he learns that Alice wants to talk to him, and which key she wants to use.

The authentication here happens for free. The KDC knows that message 1 must have come from Alice, since no one else would have been able to encrypt it with Alice's secret key. Similarly, Bob knows that message 2 must have come from the KDC, whom he trusts, since no one else knows his secret key.

Unfortunately, this protocol has a serious flaw. Trudy needs some money, so she figures out some legitimate service she can perform for Alice, makes an attractive offer, and gets the job. After doing the work, Trudy then politely requests Alice to pay by bank transfer. Alice then establishes a session key with her banker, Bob. Then she sends Bob a message requesting money to be transferred to Trudy's account.

Meanwhile, Trudy is back to her old ways, snooping on the network. She copies both message 2 in Fig. 7-17, and the money-transfer request that follows it.

Later, she replays both of them to Bob. Bob gets them and thinks: "Alice must have hired Trudy again. She clearly does good work." Bob then transfers an equal amount of money from Alice's account to Trudy's. Some time after the 50th message pair, Bob runs out of the office to find Trudy to offer her a big loan so she can expand her obviously successful business. This problem is called the **replay attack**.

Several solutions to the replay attack are possible. The first one is to include a timestamp in each message. Then if anyone receives an obsolete message, it can be discarded. The trouble with this approach is that clocks are never exactly synchronized over a network, so there has to be some interval during which a timestamp is valid. Trudy can replay the message during this interval and get away with it.

The second solution is to put a one-time, unique message number, usually called a **nonce**, in each message. Each party then has to remember all previous nonces and reject any message containing a previously used nonce. But nonces have to be remembered forever, lest Trudy try replaying a 5-year-old message. Also, if some machine crashes and it loses its nonce list, it is again vulnerable to a replay attack. Timestamps and nonces can be combined to limit how long nonces have to be remembered, but clearly the protocol is going to get a lot more complicated.

A more sophisticated approach to authentication is to use a multiway challenge-response protocol. A well-known example of such a protocol is the **Needham-Schroeder authentication** protocol (Needham and Schroeder, 1978), one variant of which is shown in Fig. 7-18.

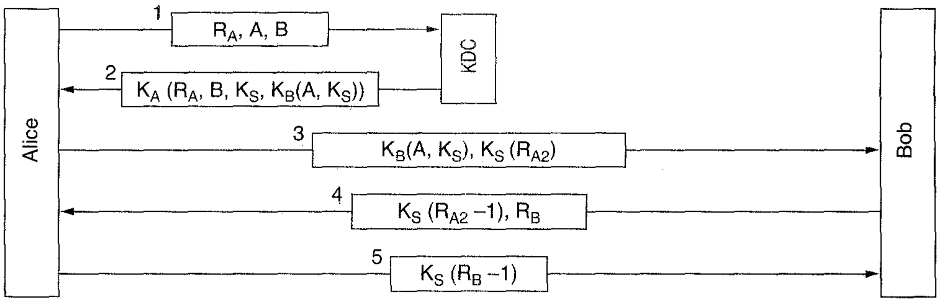

Alice
KDC
Bob
1
R_A, A, B
2
K_A (R_A, B, K_S, K_B(A, K_S))
3
K_B(A, K_S), K_S (R_{A2})
4
K_S (R_{A2} −1), R_B
5
K_S (R_B −1)

Fig. 7-18. The Needham-Schroeder authentication protocol.

The protocol begins with Alice telling the KDC that she wants to talk to Bob. This message contains a large random number, R_A, as a nonce. The KDC sends back message 2 containing Alice's random number, a session key, and a ticket that she can send to Bob. The point of the random number, R_A, is to assure Alice that message 2 is fresh, and not a replay. Bob's identity is also enclosed in case Trudy gets any funny ideas about replacing B in message 1 with her own identity so the KDC will encrypt the ticket at the end of message 2 with K_T instead of K_B. The ticket encrypted with K_B is included inside the encrypted message to prevent Trudy from replacing it with something else on the way back to Alice.

Alice now sends the ticket to Bob, along with a new random number, R_{A2}, encrypted with the session key, K_S. In message 4, Bob sends back $K_S(R_{A2} - 1)$ to prove to Alice that she is talking to the real Bob. Sending back $K_S(R_{A2})$ would not have worked, since Trudy could just have stolen it from message 3.

After receiving message 4, Alice is now convinced that she is talking to Bob, and that no replays could have been used so far. After all, she just generated R_{A2} a few milliseconds ago. The purpose of message 5 is to convince Bob that it is indeed Alice he is talking to, and no replays are being used here either. By having each party both generate a challenge and respond to one, the possibility of any kind of replay attack is eliminated.

Although this protocol seems pretty solid, it does have a slight weakness. If Trudy ever manages to obtain an old session key in plaintext, she can initiate a new session with Bob replaying the message 3 corresponding to the compromised key and convince him that she is Alice (Denning and Sacco, 1981). This time she can plunder Alice's bank account without having to perform the legitimate service even once.

Needham and Schroeder later published a protocol that corrects this problem (Needham and Schroeder, 1987). In the same issue of the same journal, Otway and Rees (1987) also published a protocol that solves the problem in a shorter way. Figure 7-19 shows a slightly modified Otway-Rees protocol.

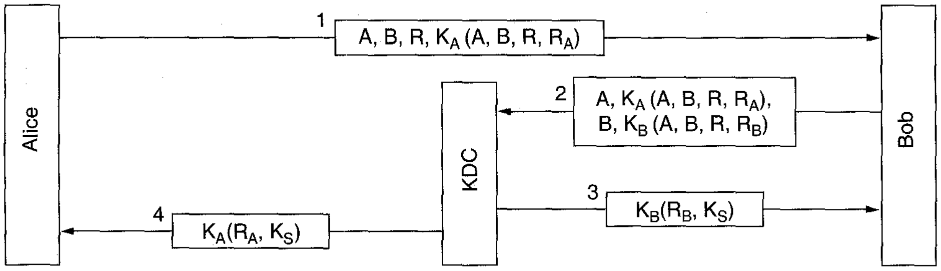

Alice
1
A, B, R, K_A (A, B, R, R_A)
KDC
2
A, K_A (A, B, R, R_A),
B, K_B (A, B, R, R_B)
Bob
3
K_B(R_B, K_S)
4
K_A(R_A, K_S)

Fig. 7-19. The Otway-Rees authentication protocol (slightly simplified).

In the Otway-Rees protocol, Alice starts out by generating a pair of random numbers, R, which will be used as a common identifier, and R_A which Alice will use to challenge Bob. When Bob gets this message, he constructs a new message from the encrypted part of Alice's message, and an analogous one of his own. Both the parts encrypted with K_A and K_B identify Alice and Bob, contain the common identifier, and contain a challenge.

The KDC checks to see if the R in both parts is the same. It might not be because Trudy tampered with R in message 1 or replaced part of message 2. If the two *R*s match, the KDC believes that the request message from Bob is valid. It then generates a session key and encrypts it twice, once for Alice and once for Bob. Each message contains the receiver's random number, as proof that the KDC, and not Trudy, generated the message. At this point both Alice and Bob are in possession of the same session key and can start communicating. The first time they exchange data messages, each one can see that the other one has an identical copy of K_S, so the authentication is then complete.

Authentication Using Kerberos

An authentication protocol used in many real systems is **Kerberos**, which is based on a variant of Needham-Schroeder. It is named for a multiheaded dog in Greek Mythology that used to guard the entrance to Hades (presumably to keep undesirables out). Kerberos was designed at M.I.T. to allow workstation users to access network resources in a secure way. Its biggest difference with Needham-Schroeder is its assumption that all clocks are fairly-well synchronized. The protocol has gone through several iterations. V4 is the version most widely used in industry, so we will describe it. Afterward, we will say a few words about its successor, V5. For more information, see (Neuman and Ts'o, 1994; and Steiner et al., 1988).

Kerberos involves three servers in addition to Alice (a client workstation):

Authentication Server (AS): verifies users during login
Ticket-Granting Server (TGS): issues "proof of identity tickets"
Bob the server: actually does the work Alice wants performed

AS is similar to a KDC in that it shares a secret password with every user. The TGS's job is to issue tickets that can convince the real servers that the bearer of a TGS ticket really is who he or she claims to be.

To start a session, Alice sits down at a arbitrary public workstation and types her name. The workstation sends her name to the AS in plaintext, as shown in Fig. 7-20. What comes back is a session key and a ticket, $K_{TGS}(A, K_S)$, intended for the TGS. These items are packaged together and encrypted using Alice's secret key, so that only Alice can decrypt them. Only when message 2 arrives, does the workstation ask for Alice's password. The password is then used to generate K_A, in order to decrypt message 2 and obtain the session key and TGS ticket inside it. At this point, the workstation overwrites Alice's password, to make sure that it is only inside the workstation for a few milliseconds at most. If Trudy tries logging in as Alice, the password she types will be wrong and the workstation will detect this because the standard part of message 2 will be incorrect.

After she logs in, Alice may tell the workstation that she wants to contact Bob the file server. The workstation then sends message 3 to the TGS asking for a ticket to use with Bob. The key element in this request is $K_{TGS}(A, K_S)$, which is

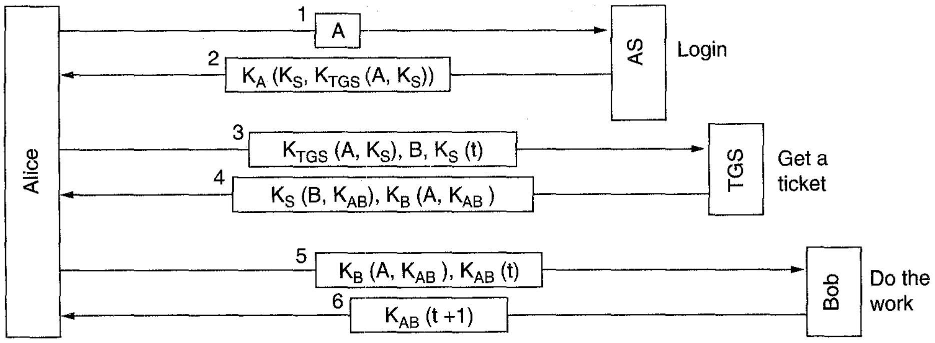

Alice
1 A
AS
Login
2 K_A (K_S, K_TGS (A, K_S))
3 K_TGS (A, K_S), B, K_S (t)
TGS
Get a ticket
4 K_S (B, K_AB), K_B (A, K_AB)
5 K_B (A, K_AB), K_AB (t)
Bob
Do the work
6 K_AB (t +1)

Fig. 7-20. The operation of Kerberos V4.

encrypted with the TGS's secret key and is used as proof that the sender really is Alice. The TGS responds by creating a session key, K_{AB}, for Alice to use with Bob. Two versions of it are sent back. The first is encrypted with only K_S, so Alice can read it. The second is encrypted with Bob's key, K_B, so Bob can read it.

Trudy can copy message 3 and try to use it again, but she will be foiled by the encrypted timestamp, *t*, sent along with it. Trudy cannot replace the timestamp with a more recent one, because she does not know K_S, the session key Alice uses to talk to the TGS. Even if Trudy replays message 3 quickly, all she will get is another copy of message 4, which she could not decrypt the first time and will not be able to decrypt the second time either.

Now Alice can send K_{AB} to Bob to establish a session with him. This exchange is also timestamped. The response is proof to Alice that she is actually talking to Bob, not to Trudy.

After this series of exchanges, Alice can communicate with Bob under cover of K_{AB}. If she later decides she needs to talk to another server, Carol, she just repeats message 3 to the TGS, only now specifying *C* instead of *B*. The TGS will promptly respond with a ticket encrypted with K_C that Alice can send to Carol and that Carol will accept as proof that it came from Alice.

The point of all this work is that now Alice can access servers all over the network in a secure way, and her password never has to go over the network. In fact, it only had to be in her own workstation for a few milliseconds. However, note that each server does its own authorization. When Alice presents her ticket to Bob, this merely proves to Bob who sent it. Precisely what Alice is allowed to do is up to Bob.

Since the Kerberos designers did not expect the entire world to trust a single authentication server, they made provision for having multiple **realms**, each with its own AS and TGS. To get a ticket for a server in a distant realm, Alice would ask her own TGS for a ticket accepted by the TGS in the distant realm. If the distant TGS has registered with the local TGS (the same way local servers do), the local TGS will give Alice a ticket valid at the distant TGS. She can then do business over there, such as getting tickets for servers in that realm. Note, however, that for parties in two realms to do business, each one must trust the other's TGS.

Kerberos V5 is fancier than V4 and has more overhead. It also uses OSI ASN.1 (Abstract Syntax Notation 1) for describing data types and has small changes in the protocols. Furthermore, it has longer ticket lifetimes, allows tickets to be renewed, and will issue postdated tickets. In addition, at least in theory, it is not DES dependent, as V4 is, and supports multiple realms.

Authentication Using Public-Key Cryptography

Mutual authentication can also be done using public-key cryptography. To start with, let us assume Alice and Bob already know each other's public keys (a nontrivial issue). They want to establish a session, and then use secret-key cryptography on that session, since it is typically 100 to 1000 times faster than public-key cryptography. The purpose of the initial exchange then is to authenticate each other and agree on a secret shared session key.

This setup can be done is various ways. A typical one is shown in Fig. 7-21. Here Alice starts by encrypting her identity and a random number, R_A, using Bob's public (or encryption) key, E_B. When Bob receives this message, he has no idea of whether it came from Alice or from Trudy, but he plays along and sends Alice back a message containing Alice's R_A, his own random number, R_B, and a proposed session key, K_S.

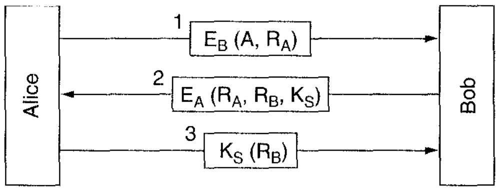


Fig. 7-21. Mutual authentication using public-key cryptography.

When Alice gets message 2, she decrypts it using her private key. She sees R_A in it, which gives her a warm feeling inside. The message must have come from Bob, since Trudy has no way of determining R_A. Furthermore, it must be fresh and not a replay, since she just sent Bob R_A. Alice agrees to the session by sending back message 3. When Bob sees R_B encrypted with the session key he just generated, he knows Alice got message 2 and verified R_A.

What can Trudy do to try to subvert this protocol? She can fabricate message 1 and trick Bob into probing Alice, but Alice will see an R_A that she did not send and will not proceed further. Trudy cannot forge message 3 convincingly because she does not know R_B or K_S and cannot determine them without Alice's private key. She is out of luck.

However, the protocol does have a weakness: it assumes that Alice and Bob already know each other's public keys. Suppose that they do not. Alice could just send Bob her public key in the first message and ask Bob to send his back in the next one. The trouble with this approach is that it is subject to a bucket brigade attack. Trudy can capture Alice's message to Bob and send her own public key back to Alice. Alice will think she has a key for talking to Bob, when, in fact, she has a key for talking to Trudy. Now Trudy can read all the messages encrypted with what Alice thinks is Bob's public key.

The initial public-key exchange can be avoided by having all the public keys stored in a public database. Then Alice and Bob can fetch each other's public keys from the database. Unfortunately, Trudy can still pull off the bucket brigade attack by intercepting the requests to the database and sending simulated replies containing her own public key. After all, how do Alice and Bob know that the replies came from the real data base and not from Trudy?

Rivest and Shamir (1984) have devised a protocol that foils Trudy's bucket brigade attack. In their **interlock protocol**, after the public key exchange, Alice sends only half of her message to Bob, say, only the even bits (after encryption). Bob then responds with his even bits. After getting Bob's even bits, Alice sends her odd bits, then Bob does too.

The trick here is that when Trudy gets Alice's even bits, she cannot decrypt the message, even though Trudy has the private key. Consequently, she is unable to reencrypt the even bits using Bob's public key. If she sends junk to Bob, the protocol will continue, but Bob will shortly discover that the fully assembled message makes no sense and realized that he has been spoofed.

7.1.6. Digital Signatures

The authenticity of many legal, financial, and other documents is determined by the presence or absence of an authorized handwritten signature. And photocopies do not count. For computerized message systems to replace the physical transport of paper and ink documents, a solution must be found to these problems.

The problem of devising a replacement for handwritten signatures is a difficult one. Basically, what is needed is a system by which one party can send a "signed" message to another party in such a way that

1. The receiver can verify the claimed identity of the sender.
2. The sender cannot later repudiate the contents of the message.
3. The receiver cannot possibly have concocted the message himself.

The first requirement is needed, for example, in financial systems. When a customer's computer orders a bank's computer to buy a ton of gold, the bank's computer needs to be able to make sure that the computer giving the order really belongs to the company whose account is to be debited.

The second requirement is needed to protect the bank against fraud. Suppose that the bank buys the ton of gold, and immediately thereafter the price of gold drops sharply. A dishonest customer might sue the bank, claiming that he never issued any order to buy gold. When the bank produces the message in court, the customer denies having sent it.

The third requirement is needed to protect the customer in the event that the price of gold shoots up and the bank tries to construct a signed message in which the customer asked for one bar of gold instead of one ton.

Secret-Key Signatures

One approach to digital signatures is to have a central authority that knows everything and whom everyone trusts, say Big Brother (*BB*). Each user then chooses a secret key and carries it by hand to *BB*'s office. Thus only Alice and *BB* know Alice's secret, K_A, and so on.

When Alice wants to send a signed plaintext message, *P*, to her banker, Bob, she generates $K_A(B, R_A, t, P)$ and sends it as depicted in Fig. 7-22. *BB* sees that the message is from Alice, decrypts it, and sends a message to Bob as shown. The message to Bob contains the plaintext of Alice's message and also the signed message $K_{BB}(A, t, P)$, where *t* is a timestamp. Bob now carries out Alice's request.

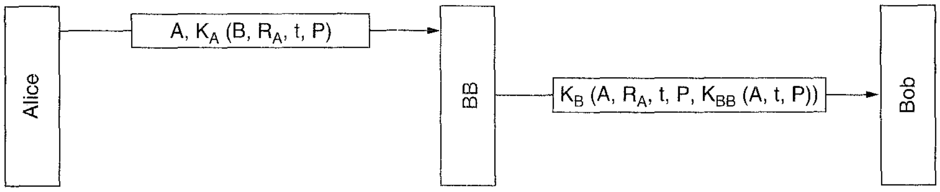

Alice
A, K_A (B, R_A, t, P)
BB
K_B (A, R_A, t, P, K_{BB} (A, t, P))
Bob

Fig. 7-22. Digital signatures with Big Brother.

What happens if Alice later denies sending the message? Step 1 is that everyone sues everyone (at least, in the United States). Finally, when the case comes to court and Alice vigorously denies sending Bob the disputed message, the judge will ask Bob how he can be sure that the disputed message came from Alice and not from Trudy. Bob first points out that *BB* will not accept a message from Alice unless it is encrypted with K_A, so there is no possibility of Trudy sending *BB* a false message from Alice.

Bob then dramatically produces Exhibit A, $K_{BB}(A, t, P)$. Bob says that this is a message signed by *BB* which proves Alice sent *P* to Bob. The judge then asks

BB (whom everyone trusts) to decrypt Exhibit A. When *BB* testifies that Bob is telling the truth, the judge decides in favor of Bob. Case dismissed.

One potential problem with the signature protocol of Fig. 7-22 is Trudy replaying either message. To minimize this problem, timestamps are used throughout. Furthermore, Bob can check all recent messages to see if R_A was used in any of them. If so, the message is discarded as a replay. Note that Bob will reject very old messages based on the timestamp. To guard against instant replay attacks, Bob just checks the R_A of every incoming message to see if such a message has been received from Alice in the past hour. If not, Bob can safely assume this is a new request.

Public-Key Signatures

A structural problem with using secret-key cryptography for digital signatures is that everyone has to agree to trust Big Brother. Furthermore, Big Brother gets to read all signed messages. The most logical candidates for running the Big Brother server are the government, the banks, or the lawyers. These organizations do not inspire total confidence in all citizens. Hence, it would be nice if signing documents did not require a trusted authority.

Fortunately, public-key cryptography can make an important contribution here. Let us assume that the public-key encryption and decryption algorithms have the property that $E(D(P)) = P$ in addition to the usual property that $D(E(P)) = P$. (RSA has this property, so the assumption is not unreasonable.) Assuming that this is the case, Alice can send a signed plaintext message, P, to Bob by transmitting $E_B(D_A(P))$. Note carefully that Alice knows her own (private) decryption key, D_A, as well as Bob's public key, E_B, so constructing this message is something Alice can do.

When Bob receives the message, he transforms it using his private key, as usual, yielding $D_A(P)$, as shown in Fig. 7-23. He stores this text in a safe place and then decrypts it using E_A to get the original plaintext.

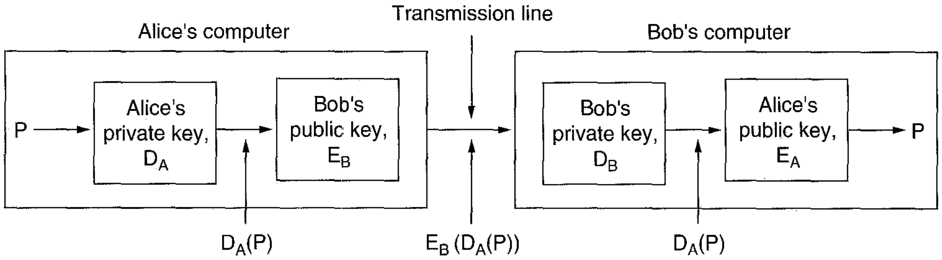


Fig. 7-23. Digital signatures using public-key cryptography.

To see how the signature property works, suppose that Alice subsequently denies having sent the message P to Bob. When the case comes up in court, Bob can produce both P and $D_A(P)$. The judge can easily verify that Bob indeed has a valid message encrypted by D_A by simply applying E_A to it. Since Bob does not know what Alice's private key is, the only way Bob could have acquired a message encrypted by it is if Alice did indeed send it. While in jail for perjury and fraud, Alice will have plenty of time to devise interesting new public-key algorithms.

Although using public-key cryptography for digital signatures is an elegant scheme, there are problems that are related to the environment in which they operate rather than with the basic algorithm. For one thing, Bob can prove that a message was sent by Alice only as long as D_A remains secret. If Alice discloses her secret key, the argument no longer holds, because anyone could have sent the message, including Bob himself.

The problem might arise, for example, if Bob is Alice's stockbroker. Alice tells Bob to buy a certain stock or bond. Immediately thereafter, the price drops sharply. To repudiate her message to Bob, Alice runs to the police claiming that her home was burglarized and her key was stolen. Depending on the laws in her state or country, she may or may not be legally liable, especially if she claims not to have discovered the break-in until getting home from work, several hours later.

Another problem with the signature scheme is what happens if Alice decides to change her key. Doing so is clearly legal, and it is probably a good idea to do so periodically. If a court case later arises, as described above, the judge will apply the *current* E_A to $D_A(P)$ and discover that it does not produce P. Bob will look pretty stupid at this point. Consequently, it appears that some authority is probably needed to record all key changes and their dates.

In principle, any public-key algorithm can be used for digital signatures. The de facto industry standard is the RSA algorithm. Many security products use it. However, in 1991, NIST (National Institute of Standards and Technology) proposed using a variant of the El Gamal public-key algorithm for their new **Digital Signature Standard (DSS)**. El Gamal gets its security from the difficulty of computing discrete logarithms, rather than the difficulty of factoring large numbers.

As usual when the government tries to dictate cryptographic standards, there was an uproar. DSS was criticized for being

1. Too secret (NSA designed the protocol for using El Gamal).
2. Too new (El Gamal has not yet been thoroughly analyzed).
3. Too slow (10 to 40 times slower than RSA for checking signatures).
4. Too insecure (fixed 512-bit key).

In a subsequent revision, the fourth point was rendered moot when keys up to 1024 bits were allowed. It is not yet clear whether DSS will catch on. For more details, see (Kaufman et al., 1995; Schneier, 1996; and Stinson, 1995).

Message Digests

One criticism of signature methods is that they often couple two distinct functions: authentication and secrecy. Often, authentication is needed but secrecy is not. Since cryptography is slow, it is frequently desirable to be able to send signed plaintext documents. Below we will describe an authentication scheme that does not require encrypting the entire message (De Jonge and Chaum, 1987).

This scheme is based on the idea of a one-way hash function that takes an arbitrarily long piece of plaintext and from it computes a fixed-length bit string. This hash function, often called a **message digest**, has three important properties:

1. Given P, it is easy to compute $MD(P)$.
2. Given $MD(P)$, it is effectively impossible to find P.
3. No one can generate two messages that have the same message digest.

To meet criterion 3, the hash should be at least 128 bits long, preferably more.

Computing a message digest from a piece of plaintext is much faster than encrypting that plaintext with a public-key algorithm, so message digests can be used to speed up digital signature algorithms. To see how this works, consider the signature protocol of Fig. 7-22 again. Instead of signing P with $K_{BB}(A, t, P)$, BB now computes the message digest by applying MD to P, yielding $MD(P)$. BB then encloses $K_{BB}(A, t, MD(P))$ as the fifth item in the list encrypted with K_B that is sent to Bob, instead of $K_{BB}(A, t, P)$.

If a dispute arises, Bob can produce both P and $K_{BB}(A, t, MD(P))$. After Big Brother has decrypted it for the judge, Bob has $MD(P)$, which is guaranteed to be genuine, and the alleged P. However, since it is effectively impossible for Bob to find any other message that gives this hash, the judge will easily be convinced that Bob is telling the truth. Using message digests in this way saves both encryption time and message transport and storage costs.

Message digests work in public-key cryptosystems, too, as shown in Fig. 7-24. Here, Alice first computes the message digest of her plaintext. She then signs the message digest and sends both the signed digest and the plaintext to Bob. If Trudy replaces P underway, Bob will see this when he computes $MD(P)$ himself.

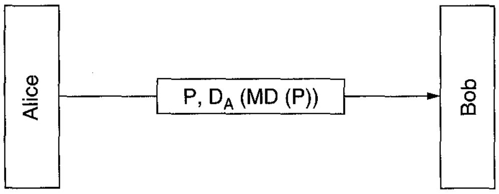


Fig. 7-24. Digital signatures using message digests.

A variety of message digest functions have been proposed. The most widely used ones are MD5 (Rivest, 1992) and SHA (NIST, 1993). **MD5** is the fifth in a series of hash functions designed by Ron Rivest. It operates by mangling bits in a sufficiently complicated way that every output bit is affected by every input bit. Very briefly, it starts out by padding the message to a length of 448 bits (modulo 512). Then the original length of the message is appended as a 64-bit integer to give a total input whose length is a multiple of 512 bits. The last precomputation step is initializing a 128-bit buffer to a fixed value.

Now the computation starts. Each round takes a 512-bit block of input and mixes it thoroughly with the 128-bit buffer. For good measure, a table constructed from the sine function is also thrown in. The point of using a known function like the sine is not because it is more random than a random number generator, but to avoid any suspicion that the designer built in a clever trapdoor through which only he can enter. IBM's refusal to disclose the principles behind the design of the S-boxes in DES led to a great deal of speculation about trapdoors. Four rounds are performed per input block. This process continues until all the input blocks have been consumed. The contents of the 128-bit buffer form the message digest. The algorithm has been optimized for software implementation on 32-bit machines. As a consequence, it may not be fast enough for future high-speed networks (Touch, 1995).

The other major message digest function is **SHA** (**Secure Hash Algorithm**), developed by NSA and blessed by NIST. Like MD5, it processes input data in 512-bit blocks, only unlike MD5, it generates a 160-bit message digest. It starts out by padding the message, then adding a 64-bit length to get a multiple of 512 bits. Then it initializes its 160-bit output buffer.

For each input block, the output buffer is updated using the 512-bit input block. No table of random numbers (or sine function values) is used, but for each block 80 rounds are computed, resulting in a thorough mixing. Each group of 20 rounds uses different mixing functions.

Since SHA's hash code is 32 bits longer than MD5's, all other things being equal, it is a factor of 2^{32} more secure than MD5. However, it is also slower than MD5, and having a hash code that is not a power of two might sometimes be an inconvenience. Otherwise, the two are roughly similar technically. Politically, MD5 is defined in an RFC and used heavily on the Internet. SHA is a government standard, and used by companies that have to use it because the government tells them to, or by those that want the extra security. A revised version, SHA-1, has been approved as a standard by NIST.

The Birthday Attack

In the world of crypto, nothing is ever what it seems to be. One might think that it would take on the order of 2^m operations to subvert an m-bit message digest. In fact, $2^{m/2}$ operations will often do using the **birthday attack,** an approach published by Yuval (1979) in his now-classic paper "How to Swindle Rabin."

The idea for this attack comes from a technique that math professors often use in their probability courses. The question is: How many students do you need in a class before the probability of having two people with the same birthday exceeds 1/2? Most students expect the answer to be way over 100. In fact, probability theory says it is just 23. Without giving a rigorous analysis, intuitively, with 23 people, we can form $(23 \times 22)/2 = 253$ different pairs, each of which has a probability of 1/365 of being a hit. In this light, it is not really so surprising any more.

More generally, if there is some mapping between inputs and outputs with n inputs (people, messages, etc.) and k possible outputs (birthdays, message digests, etc.), there are $n(n-1)/2$ input pairs. If $n(n-1)/2 > k$, the chance of having at least one match is pretty good. Thus, approximately, a match is likely for $n > \sqrt{k}$. This result means that a 64-bit message digest can probably be broken by generating about 2^{32} messages and looking for two with the same message digest.

Let us look at a practical example. The Dept. of Computer Science at State University has one position for a tenured faculty member and two candidates, Tom and Dick. Tom was hired two years before Dick, so he goes up for review first. If he gets it, Dick is out of luck. Tom knows that the department chairperson, Marilyn, thinks highly of his work, so he asks her to write him a letter of recommendation to the Dean, who will decide on Tom's case. Once sent, all letters become confidential.

Marilyn tells her secretary, Ellen, to write the Dean a letter, outlining what she wants in it. When it is ready, Marilyn will review it, compute and sign the 64-bit digest, and send it to the Dean. Ellen can send the letter later by email.

Unfortunately for Tom, Ellen is romantically involved with Dick and would like to do Tom in, so she writes the letter below with the 32 bracketed options.

Dear Dean Smith,

This [*letter* | *message*] is to give my [*honest* | *frank*] opinion of Prof. Tom Wilson, who is [*a candidate* | *up*] for tenure [*now* | *this year*]. I have [*known* | *worked with*] Prof. Wilson for [*about* | *almost*] six years. He is an [*outstanding* | *excellent*] researcher of great [*talent* | *ability*] known [*worldwide* | *internationally*] for his [*brilliant* | *creative*] insights into [*many* | *a wide variety of*] [*difficult* | *challenging*] problems.

He is also a [*highly* | *greatly*] [*respected* | *admired*] [*teacher* | *educator*]. His students give his [*classes* | *courses*] [*rave* | *spectacular*] reviews. He is [*our* | *the Department's*] [*most popular* | *best-loved*] [*teacher* | *instructor*].

[*In addition* | *Additionally*] Prof. Wilson is a [*gifted* | *effective*] fund raiser. His [*grants* | *contracts*] have brought a [*large* | *substantial*] amount of money into [*the* | *our*] Department. [*This money has* | *These funds have*] [*enabled* | *permitted*] us to [*pursue* | *carry out*] many [*special* | *important*] programs, [*such as* | *for example*] your State 2000 program. Without these funds we would [*be unable* | *not be able*] to continue this program, which is so [*important* | *essential*] to both of us. I strongly urge you to grant him tenure.

Unfortunately for Tom, as soon as Ellen finishes composing and typing in this letter, she also writes a second one:

Dear Dean Smith,

This [*letter* | *message*] is to give my [*honest* | *frank*] opinion of Prof. Tom Wilson, who is [*a candidate* | *up*] for tenure [*now* | *this year*]. I have [*known* | *worked with*] Tom for [*about* | *almost*] six years. He is a [*poor* | *weak*] researcher not well known in his [*field* | *area*]. His research [*hardly ever* | *rarely*] shows [*insight in* | *understanding of*] the [*key* | *major*] problems of [*the* | *our*] day.

Furthermore, he is not a [*respected* | *admired*] [*teacher* | *educator*]. His students give his [*classes* | *courses*] [*poor* | *bad*] reviews. He is [*our* | *the Department's*] least popular [*teacher* | *instructor*], known [*mostly* | *primarily*] within [*the* | *our*] Department for his [*tendency* | *propensity*] to [*ridicule* | *embarrass*] students [*foolish* | *imprudent*] enough to ask questions in his classes.

[*In addition* | *Additionally*] Tom is a [*poor* | *marginal*] fund raiser. His [*grants* | *contracts*] have brought only a [*meager* | *insignificant*] amount of money into [*the* | *our*] Department. Unless new [*money is* | *funds are*] quickly located, we may have to cancel some essential programs, such as your State 2000 program. Unfortunately, under these [*conditions* | *circumstances*] I cannot in good [*conscience* | *faith*] recommend him to you for [*tenure* | *a permanent position*].

Now Ellen sets up her computer to compute the 2^{32} message digests of each letter overnight. Chances are, one digest of the first letter will match one digest of the second letter. If not, she can add a few more options and try again during the weekend. Suppose that she finds a match. Call the "good" letter *A* and the "bad" one *B*.

Ellen now emails letter *A* to Marilyn for her approval. Marilyn, of course, approves, computes her 64-bit message digest, signs the digest, and emails the signed digest off to Dean Smith. Independently, Ellen emails letter *B* to the Dean.

After getting the letter and signed message digest, the Dean runs the message digest algorithm on letter *B*, sees that it agrees with what Marilyn sent him, and fires Tom. (Optional ending: Ellen tells Dick what she did. Dick is appalled and breaks off with her. Ellen is furious and confesses to Marilyn. Marilyn calls the Dean. Tom gets tenure after all.) With MD5 the birthday attack is infeasible because even at 1 billion digests per second, it would take over 500 years to compute all 2^{64} digests of two letters with 64 variants each, and even then a match is not guaranteed.

7.1.7. Social Issues

The implications of network security for individual privacy and society in general are staggering. Below we will just mention a few of the salient issues.

Governments do not like citizens keeping secrets from them. In some countries (e.g., France) all nongovernmental cryptography is simply forbidden unless the government is given all the keys being used. As Kahn (1980) and Selfridge and Schwartz (1980) point out, government eavesdropping has been practiced on a far more massive scale than most people could dream of, and governments want more than just a pile of indecipherable bits for their efforts.

The U.S. government has proposed an encryption scheme for future digital telephones that includes a special feature to allow the police to tap and decrypt all telephone calls made in the United States. The government promises not to use this feature without a court order, but many people still remember how former FBI Director J. Edgar Hoover illegally tapped the telephones of Martin Luther King, Jr. and other people in an attempt to neutralize them. The police say they need this power to catch criminals. The debate on both sides is vehement, to put it mildly. A discussion of the technology involved (Clipper) is given in (Kaufman et al., 1995). A way to circumvent this technology and send messages that the government cannot read is described in (Blaze, 1994; and Schneier, 1996). Position statements on all sides are given in (Hoffman, 1995).

The United States has a law (22 U.S.C. 2778) that prohibits citizens from exporting munitions (war materiel), such as tanks and jet fighters, without authorization from the DoD. For purposes of this law, cryptographic software is classified as a munition. Phil Zimmermann, who wrote PGP (Pretty Good Privacy), an email protection program, has been accused of violating this law, even though the government admits that he did not export it (but he did give it to a friend who put it on the Internet where foreigners could obtain it). Many people regarded this widely-publicized incident as a gross violation of the rights of an American citizen working to enhance people's privacy.

Not being an American does not help. On July 9, 1986, three Israeli researchers working at the Weizmann Institute in Israel filed a U.S. patent application for a new digital signature scheme that they had invented. They spent the next 6 months discussing their research at conferences all over the world. On Jan. 6, 1987, the U.S. patent office told them to notify all Americans who knew about their results that disclosure of the research would subject them to two years in prison, a 10,000-dollar fine, or both. The patent office also wanted a list of all foreign nationals who knew about the research. To find out how this story turned out, see (Landau, 1988).

Patents are another hot topic. Nearly all public-key algorithms are patented. Patent protection lasts for 17 years. The RSA patent, for example, expires on Sept. 20, 2000.

Network security is politicized to an extent few other technical issues are, and rightly so, since it relates to the difference between a democracy and a police state in the digital era. The March 1993 and November 1994 issues of *Communications of the ACM* have long sections on telephone and network security, respectively, with vigorous arguments explaining and defending many points of view. Chapter 25 of Schneier's security book deals with the politics of cryptography

(Schneier, 1996). Chapter 8 of his email book does too (Schneier, 1995). Privacy and computers are also discussed in (Adam, 1995). These references are highly recommended for readers who wish to pursue their study of this subject.

7.2. DNS—Domain Name System

Programs rarely refer to hosts, mailboxes, and other resources by their binary network addresses. Instead of binary numbers, they use ASCII strings, such as *tana@art.ucsb.edu*. Nevertheless, the network itself only understands binary addresses, so some mechanism is required to convert the ASCII strings to network addresses. In the following sections we will study how this mapping is accomplished in the Internet.

Way back in the ARPANET, there was simply a file, *hosts.txt*, that listed all the hosts and their IP addresses. Every night, all the hosts would fetch it from the site at which it was maintained. For a network of a few hundred large timesharing machines, this approach worked reasonably well.

However, when thousands of workstations were connected to the net, everyone realized that this approach could not continue to work forever. For one thing, the size of the file would become too large. However, even more important, host name conflicts would occur constantly unless names were centrally managed, something unthinkable in a huge international network. To solve these problems, **DNS** (the **Domain Name System**) was invented.

The essence of DNS is the invention of a hierarchical, domain-based naming scheme and a distributed database system for implementing this naming scheme. It is primarily used for mapping host names and email destinations to IP addresses but can also be used for other purposes. DNS is defined in RFCs 1034 and 1035.

Very briefly, the way DNS is used is as follows. To map a name onto an IP address, an application program calls a library procedure called the **resolver**, passing it the name as a parameter. The resolver sends a UDP packet to a local DNS server, which then looks up the name and returns the IP address to the resolver, which then returns it to the caller. Armed with the IP address, the program can then establish a TCP connection with the destination, or send it UDP packets.

7.2.1. The DNS Name Space

Managing a large and constantly changing set of names is a nontrivial problem. In the postal system, name management is done by requiring letters to specify (implicitly or explicitly) the country, state or province, city, and street address of the addressee. By using this kind of hierarchical addressing, there is no confusion between the Marvin Anderson on Main St. in White Plains, N.Y. and the Marvin Anderson on Main St. in Austin, Texas. DNS works the same way.

Conceptually, the Internet is divided into several hundred top-level **domains**, where each domain covers many hosts. Each domain is partitioned into subdomains, and these are further partitioned, and so on. All these domains can be represented by a tree, as shown in Fig. 7-25. The leaves of the tree represent domains that have no subdomains (but do contain machines, of course) A leaf domain may contain a single host, or it may represent a company and contains thousands of hosts.

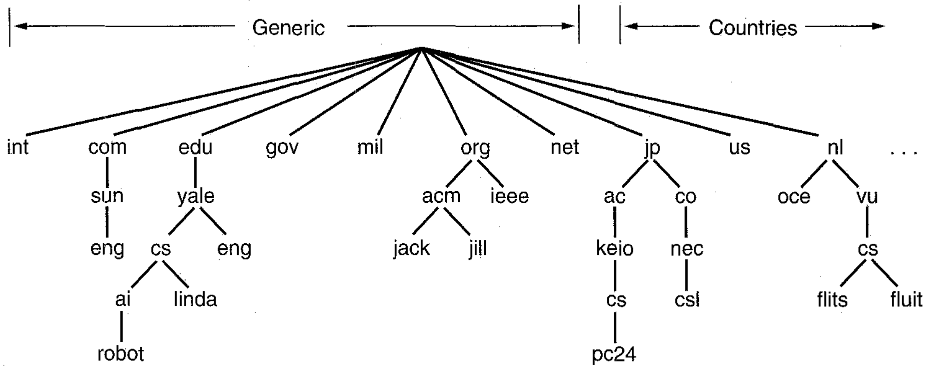


Fig. 7-25. A portion of the Internet domain name space.

The top-level domains come in two flavors: generic and countries. The generic domains are *com (commercial), edu* (educational institutions), *gov* (the U.S. federal government), *int* (certain international organizations), *mil* (the U.S. armed forces), *net* (network providers), and *org* (nonprofit organizations). The country domains include one entry for every country, as defined in ISO 3166.

Each domain is named by the path upward from it to the (unnamed) root. The components are separated by periods (pronounced "dot"). Thus Sun Microsystems engineering department might be *eng.sun.com.*, rather than a UNIX-style name such as */com/sun/eng*. Notice that this hierarchical naming means that *eng.sun.com.* does not conflict with a potential use of *eng* in *eng.yale.edu.*, which might be used by the Yale English department.

Domain names can be either absolute or relative. An absolute domain name ends with a period (e.g., *eng.sun.com.*), whereas a relative one does not. Relative names have to be interpreted in some context to uniquely determine their true meaning. In both cases, a named domain refers to a specific node in the tree and all the nodes under it.

Domain names are case insensitive, so *edu* and *EDU* mean the same thing. Component names can be up to 63 characters long, and full path names must not exceed 255 characters.

In principle, domains can be inserted into the tree in two different ways. For example, *cs.yale.edu* could equally well be listed under the *us* country domain as *cs.yale.ct.us*. In practice, however, nearly all organizations in the United States are under a generic domain, and nearly all outside the United States are under the domain of their country. There is no rule against registering under two top-level domains, but doing so might be confusing, so few organizations do it.

Each domain controls how it allocates the domains under it. For example, Japan has domains *ac.jp* and *co.jp* that mirror *edu* and *com*. The Netherlands does not make this distinction and puts all organizations directly under *nl*. Thus all three of the following are university computer science departments:

1. *cs.yale.edu* (Yale University, in the United States)
2. *cs.vu.nl* (Vrije Universiteit, in The Netherlands)
3. *cs.keio.ac.jp* (Keio University, in Japan)

To create a new domain, permission is required of the domain in which it will be included. For example, if a VLSI group is started at Yale and wants to be known as *vlsi.cs.yale.edu*, it needs permission from whomever manages *cs.yale.edu*. Similarly, if a new university is chartered, say, the University of Northern South Dakota, it must ask the manager of the *edu* domain to assign it *unsd.edu*. In this way, name conflicts are avoided and each domain can keep track of all its subdomains. Once a new domain has been created and registered, it can create subdomains, such as *cs.unsd.edu*, without getting permission from anybody higher up the tree.

Naming follows organizational boundaries, not physical networks. For example, if the computer science and electrical engineering departments are located in the same building and share the same LAN, they can nevertheless have distinct domains. Similarly, even if computer science is split over Babbage Hall and Turing Hall, all the hosts in both buildings will normally belong to the same domain.

7.2.2. Resource Records

Every domain, whether it is a single host or a top-level domain, can have a set of **resource records** associated with it. For a single host, the most common resource record is just its IP address, but many other kinds of resource records also exist. When a resolver gives a domain name to DNS, what it gets back are the resource records associated with that name. Thus the real function of DNS is to map domain names onto resource records.

A resource record is a five-tuple. Although they are encoded in binary for efficiency, in most expositions resource records are presented as ASCII text, one line per resource record. The format we will use is as follows:

```
Domain_name   Time_to_live   Type   Class   Value
```

The *Domain_name* tells the domain to which this record applies. Normally, many records exist for each domain and each copy of the database holds information about multiple domains. This field is thus the primary search key used to satisfy queries. The order of the records in the database is not significant. When a query is made about a domain, all the matching records of the class requested are returned.

The *Time_to_live* field gives an indication of how stable the record is. Information that is highly stable is assigned a large value, such as 86400 (the number of seconds in 1 day). Information that is highly volatile is assigned a small value, such as 60 (1 minute). We will come back to this point later when we have discussed caching.

The *Type* field tells what kind of record this is. The most important types are listed in Fig. 7-26.

Type	Meaning	Value
SOA	Start of Authority	Parameters for this zone
A	IP address of a host	32-Bit integer
MX	Mail exchange	Priority, domain willing to accept email
NS	Name Server	Name of a server for this domain
CNAME	Canonical name	Domain name
PTR	Pointer	Alias for an IP address
HINFO	Host description	CPU and OS in ASCII
TXT	Text	Uninterpreted ASCII text

Fig. 7-26. The principal DNS resource record types.

An *SOA* record provides the name of the primary source of information about the name server's zone (described below), the email address of its administrator, a unique serial number, and various flags and timeouts.

The most important record type is the *A* (Address) record. It holds a 32-bit IP address for some host. Every Internet host must have at least one IP address, so other machines can communicate with it. Some hosts have two or more network connections, in which case they will have one type *A* resource record per network connection (and thus per IP address).

The next most important record type is the *MX* record. It specifies the name of the domain prepared to accept email for the specified domain. A common use of this record is to allow a machine that is not on the Internet to receive email from Internet sites. Delivery is accomplished by having the non-Internet site make an arrangement with some Internet site to accept email for it and forward it using whatever protocol the two of them agree on.

For example, suppose that Cathy is a computer science graduate student at UCLA. After she gets her degree in AI, she sets up a company, Electrobrain

Corporation, to commercialize her ideas. She cannot afford an Internet connection yet, so she makes an arrangement with UCLA to allow her to have her email sent there. A few times a day she will call up and collect it.

Next, she registers her company with the *com* domain and is assigned the domain *electrobrain.com*. She might then ask the administrator of the *com* domain to add an *MX* record to the *com* database as follows:

```
electrobrain.com   86400   IN   MX   1   mailserver.cs.ucla.edu
```

In this way, mail will be forwarded to UCLA where she can pick it up by logging in. Alternatively, UCLA could call her and transfer the email by any protocol they mutually agree on.

The *NS* records specify name servers. For example, every DNS database normally has an *NS* record for each of the top-level domains, so email can be sent to distant parts of the naming tree. We will come back to this point later.

CNAME records allow aliases to be created. For example, a person familiar with Internet naming in general wanting to send a message to someone whose login name is *paul* in the computer science department at M.I.T. might guess that *paul@cs.mit.edu* will work. Actually this address will not work, because the domain for M.I.T.'s computer science department is *lcs.mit.edu*. However, as a service to people who do not know this, M.I.T. could create a *CNAME* entry to point people and programs in the right direction. An entry like this one might do the job:

```
cs.mit.edu   86400   IN   CNAME   lcs.mit.edu
```

Like *CNAME*, *PTR* points to another name. However, unlike *CNAME*, which is really just a macro definition, *PTR* is a regular DNS datatype whose interpretation depends on the context in which it is found. In practice, it is nearly always used to associate a name with an IP address to allow lookups of the IP address and return the name of the corresponding machine.

HINFO records allow people to find out what kind of machine and operating system a domain corresponds to. Finally, *TXT* records allow domains to identify themselves in arbitrary ways. Both of these record types are for user convenience. Neither is required, so programs cannot count on getting them (and probably cannot deal with them if they do get them).

Getting back to the general structure of resource records, the fourth field of every resource record is the *Class*. For Internet information, it is always *IN*. For non-Internet information, other codes can be used.

Finally, we come to the *Value* field. This field can be a number, a domain name, or an ASCII string. The semantics depend on the record type. A short description of the *Value* fields for each of the principal records types is given in Fig. 7-26.

As an example of the kind of information one might find in the DNS database of a domain, see Fig. 7-27. This figure depicts part of a (semihypothetical) database for the *cs.vu.nl* domain shown in Fig. 7-25. The database contains seven types of resource records.

```
; Authoritative data for cs.vu.nl
cs.vu.nl.         86400  IN  SOA     star boss (952771,7200,7200,2419200,86400)
cs.vu.nl.         86400  IN  TXT     "Faculteit Wiskunde en Informatica."
cs.vu.nl.         86400  IN  TXT     "Vrije Universiteit Amsterdam."
cs.vu.nl.         86400  IN  MX      1 zephyr.cs.vu.nl.
cs.vu.nl.         86400  IN  MX      2 top.cs.vu.nl.

flits.cs.vu.nl.   86400  IN  HINFO   Sun Unix
flits.cs.vu.nl.   86400  IN  A       130.37.16.112
flits.cs.vu.nl.   86400  IN  A       192.31.231.165
flits.cs.vu.nl.   86400  IN  MX      1 flits.cs.vu.nl.
flits.cs.vu.nl.   86400  IN  MX      2 zephyr.cs.vu.nl.
flits.cs.vu.nl.   86400  IN  MX      3 top.cs.vu.nl.
www.cs.vu.nl.     86400  IN  CNAME   star.cs.vu.nl
ftp.cs.vu.nl.     86400  IN  CNAME   zephyr.cs.vu.nl

rowboat                  IN  A       130.37.56.201
                         IN  MX      1 rowboat
                         IN  MX      2 zephyr
                         IN  HINFO   Sun Unix

little-sister            IN  A       130.37.62.23
                         IN  HINFO   Mac MacOS

laserjet                 IN  A       192.31.231.216
                         IN  HINFO   "HP Laserjet IIISi" Proprietary
```

Fig. 7-27. A portion of a possible DNS database for *cs.vu.nl*

The first noncomment line of Fig. 7-27 gives some basic information about the domain, which will not concern us further. The next two lines give textual information about where the domain is located. Then come two entries giving the first and second places to try to deliver email sent to *person@cs.vu.nl*. The *zephyr* (a specific machine) should be tried first. If that fails, the *top* should be tried next.

After the blank line, added for readability, come lines telling that the *flits* is a Sun workstation running UNIX and giving both of its IP addresses. Then three choices are given for handling email sent to *flits.cs.vu.nl*. First choice is naturally the *flits* itself, but if it is down, the *zephyr* and *top* are the second and third choices. Next comes an alias, *www.cs.vu.nl*, so that this address can be used without designating a specific machine. Creating this alias allows *cs.vu.nl* to change its World Wide Web server without invalidating the address people use to get to it. A similar argument holds for *ftp.cs.vu.nl*.

The next four lines contain a typical entry for a workstation, in this case, *rowboat.cs.vu.nl*. The information provided contains the IP address, the primary and secondary mail drops, and information about the machine. Then comes an entry for a non-UNIX system that is not capable of receiving mail itself, followed by an entry for a laser printer.

What is not shown (and is not in this file), are the IP addresses to use to look up the top level domains. These are needed to look up distant hosts, but since they are not part of the *cs.vu.nl* domain, they are not in this file. They are supplied by the root servers, whose IP addresses are present in a system configuration file and loaded into the DNS cache when the DNS server is booted. They have very long timeouts, so once loaded, they are never purged from the cache.

7.2.3. Name Servers

In theory at least, a single name server could contain the entire DNS database and respond to all queries about it. In practice, this server would be so overloaded as to be useless. Furthermore, if it ever went down, the entire Internet would be crippled.

To avoid the problems associated with having only a single source of information, the DNS name space is divided up into nonoverlapping **zones**. One possible way to divide up the name space of Fig. 7-25 is shown in Fig. 7-28. Each zone contains some part of the tree and also contains name servers holding the authoritative information about that zone. Normally, a zone will have one primary name server, which gets its information from a file on its disk, and one or more secondary name servers, which get their information from the primary name server. To improve reliability, some servers for a zone can be located outside the zone.

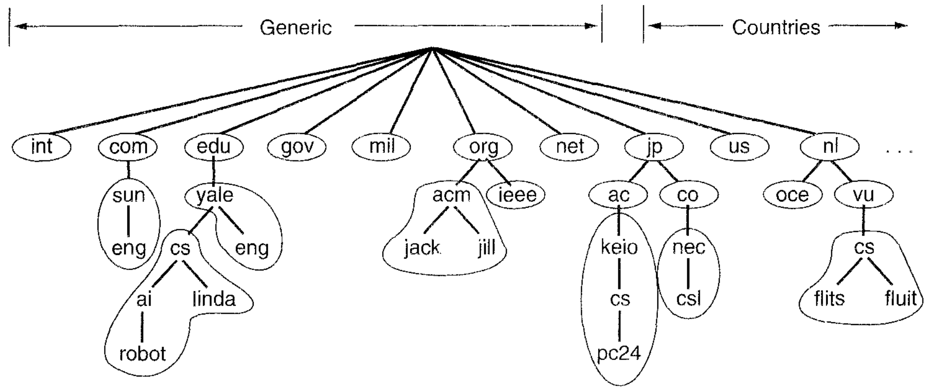


Fig. 7-28. Part of the DNS name space showing the division into zones.

Where the zone boundaries are placed within a zone is up to that zone's administrator. This decision is made in large part based on how many name servers are desired, and where. For example, in Fig. 7-28, Yale has a server for *yale.edu* that handles *eng.yale.edu* but not *cs.yale.edu*, which is a separate zone with its own name servers. Such a decision might be made when a department such as English does not wish to run its own name server, but a department such as computer science does. Consequently, *cs.yale.edu* is a separate zone but *eng.yale.edu* is not.

When a resolver has a query about a domain name, it passes the query to one of the local name servers. If the domain being sought falls under the jurisdiction of the name server, such as *ai.cs.yale.edu* falling under *cs.yale.edu*, it returns the authoritative resource records. An **authoritative record** is one that comes from the authority that manages the record, and is thus always correct. Authoritative records are in contrast to cached records, which may be out of date.

If, however, the domain is remote and no information about the requested domain is available locally, the name server sends a query message to the top-level name server for the domain requested. To make this process clearer, consider the example of Fig. 7-29. Here, a resolver on *flits.cs.vu.nl* wants to know the IP address of the host *linda.cs.yale.edu*. In step 1, it sends a query to the local name server, *cs.vu.nl*. This query contains the domain name sought, the type (*A*) and the class (*IN*).

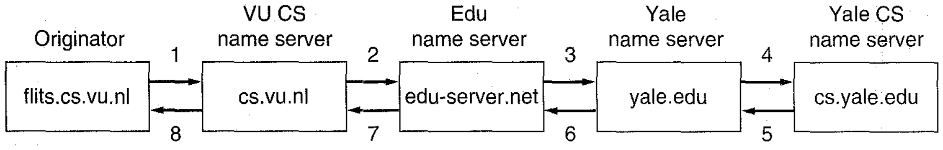


Fig. 7-29. How a resolver looks up a remote name in eight steps.

Let us suppose the local name server has never had a query for this domain before and knows nothing about it. It may ask a few other nearby name servers, but if none of them know, it sends a UDP packet to the server for *edu* given in its database (see Fig. 7-29), *edu-server.net*. It is unlikely that this server knows the address of *linda.cs.yale.edu*, and probably does not know *cs.yale.edu* either, but it must know all of its own children, so it forwards the request to the name server for *yale.edu* (step 3). In turn, this one forwards the request to *cs.yale.edu* (step 4), which must have the authoritative resource records. Since each request is from a client to a server, the resource record requested works its way back in steps 5 through 8.

Once these records get back to the *cs.vu.nl* name server, they will be entered into a cache there, in case they are needed later. However, this information is not authoritative, since changes made at *cs.yale.edu* will not be propagated to all the caches in the world that may know about it. For this reason, cache entries should not live too long. This is the reason that the *Time_to_live* field is included in each resource record. It tells remote name servers how long to cache records. If a certain machine has had the same IP address for years, it may be safe to cache that information for 1 day. For more volatile information, it might be safer to purge the records after a few seconds or a minute.

It is worth mentioning that the query method described here is known as a **recursive query**, since each server that does not have the requested information goes and finds it somewhere, then reports back. An alternative form is also possible. In this form, when a query cannot be satisfied locally, the query fails, but the name of the next server along the line to try is returned. This procedure gives the client more control over the search process. Some servers do not implement recursive queries and always return the name of the next server to try.

It is also worth pointing out that when a DNS client fails to get a response before its timer goes off, it normally will try another server next time. The assumption here is that the server is probably down, rather than the request or reply got lost.

7.3. SNMP—SIMPLE NETWORK MANAGEMENT PROTOCOL

In the early days of the ARPANET, if the delay to some host became unexpectedly large, the person detecting the problem would just run the Ping program to bounce a packet off the destination. By looking at the timestamps in the header of the packet returned, the location of the problem could usually be pinpointed and some appropriate action taken. In addition, the number of routers was so small, that it was feasible to ping each one to see if it was sick.

When the ARPANET turned into the worldwide Internet, with multiple backbones and multiple operators, this solution ceased to be adequate, so better tools for network management were needed. Two early attempts were defined in RFC 1028 and RFC 1067, but these were short lived. In May 1990, RFC 1157 was published, defining version 1 of **SNMP** (**Simple Network Management Protocol**). Along with a companion document (RFC 1155) on management information, SNMP provided a systematic way of monitoring and managing a computer network. This framework and protocol were widely implemented in commercial products and became the de facto standards for network management.

As experience was gained, shortcomings in SNMP came to light, so an enhanced version of SNMP (SNMPv2) was defined (in RFCs 1441 to 1452) and started along the road to become an Internet standard. In the sections to follow, we will give a brief discussion of the SNMP (meaning SNMPv2) model and protocol.

Although SNMP was designed with the idea of its being simple, at least one author has managed to produce a 600-page book on it (Stallings, 1993a). For more compact descriptions (450-550 pages), see the books by Rose (1994) and Rose and McCloghrie (1995), both of whom were among the designers of SNMP. Other references are (Feit, 1995; and Hein and Griffiths, 1995).

7.3.1. The SNMP Model

The SNMP model of a managed network consists of four components:

1. Managed nodes.
2. Management stations.
3. Management information.
4. A management protocol.

These pieces are illustrated in Fig. 7-30 and discussed below.

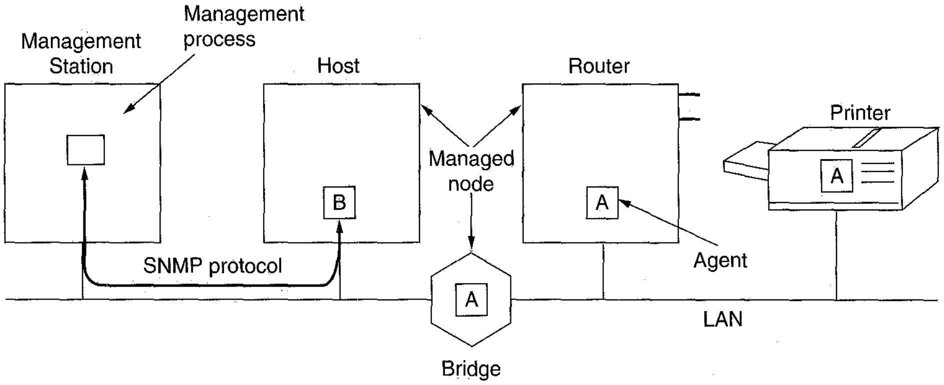


Fig. 7-30. Components of the SNMP management model.

The managed nodes can be hosts, routers, bridges, printers, or any other devices capable of communicating status information to the outside world. To be managed directly by SNMP, a node must be capable of running an SNMP management process, called an **SNMP agent**. All computers meet this requirement, as do increasingly many bridges, routers, and peripheral devices designed for network use. Each agent maintains a local database of variables that describe its state and history and affect its operation.

Network management is done from **management stations**, which are, in fact, general-purpose computers running special management software. The management stations contain one or more processes that communicate with the agents over the network, issuing commands and getting responses. In this design, all the intelligence is in the management stations, in order to keep the agents as simple as possible and minimize their impact on the devices they are running on. Many management stations have a graphical user interface to allow the network manager to inspect the status of the network and take action when required.

Most real networks are multivendor, with hosts from one or more manufacturers, bridges and routers from other companies, and printers from still other ones.

In order to allow a management station (potentially from yet another supplier) to talk to all these diverse components, the nature of the information maintained by all the devices must be rigidly specified. Having the management station ask a router what its packet loss rate is of no use if the router does not keep track of its loss rate. Therefore, SNMP describes (in excruciating detail) the exact information each kind of agent has to maintain and the format it has to supply it in. The largest portion of the SNMP model is the definition of who has to keep track of what and how this information is communicated.

Very briefly, each device maintains one or more variables that describe its state. In the SNMP literature, these variables are called **objects**, but the term is misleading because they are not objects in the sense of an object-oriented system because they just have state and no methods (other than reading and writing their values). Nevertheless, the term is so ingrained (e.g., used in various reserved words in the specification language used) that we will use it here. The collection of all possible objects in a network is given in a data structure called the **MIB** (**Management Information Base**).

The management station interacts with the agents using the SNMP protocol. This protocol allows the management station to query the state of an agent's local objects, and change them if necessary. Most of SNMP consists of this query-response type communication.

However, sometimes events happen that are not planned. Managed nodes can crash and reboot, lines can go down and come back up, congestion can occur, and so on. Each significant event is defined in a MIB module. When an agent notices that a significant event has occurred, it immediately reports the event to all management stations in its configuration list. This report is called an SNMP **trap** (for historical reasons). The report usually just states that some event has occurred. It is up to the management station to then issue queries to find out all the gory details. Because communication from managed nodes to the management station is not reliable (i.e., is not acknowledged), it is wise for the management station to poll each managed node occasionally anyway, checking for unusual events, just in case. The model of polling at long intervals with acceleration on receipt of a trap is called **trap directed polling**.

This model assumes that each managed node is capable of running an SNMP agent internally. Older devices or devices not originally intended for use on a network may not have this capability. To handle them, SNMP defines what is called a **proxy agent**, namely an agent that watches over one or more nonSNMP devices and communicates with the management station on their behalf, possibly communicating with the devices themselves using some nonstandard protocol.

Finally, security and authentication play a major role in SNMP. A management station has the capability of learning a great deal about every node under its control and also has the capability of shutting them all down. Hence it is of great importance that agents be convinced that queries allegedly coming from the management station, in fact, come from the management station. In SNMPv1, the management station proved who it was by putting a (plaintext) password in each message. In SNMPv2, security was improved considerably using modern cryptographic techniques of the type we have already studied. However, this addition made an already bulky protocol every bulkier, and it was later thrown out.

7.3.2. ASN.1—Abstract Syntax Notation 1

The heart of the SNMP model is the set of objects managed by the agents and read and written by the management station. To make multivendor communication possible, it is essential that these objects be defined in a standard and vendor-neutral way. Furthermore, a standard way is needed to encode them for transfer over a network. While definitions in C would satisfy the first requirement, such definitions do not define a bit encoding on the wire in such a way that a 32-bit two's complement little endian management station can exchange information unambiguously with an agent on a 16-bit one's complement big endian CPU.

For this reason, a standard object definition language, along with encoding rules, is needed. The one used by SNMP is taken from OSI and called **ASN.1** (**Abstract Syntax Notation One**). Like much of OSI, it is large, complex, and not especially efficient. (The author is tempted to say that by calling it ASN.1 instead of just ASN, the designers implicitly admitted that it would soon be replaced by ASN.2, but he will politely refrain from saying this.) The one alleged strength of ASN.1 (the existence of unambiguous bit encoding rules) is now really a weakness, because the encoding rules are optimized to minimize the number of bits on the wire, at the cost of wasting CPU time at both ends encoding and decoding them. A simpler scheme, using 32-bit integers aligned on 4-byte boundaries would probably have been better. Nevertheless, for better or worse, SNMP is drenched in ASN.1, (albeit a simplified subset of it), so anyone wishing to truly understand SNMP must become fluent in ASN.1. Hence the following explanation.

Let us start with the data description language, described in International Standard 8824. After that we will discuss the encoding rules, described in International Standard 8825. The ASN.1 abstract syntax is essentially a primitive data declaration language. It allows the user to define primitive objects and then combine them into more complex ones. A series of declarations in ASN.1 is functionally similar to the declarations found in the header files associated with many C programs.

SNMP has some lexical conventions that we will follow. These are not entirely the same as pure ASN.1 uses, however. Built-in data types are written in uppercase (e.g., *INTEGER*). User-defined types begin with an uppercase letter but must contain at least one character other than an uppercase letter. Identifiers may contain upper and lowercase letters, digits, and hyphens, but must begin with a lowercase letter (e.g., *counter*). White space (tabs, carriage returns, etc.) is not significant. Finally, comments start with -- and continue until the end of the line or the next occurrence of --.

The ASN.1 basic data types allowed in SNMP are shown in Fig. 7-31. (We will generally ignore features of ASN.1, such as *BOOLEAN* and *REAL* types, not permitted in SNMP.) The use of the codes will be described later.

Primitive type	Meaning	Code
INTEGER	Arbitrary length integer	2
BIT STRING	A string of 0 or more bits	3
OCTET STRING	A string of 0 of more unsigned bytes	4
NULL	A place holder	5
OBJECT IDENTIFIER	An officially defined data type	6

Fig. 7-31. The ASN.1 primitive data types permitted in SNMP.

A variable of type *INTEGER* may, in theory, take on any integral value, but other SNMP rules limit the range. As an example of how types are used, consider how a variable, *count*, of type *INTEGER* would be declared and (optionally) initialized to 100 in ASN.1:

```
count INTEGER ::= 100
```

Often a subtype whose variables are restricted to specific values or to a specific range is required. These can be declared as follows:

```
Status ::= INTEGER { up(1), down(2), unknown(3) }

PacketSize ::= INTEGER (0..1023)
```

Variables of type *BIT STRING* and *OCTET STRING* contain zero or more bits and bytes, respectively. A bit is either 0 or 1. A byte falls in the range 0 to 255, inclusive. For both types, a string length and an initial value may be given.

*OBJECT IDENTIFIER*s provide a way of identifying objects. In principle, every object defined in every official standard can be uniquely identified. The mechanism that is used is to define a standards tree, and place every object in every standard at a unique location in the tree. The portion of the tree that includes the SNMP MIB is shown in Fig. 7-32.

The top level of the tree lists all the important standards organizations in the world (in ISO's view), namely ISO and CCITT (now ITU), plus the combination of the two. From the *iso* node, four arcs are defined, one of which is for *identified-organization*, which is ISO's concession that maybe some other folks are vaguely involved with standards, too. The U.S. Dept. of Defense has been assigned a place in this subtree, and DoD has assigned the Internet number 1 in its hierarchy. Under the Internet hierarchy, the SNMP MIB has code 1.

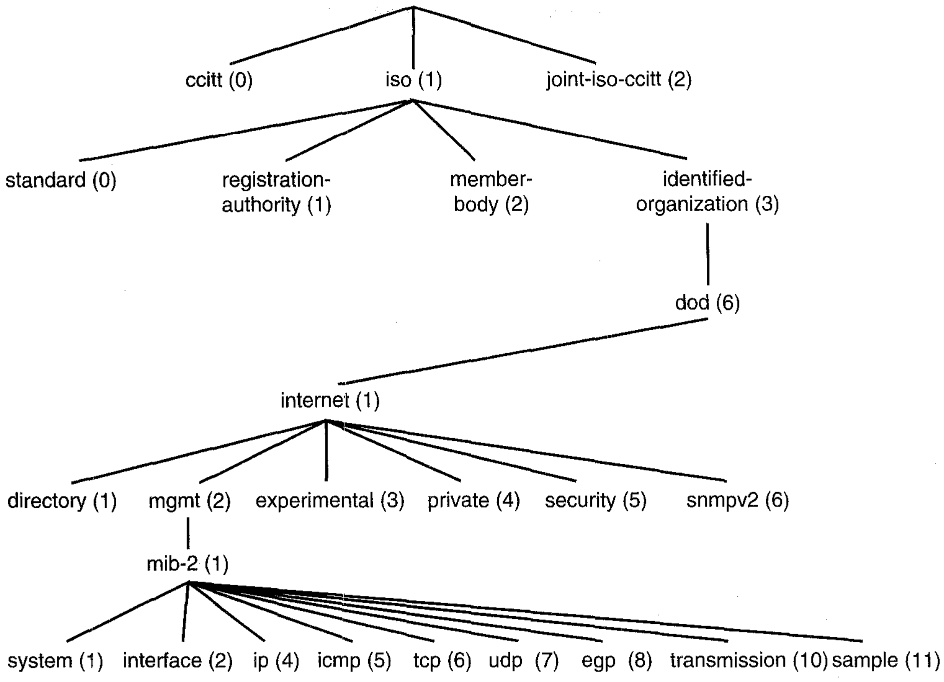


Fig. 7-32. Part of the ASN.1 object naming tree.

Every arc in Fig. 7-32 has both a label and a number, so nodes can be identified by a list of arcs, using label(number) or numbers. Thus all SNMP MIB objects are identified by a label of the form

{iso identified-organization(3) dod(6) internet(1) mgmt(2) mib-2(1) ...}

or alternatively {1 3 6 1 2 1 ...}. Mixed forms are also permitted. For example, the above identification can also be written as

{internet(1) 2 1 ...}

In this way, every object in every standard can be represented as an *OBJECT IDENTIFIER*.

ASN.1 defines five ways to construct new types from the basic ones. *SEQUENCE* is an ordered list of types, similar to a structure in C and a record in Pascal. *SEQUENCE OF* is a one-dimensional array of a single type. *SET* and *SET OF* are analogous, but unordered. *CHOICE* creates a union from a given list of types. The two set constructors are not used in any of the SNMP documents.

Another way to create new types is to tag old ones. Tagging a type is somewhat similar to the practice in C of defining new types, say *time_t* and *size_t*, both of which are longs, but which are used in different contexts. Tags come in four categories: universal, application-wide, context-specific and private. Each tag consists of a label and an integer identifying the tag. For example,

```
Counter32 ::= [APPLICATION 1] INTEGER (0..4294967295)

Gauge32   ::= [APPLICATION 2] INTEGER (0..4294967295)
```

define two different application-wide types, both of which are implemented by 32-bit unsigned integers, but which are conceptually different. The former might, for example, wrap around when it gets to the maximum value, whereas the latter might just continue to return the maximum value until its is decreased or reset.

A tagged type can have the keyword *IMPLICIT* after the closing square bracket when the type of what follows is obvious from the context (not true in a *CHOICE*, for example). Doing so allows a more efficient bit encoding since the tag does not have to be transmitted. In a type involving a *CHOICE* between two different types, a tag must be transmitted to tell the receiver which type is present.

ASN.1 defines a complex macro mechanism, which is heavily used in SNMP. A macro can be used as a kind of prototype to generate a set of new types and values, each with its own syntax. Each macro defines some (possibly optional) keywords, that are used in the call to identify which parameter is which (i.e., the macro parameters are identified by keyword, not by position). The details of how ASN.1 macros work is beyond the scope of this book. Suffice it to say that a macro is invoked by giving its name and then listing (some of) its keywords and their values for this invocation. Macros are expanded at compile time, not at run time. Some examples of macros will be cited below.

ASN.1 Transfer Syntax

An ASN.1 **transfer syntax** defines how values of ASN.1 types are unambiguously converted to a sequence of bytes for transmission (and unambiguously decoded at the other end). The transfer syntax used by ASN.1 is called **BER** (**Basic Encoding Rules**). ASN.1 has other transfer syntaxes that SNMP does not use. The rules are recursive, so the encoding of a structured object is just the concatenation of the encodings of the component objects. In this way, all object encodings can be reduced to a well-defined sequence of encoded primitive objects. The encoding of these objects, in turn, is defined by the BER.

The guiding principle behind the basic encoding rules is that every value transmitted, both primitive and constructed ones, consists of up to four fields:

1. The identifier (type or tag).
2. The length of the data field, in bytes.
3. The data field.
4. The end-of-contents flag, if the data length is unknown.

The last one is permitted by ASN.1, but specifically forbidden by SNMP, so we will assume the data length is always known.

The first field identifies the item that follows. It, itself, has three subfields, as shown in Fig. 7-33. The high-order 2 bits identify the tag type. The next bit tells whether the value is primitive (0) or not (1). The tag bits are 00, 01, 10, and 11, for *UNIVERSAL*, *APPLICATION*, context-specific, and *PRIVATE*, respectively. The remaining 5 bits can be used to encode the value of the tag if it is in the range 0 through 30. If the tag is 31 or more, the low-order 5 bits contain 11111, with the true value in the next byte or bytes.

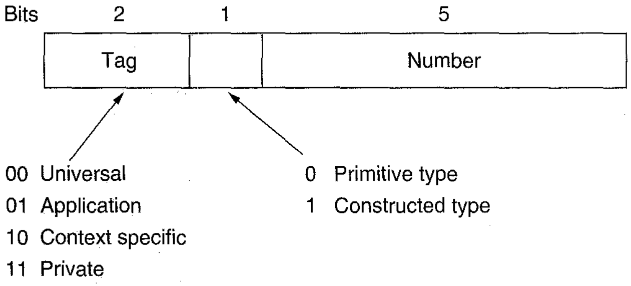


Fig. 7-33. The first byte of each data item sent in the ASN.1 transfer syntax.

The rule used to encode tags greater than 30 has been designed to handle arbitrarily large numbers. Each identifier byte following the first one contains 7 data bits. The high-order bit is set to 0 in all but the last one. Thus tag values up to $2^7 - 1$ can be handled in 2 bytes, and up to $2^{14} - 1$ can be handled in 3 bytes.

The encoding of the *UNIVERSAL* types is straightforward. Each primitive type has been assigned a code, as given in the third column of Fig. 7-31. *SEQUENCE* and *SEQUENCE OF* share code 16. *CHOICE* does not have a code, since any actual value sent always has a specific type. The other codes are for types not used in SNMP.

Following the 1-byte identifier field comes a field telling how many bytes the data occupy. Lengths shorter than 128 bytes are directly encoded in 1 byte whose leftmost bit is 0. Those that are longer use multiple bytes, with first byte containing a 1 in the high-order bit and the length field (up to 127 bytes) in the low-order 7 bits. For example, if the data length is 1000 bytes, the first byte contains 130 to indicate a two byte length field follows. Then come two bytes whose value is 1000, with the high-order byte first.

The encoding of the data field depends on the type of data present. Integers are encoded in two's complement. A positive integer below 128 requires 1 byte, a positive integer below 32,768 requires 2 bytes, and so forth. The most significant byte is transmitted first.

Bit strings are encoded as themselves. The only problem is how to indicate the length. The length field tells how many *bytes* the value has, not how many *bits.* The solution chosen is to transmit 1 byte before the actual bit string telling how many bits (0 through 7) of the final byte are unused. Thus the encoding of the 9-bit string ′010011111′ would be 07, 4F, 80 (hexadecimal).

Octet strings are easy. The bytes of the string are just transmitted in standard big endian style, left to right.

The null value is indicated by setting the length field to 0. No numerical value is actually transmitted.

An *OBJECT IDENTIFIER* is encoded as the sequence of integers it represents. For example, the Internet is {1, 3, 6, 1}. However, since the first number is always 0, 1, or 2, and the second is less than 40 (by definition—ISO simply will not recognize the 41st category to show up on its doorstep), the first two numbers, *a and b*, are encoded as 1 byte having the value 40*a* + *b*. For the Internet, this number is 43. As usual, numbers exceeding 127 are encoded in multiple bytes, the first of which contains the high-order bit set to 1 and a byte count in the other 7 bits.

Both sequence types are transmitted by first sending the type or tag, then the total length of the encoding for all the fields, followed by the fields themselves. The fields are sent in order.

The encoding of a *CHOICE* value is the same as the encoding of the actual data structure being transferred.

An example showing encoding of some values is given in Fig. 7-34. The values encoded are the *INTEGER* 49, the *OCTET STRING* ′110′, ″xy″, the only possible value for *NULL*, the *OBJECT IDENTIFIER* for the Internet {1, 3, 6, 1}, and a *Gauge32* value of 14.

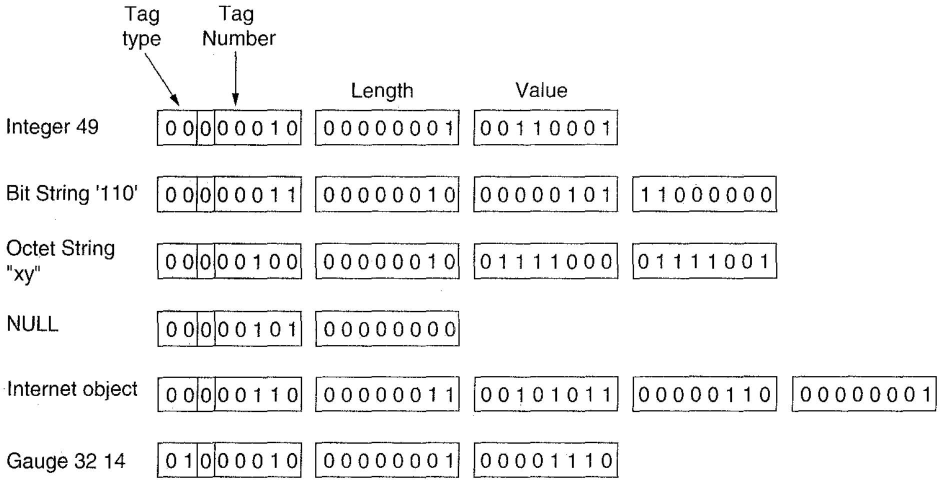


Fig. 7-34. ASN.1 encoding of some example values.

7.3.3. SMI—Structure of Management Information

In the preceding section, we have discussed only those parts of ASN.1 that are used in SNMP. In reality, the SNMP documents are organized differently. RFC 1442 first says that ASN.1 will be used to describe SNMP data structures, then it goes on for 57 pages scratching out parts of the ASN.1 standard that it does not want and adding new definitions (in ASN.1) that are needed. In particular, RFC 1442 defines four key macros and eight new data types that are heavily used throughout SNMP. It is this sub-super-set of ASN.1, which goes by the ungainly name of **SMI** (**Structure of Management Information**), that is really used to define the SNMP data structures.

Although this approach is somewhat bureaucratic, some rules and regulations are necessary if products from hundreds of vendors are expected to talk to one another and actually understand what the others are saying. A few words about SMI are therefore now in order.

At the lowest level, SNMP variables are defined as individual objects. Related objects are collected together into groups, and groups are assembled into modules. For example, groups exist for IP objects and TCP objects. A router might support the IP group, since its manager cares about how many packets it has lost. On the other hand, a low-end router might not support the TCP group, since it need not use TCP to perform its routing functions. It is the intention that vendors supporting a group support all the objects in that group. However, a vendor supporting a module need not support all of its groups, since not all may be applicable to the device.

All MIB modules start with an invocation of the *MODULE-IDENTITY* macro. Its parameters provide the name and address of the implementer, the revision history, and other administrative information. Typically, this call is followed by an invocation of the *OBJECT-IDENTITY* macro, which tells where the module fits in the naming tree of Fig. 7-32.

Later on come one or more invocations of the *OBJECT-TYPE* macro, which name the actual variables being managed and specify their properties. Grouping variables into groups is done by convention; there are no *BEGIN-GROUP* and *END-GROUP* statements in ASN.1 or SMI.

The *OBJECT-TYPE* macro has four required parameters and four (sometimes) optional ones. The first required parameter is *SYNTAX* and defines the variable's data type from among the types listed in Fig. 7-35. For the most part, these types should be self explanatory, with the following comments. The suffix 32 is used when the implementer really wants a 32-bit number, even if all the machines in sight have 64-bit CPUs. Gauges differ from counters in that they do not wrap around when they hit their limits. They stick there. If a router has lost exactly 2^{32} packets, it is better to report this as $2^{32} - 1$ than as 0. SMI also supports arrays, but we will not go into those here. For details, see (Rose, 1994).

In addition to requiring a specification of the data type used by the variable

Name	Type	Bytes	Meaning
INTEGER	Numeric	4	Integer (32 bits in current implementations)
Counter32	Numeric	4	Unsigned 32-bit counter that wraps
Gauge32	Numeric	4	Unsigned value that does not wrap
Integer32	Numeric	4	32 Bits, even on a 64-bit CPU
UInteger32	Numeric	4	Like Integer32, but unsigned
Counter64	Numeric	8	A 64-bit counter
TimeTicks	Numeric	4	In hundredths of a second since some epoch
BIT STRING	String	4	Bit map of 1 to 32 bits
OCTET STRING	String	≥ 0	Variable length byte string
Opaque	String	≥ 0	Obsolete; for backward compatibility only
OBJECT IDENTIFIER	String	>0	A list of integers from Fig. 7-32
IpAddress	String	4	A dotted decimal Internet address
NsapAddress	String	< 22	An OSI NSAP address

Fig. 7-35. Data types used for SNMP monitored variables.

being declared, the *OBJECT TYPE* macro also requires three other parameters. *MAX-ACCESS* contains information about the variable's access. The most common values are read-write and read-only. If a variable is read-write, the management station can set it. If it is read-only, the management station can read it but cannot set it.

The *STATUS* has three possible values. A current variable is conformant with the current SNMP specification. An obsolete variable is not conformant but was conformant with an older version. A deprecated variable is in between. It is really obsolete, but the committee that wrote the standard did not dare say this in public for fear of the reaction from vendors whose products use it. Nevertheless, the handwriting is on the wall.

The last required parameter is *DESCRIPTION*, which is an ASCII string telling what the variable does. If a manager buys a nice new shiny device, queries it from the management station, and discovers that it keeps track of *pktCnt*, fetching the *DESCRIPTION* field is supposed to give a clue as to what kind of packets it is counting. This field is intended exclusively for human (as opposed to computer) consumption.

A simple example of an *OBJECT TYPE* declaration is given in Fig. 7-36. The variable is called *lostPackets* and might be useful in a router or other device dealing with packets. The value after the ::= sign places it in the tree.

```
lostPackets OBJECT TYPE
   SYNTAX Counter32            -- use a 32-bit counter
   MAX-ACCESS read-only        -- the management station may not change it
   STATUS current              -- this variable is not obsolete (yet)
   DESCRIPTION
       "The number of packets lost since the last boot"
   ::= {experimental 20}
```

Fig. 7-36. An example SNMP variable.

7.3.4. The MIB—Management Information Base

The collection of objects managed by SNMP is defined in the MIB. For convenience, these objects are (currently) grouped into ten categories, which correspond to the ten nodes under *mib-2* in Fig. 7-32. (Note that *mib-2* corresponds to SNMPv2 and that object 9 is no longer present.) The ten categories are intended to provide a basis of what a management station should understand. New categories and objects will certainly be added in the future, and vendors are free to define additional objects for their products. The ten categories are summarized in Fig. 7-37.

Group	# Objects	Description
System	7	Name, location, and description of the equipment
Interfaces	23	Network interfaces and their measured traffic
AT	3	Address translation (deprecated)
IP	42	IP packet statistics
ICMP	26	Statistics about ICMP messages received
TCP	19	TCP algorithms, parameters, and statistics
UDP	6	UDP traffic statistics
EGP	20	Exterior gateway protocol traffic statistics
Transmission	0	Reserved for media-specific MIBs
SNMP	29	SNMP traffic statistics

Fig. 7-37. The object groups of the Internet MIB-II.

Although space limitations prevent us from delving into the details of all 175 objects defined in MIB-II, a few comments may be helpful. The system group allows the manager to find out what the device is called, who made it, what hardware and software it contains, where it is located, and what it is supposed to do. The time of the last boot and the name and address of the contact person are also provided. This information means that a company can contract out system management to another company in a distant city and have the latter be able to easily figure out what the configuration being managed actually is and who should be contacted if there are problems with various devices.

The interfaces group deals with the network adapters. It keeps track of the number of packets and bytes sent and received from the network, the number of discards, the number of broadcasts, and the current output queue size.

The AT group was present in MIB-I and provided information about address mapping (e.g., Ethernet to IP addresses). This information was moved to protocol-specific MIBs in SNMPv2.

The IP group deals with IP traffic into and out of the node. It is especially rich in counters keeping track of the number of packets discarded for each of a variety of reasons (e.g., no known route to the destination or lack of resources). Statistics about datagram fragmentation and reassembly are also available. All these items are particular important for managing routers.

The ICMP group is about IP error messages. Basically, it has a counter for each ICMP message that records how many of that type have been seen.

The TCP group monitors the current and cumulative number of connections opened, segments sent and received, and various error statistics.

The UDP group logs the number of UDP datagrams sent and received, and how many of the latter were undeliverable due to an unknown port or some other reason.

The EGP group is used for routers that support the exterior gateway protocol. It keeps track of how many packets of what kind went out, came in and were forwarded correctly, and came in and were discarded.

The transmission group is a place holder for media-specific MIBs. For example, Ethernet-specific statistics can be kept here. The purpose of including an empty group in MIB-II is to reserve the identifier {internet 2 1 9} for such purposes.

The last group is for collecting statistics about the operation of SNMP itself. How many messages are being sent, what kinds of messages are they, and so on.

MIB-II is formally defined in RFC 1213. The bulk of RFC 1213 consists of 175 macro calls similar to those of Fig. 7-36, with comments delineating the ten groups. For each of the 175 objects defined, the data type is given along with an English text description of what the variable is used for. For further information about MIB-II, the reader is referred to this RFC.

7.3.5. The SNMP Protocol

We have now seen that the model underlying SNMP is a management station that sends requests to agents in managed nodes, inquiring about the 175 variables just alluded to, and many other vendor-specific variables. Our last topic is the actual protocol that the management station and agents speak. The protocol itself is defined in RFC 1448.

The normal way that SNMP is used is that the management station sends a request to an agent asking it for information or commanding it to update its state in a certain way. Ideally, the agent just replies with the requested information or confirms that it has updated its state as requested. Data are sent using the ASN.1 transfer syntax. However, various errors can also be reported, such as No Such Variable.

SNMP defines seven messages that can be sent. The six messages from an initiator are listed in Fig. 7-38 (the seventh message is the response message). The first three request variable values to be sent back. The first format names the variables it wants explicitly. The second one asks for the next variable, allowing a manager to step through the entire MIB alphabetically (the default is the first variable). The third is for large transfers, such as tables.

Message	Description
Get-request	Requests the value of one or more variables
Get-next-request	Requests the variable following this one
Get-bulk-request	Fetches a large table
Set-request	Updates one or more variables
Inform-request	Manager-to-manager message describing local MIB
SnmpV2-trap	Agent-to-manager trap report

Fig. 7-38. SNMP message types.

Then comes a message that allows the manager to update an agent's variables, to the extent that the object specification permits such updates, of course. Next is an informational request that allows one manager to tell another one which variables it is managing. Finally, comes the message sent from an agent to a manager when a trap has sprung.

7.4. ELECTRONIC MAIL

Having finished looking at some of the support protocols used in the application layer, we finally come to real applications. When asked: "What are you going to do now?" few people will say: "I am going to look up some names with DNS." People do say they are going to read their email or news, surf the Web, or watch a movie over the net. In the remainder of this chapter, we will explain in a fair amount of detail how these four applications work.

Electronic mail, or **email**, as it is known to its many fans, has been around for over two decades. The first email systems simply consisted of file transfer protocols, with the convention that the first line of each message (i.e., file) contained the recipient's address. As time went on, the limitations of this approach became more obvious. Some of the complaints were

1. Sending a message to a group of people was inconvenient. Managers often need this facility to send memos to all their subordinates.
2. Messages had no internal structure, making computer processing difficult. For example, if a forwarded message was included in the body of another message, extracting the forwarded part from the received message was difficult.
3. The originator (sender) never knew if a message arrived or not.
4. If someone was planning to be away on business for several weeks and wanted all incoming email to be handled by his secretary, this was not easy to arrange.
5. The user interface was poorly integrated with the transmission system requiring users first to edit a file, then leave the editor and invoke the file transfer program.
6. It was not possible to create and send messages containing a mixture of text, drawings, facsimile, and voice.

As experience was gained, more elaborate email systems were proposed. In 1982, the ARPANET email proposals were published as RFC 821 (transmission protocol) and RFC 822 (message format). These have since become the de facto Internet standards. Two years later, CCITT drafted its X.400 recommendation, which was later taken over as the basis for OSI's MOTIS. In 1988, CCITT modified X.400 to align it with MOTIS. MOTIS was to be the flagship application for OSI, a system that was to be all things to all people.

After a decade of competition, email systems based on RFC 822 are widely used, whereas those based on X.400 have disappeared under the horizon. How a system hacked together by a handful of computer science graduate students beat an official international standard strongly backed by all the PTTs worldwide, many governments, and a substantial part of the computer industry brings to mind the Biblical story of David and Goliath. The reason for RFC 822's success is not that it is so good, but that X.400 is so poorly designed and so complex that nobody could implement it well. Given a choice between a simple-minded, but working, RFC 822-based email system and a supposedly truly wonderful, but nonworking, X.400 email system, most organizations chose the former. For a long diatribe on what is wrong with X.400, see Appendix C of (Rose, 1993). Consequently, our discussion of email will focus on RFC 821 and RFC 822 as used in the Internet.

7.4.1. Architecture and Services

In this section we will provide an overview of what email systems can do and how they are organized. They normally consist of two subsystems: the **user agents**, which allow people to read and send email, and the **message transfer agents**, which move the messages from the source to the destination. The user agents are local programs that provide a command-based, menu-based, or graphical method for interacting with the email system. The message transfer agents are typically system daemons that run in the background and move email through the system.

Typically, email systems support five basic functions, as described below. **Composition** refers to the process of creating messages and answers. Although any text editor can be used for the body of the message, the system itself can provide assistance with addressing and the numerous header fields attached to each message. For example, when answering a message, the email system can extract the originator's address from the incoming email and automatically insert it into the proper place in the reply.

Transfer refers to moving messages from the originator to the recipient. In large part, this requires establishing a connection to the destination or some intermediate machine, outputting the message, and releasing the connection. The email system should do this automatically, without bothering the user.

Reporting has to do with telling the originator what happened to the message. Was it delivered? Was it rejected? Was it lost? Numerous applications exist in which confirmation of delivery is important and may even have legal significance ("Well, Your Honor, my email system is not very reliable, so I guess the electronic subpoena just got lost somewhere").

Displaying incoming messages is needed so people can read their email. Sometimes conversion is required or a special viewer must be invoked, for example, if the message is a PostScript file or digitized voice. Simple conversions and formatting are sometimes attempted as well.

Disposition is the final step and concerns what the recipient does with the message after receiving it. Possibilities include throwing it away before reading, throwing it away after reading, saving it, and so on. It should also be possible to retrieve and reread saved messages, forward them, or process them in other ways.

In addition to these basic services, most email systems provide a large variety of advanced features. Let us just briefly mention a few of these. When people move, or when they are away for some period of time, they may want their email forwarded, so the system should be able to do this automatically.

Most systems allow users to create **mailboxes** to store incoming email. Commands are needed to create and destroy mailboxes, inspect the contents of mailboxes, insert and delete messages from mailboxes, and so on.

Corporate managers often need to send a message to each of their subordinates, customers, or suppliers. This gives rise to the idea of a **mailing list**, which is a list of email addresses. When a message is sent to the mailing list, identical copies are delivered to everyone on the list.

Registered email is another important idea, to allow the originator to know that his message has arrived. Alternatively, automatic notification of undeliverable email may be desired. In any case, the originator should have some control over reporting what happened.

Other advanced features are carbon copies, high-priority email, secret (encrypted) email, alternative recipients if the primary one is not available, and the ability for secretaries to handle their bosses' email.

Email is now widely used within industry for intracompany communication. It allows far-flung employees to cooperate on complex projects, even over many time zones. By eliminating most cues associated with rank, age, and gender, email debates tend to focus on ideas, not on corporate status. With email, a brilliant idea from a summer student can have more impact than a dumb one from an executive vice president. Some companies have estimated that email has improved their productivity by as much as 30 percent (Perry and Adam, 1992).

A key idea in all modern email systems is the distinction between the **envelope** and its contents. The envelope encapsulates the message. It contains all the information needed for transporting the message, such as the destination address, priority, and security level, all of which are distinct from the message itself. The message transport agents use the envelope for routing, just as the post office does.

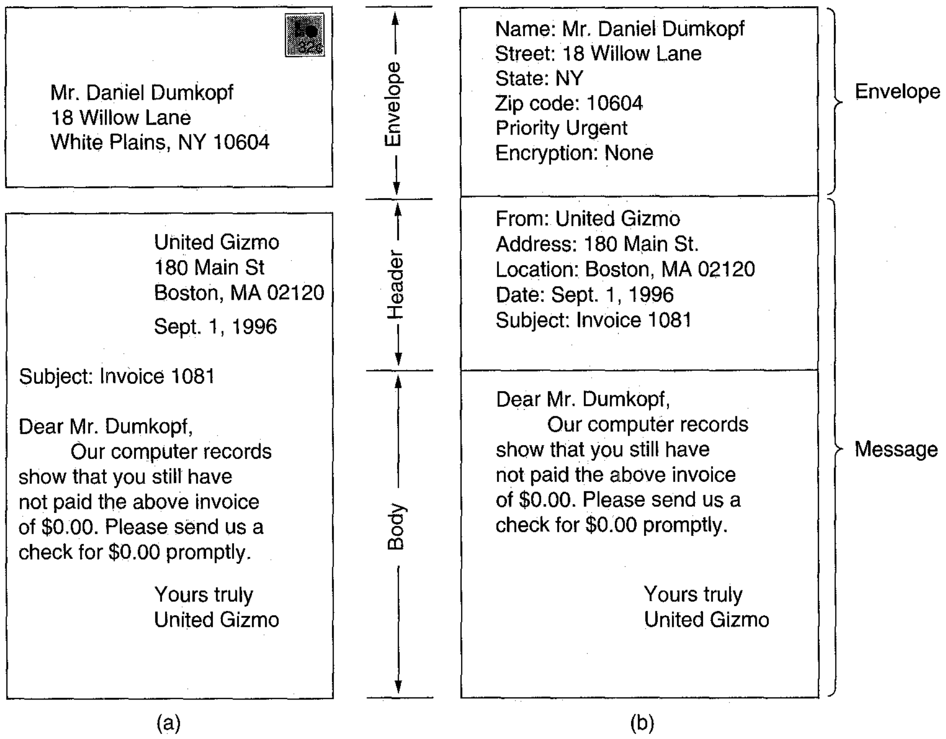
The message inside the envelope contains two parts: the **header** and the **body**. The header contains control information for the user agents. The body is entirely for the human recipient. Envelopes and messages are illustrated in Fig. 7-39.

7.4.2. The User Agent

Email systems have two basic parts, as we have seen: the user agents and the message transfer agents. In this section we will look at the user agents. A user agent is normally a program (sometimes called a mail reader) that accepts a variety of commands for composing, receiving, and replying to messages, as well as for manipulating mailboxes. Some user agents have a fancy menu- or icon-driven interface that requires a mouse, while others expect 1-character commands from the keyboard. Functionally, these are the same.

Sending Email

To send an email message, a user must provide the message, the destination address, and possibly some other parameters (e.g., the priority or security level). The message can be produced with a free-standing text editor, a word processing



Mr. Daniel Dumkopf
18 Willow Lane
White Plains, NY 10604

United Gizmo
180 Main St
Boston, MA 02120

Sept. 1, 1996

Subject: Invoice 1081

Dear Mr. Dumkopf,
Our computer records show that you still have not paid the above invoice of $0.00. Please send us a check for $0.00 promptly.

Yours truly
United Gizmo

Name: Mr. Daniel Dumkopf
Street: 18 Willow Lane
State: NY
Zip code: 10604
Priority Urgent
Encryption: None

From: United Gizmo
Address: 180 Main St.
Location: Boston, MA 02120
Date: Sept. 1, 1996
Subject: Invoice 1081

Dear Mr. Dumkopf,
Our computer records show that you still have not paid the above invoice of $0.00. Please send us a check for $0.00 promptly.

Yours truly
United Gizmo

Fig. 7-39. Envelopes and messages. (a) Postal email. (b) Electronic email.

program, or possibly with a text editor built into the user agent. The destination address must be in a format that the user agent can deal with. Many user agents expect DNS addresses of the form *mailbox@location*. Since we have studied these earlier in this chapter, we will not repeat that material here.

However, it is worth noting that other forms of addressing exist. In particular, X.400 addresses look radically different than DNS addresses. They are composed of *attribute = value* pairs, for example,

/C=US/SP=MASSACHUSETTS/L=CAMBRIDGE/PA=360 MEMORIAL DR./CN=KEN SMITH/

This address specifies a country, state, locality, personal address and a common name (Tom Smith). Many other attributes are possible, so you can send email to someone whose name you do not know, provided you know enough other attributes (e.g., company and job title). Many people feel that this form of naming is considerably less convenient than DNS names.

In all fairness, however, the X.400 designers assumed that people would use **aliases** (short user-assigned strings) to identify recipients, so that they would never even see the full addresses. However, the necessary software was never widely available, so people sending mail to users with X.400 addresses often had to type in strings like the one above. In contrast, most email systems for the Internet have always allowed users to have alias files.

Most email systems support mailing lists, so that a user can send the same message to a list of people with a single command. If the mailing list is maintained locally, the user agent can just send a separate message to each intended recipient. However, if the list is maintained remotely, then messages will be expanded there. For example, if a group of bird watchers have a mailing list called *birders* installed on *meadowlark.arizona.edu*, then any message sent to *birders@meadowlark.arizona.edu* will be routed to the University of Arizona and expanded there into individual messages to all the mailing list members, wherever in the world they may be. Users of this mailing list cannot tell that it is a mailing list. It could just as well be the personal mailbox of Prof. Gabriel O. Birders.

Reading Email

Typically, when a user agent is started up, it will look at the user's mailbox for incoming email before displaying anything on the screen. Then it may announce the number of messages in the mailbox or display a one-line summary of each one and wait for a command.

As an example of how a user agent works, let us take a look at a typical mail scenario. After starting up the user agent, the user asks for a summary of his email. A display like that of Fig. 7-40 then appears on the screen. Each line refers to one message. In this example, the mailbox contains eight messages.

#	Flags	Bytes	Sender	Subject
1	K	1030	asw	Changes to MINIX
2	KA	6348	radia	Comments on material you sent me
3	K F	4519	Amy N. Wong	Request for information
4		1236	bal	Deadline for grant proposal
5		103610	kaashoek	Text of DCS paper
6		1223	emily E.	Pointer to WWW page
7		3110	saniya	Referee reports for the paper
8		1204	dmr	Re: My student's visit

Fig. 7-40. An example display of the contents of a mailbox.

Each display line contains several fields extracted from the envelope or header of the corresponding message. In a simple email system, the choice of fields displayed is built into the program. In a more sophisticated system, the user can specify which fields are to be displayed by providing a **user profile**, a file describing the display format. In this example, the first field is the message number. The second field, *Flags*, can contain a *K*, meaning that the message is not new but was read previously and kept in the mailbox; an *A*, meaning that the message has already been answered; and/or an *F*, meaning that the message has been forwarded to someone else. Other flags are also possible.

The third field tells how long the message is and the fourth one tells who sent the message. Since this field is simply extracted from the message, this field may contain first names, full names, initials, login names, or whatever else the sender chooses to put there. Finally, the *Subject* field gives a brief summary of what the message is about. People who fail to include a *Subject* field often discover that responses to their email tend not to get the highest priority.

After the headers have been displayed, the user can perform any of the commands available. A typical collection is listed in Fig. 7-41. Some of the commands require a parameter. The # sign means that the number of a message (or perhaps several messages) is expected. Alternatively, the letter *a* can be used to mean all messages.

Command	Parameter	Description
h	#	Display header(s) on the screen
c		Display current header only
t	#	Type message(s) on the screen
s	address	Send a message
f	#	Forward message(s)
a	#	Answer message(s)
d	#	Delete message(s)
u	#	Undelete previously deleted message(s)
m	#	Move message(s) to another mailbox
k	#	Keep message(s) after exiting
r	mailbox	Read a new mailbox
n		Go to the next message and display it
b		Backup to the previous message and display it
g	#	Go to a specific message but do not display it
e		Exit the mail system and update the mailbox

Fig. 7-41. Typical mail handling commands.

Innumerable email programs exist. Our example email program is patterned after the one used by the UNIX Mmdf system, as it is quite straightforward. The *h* command displays one or more headers in the format of Fig. 7-40. The *c* command prints the current message's header. The *t* command types (i.e., displays on the screen) the requested message or messages. Possible commands are *t 3*, to type message 3, *t 4–6*, to type messages 4 through 6, and *t a* to type them all.

The next group of three commands deals with sending messages rather than receiving them. The *s* command sends a message by calling an appropriate editor (e.g., specified in the user's profile) to allow the user to compose the message. Spelling, grammar, and diction checkers can see if the message is syntactically correct. Unfortunately, the current generation of email programs do not have checkers to see if the sender knows what he is talking about. When the message is finished, it is prepared for transmission to the message transfer agent.

The *f* command forwards a message from the mailbox, prompting for an address to send it to. The *a* command extracts the source address from the message to be answered and calls the editor to allow the user to compose the reply.

The next group of commands is for manipulating mailboxes. Users typically have one mailbox for each person with whom they correspond, in addition to the mailbox for incoming email that we have already seen. The *d* command deletes a message from the mailbox, but the *u* command undoes the delete. (The message is not actually deleted until the email program is exited.) The *m* command moves a message to another mailbox. This is the usual way to save important email after reading it. The *k* command keeps the indicated message in the mailbox even after it is read. If a message is read but not explicitly kept, some default action is taken when the email program is exited, such as moving it to a special default mailbox. Finally, the *r* command is used to finish up with the current mailbox and go read another one.

The *n*, *b*, and *g* commands are for moving about in the current mailbox. It is common for a user to read message 1, answer, move, or delete it, and then type *n* to get the next one. The value of this command is that the user does not have to keep track of where he is. It is possible to go backward using *b* or to a given message with *g*.

Finally, the *e* command exits the email program and makes whatever changes are required, such as deleting some messages and marking others as kept. This command overwrites the mailbox, replacing its contents.

In mail systems designed for beginners, each of these commands is typically associated with an on-screen icon, so that the user does not have to remember that *a* stands for *answer*. Instead, she has to remember that the little picture of a person with his mouth open means answer and not display message.

It should be clear from this example that email has come a long way from the days when it was just file transfer. Sophisticated user agents make managing a large volume of email possible. For people such as the author who (reluctantly) receive and send thousands of messages a year, such tools are invaluable.

7.4.3. Message Formats

Let us now turn from the user interface to the format of the email messages themselves. First we will look at basic ASCII email using RFC 822. After that, we will look at multimedia extensions to RFC 822

RFC 822

Messages consist of a primitive envelope (described in RFC 821), some number of header fields, a blank line, and then the message body. Each header field (logically) consists of a single line of ASCII text containing the field name, a colon, and, for most fields, a value. RFC 822 is an old standard, and does not clearly distinguish envelope from header fields, as a new standard would do. In normal usage, the user agent builds a message and passes it to the message transfer agent, which then uses some of the header fields to construct the actual envelope, a somewhat old-fashioned mixing of message and envelope.

The principal header fields related to message transport are listed in Fig. 7-42. The *To:* field gives the DNS address of the primary recipient. Having multiple recipients is also allowed. The *Cc:* field gives the addresses of any secondary recipients. In terms of delivery, there is no distinction between the primary and secondary recipients. It is entirely a psychological difference that may be important to the people involved but is not important to the mail system. The term *Cc:* (Carbon copy) is a bit dated, since computers do not use carbon paper, but it is well established. The *Bcc:* (Blind carbon copy) field is like the *Cc:* field, except that this line is deleted from all the copies sent to the primary and secondary recipients. This feature allows people to send copies to third parties without the primary and secondary recipients knowing this.

Header	Meaning
To:	Email address(es) of primary recipient(s)
Cc:	Email address(es) of secondary recipient(s)
Bcc:	Email address(es) for blind carbon copies
From:	Person or people who created the message
Sender:	Email address of the actual sender
Received:	Line added by each transfer agent along the route
Return-Path:	Can be used to identify a path back to the sender

Fig. 7-42. RFC 822 header fields related to message transport.

The next two fields, *From:* and *Sender:* tell who wrote and sent the message, respectively. These may not be the same. For example, a business executive may write a message, but her secretary may be the one who actually transmits it. In this case, the executive would be listed in the *From:* field and the secretary in the *Sender:* field. The *From:* field is required, but the *Sender:* field may be omitted if it is the same as the *From:* field. These fields are needed in case the message is undeliverable and must be returned to the sender.

A line containing *Received:* is added by each message transfer agent along the way. The line contains the agent's identity, the date and time the message was received, and other information that can be used for finding bugs in the routing system.

The *Return-Path:* field is added by the final message transfer agent and was intended to tell how to get back to the sender. In theory, this information can be gathered from all the *Received:* headers (except for the name of the sender's mailbox), but it is rarely filled in as such and typically just contains the sender's address.

In addition to the fields of Fig. 7-42, RFC 822 messages may also contain a variety of header fields used by the user agents or human recipients. The most common ones are listed in Fig. 7-43. Most of these are self-explanatory, so we will not go into all of them in detail.

Header	Meaning
Date:	The date and time the message was sent
Reply-To:	Email address to which replies should be sent
Message-Id:	Unique number for referencing this message later
In-Reply-To:	Message-Id of the message to which this is a reply
References:	Other relevant Message-Ids
Keywords:	User chosen keywords
Subject:	Short summary of the message for the one-line display

Fig. 7-43. Some fields used in the RFC 822 message header.

The *Reply-To:* field is sometimes used when neither the person composing the message nor the person sending the message wants to see the reply. For example, a marketing manager writes an email message telling customers about a new product. The message is sent by a secretary, but the *Reply-To:* field lists the head of the sales department, who can answer questions and take orders.

The RFC 822 document explicitly says that users are allowed to invent new headers for their own private use, provided that these headers start with the string *X-*. It is guaranteed that no future headers will use names starting with *X-*, to avoid conflicts between official and private headers. Sometimes wiseguy undergraduates include fields like *X-Fruit-of-the-Day:* or *X-Disease-of-the-Week:*, which are legal, although not always illuminating.

After the headers comes the message body. Users can put whatever they want here. Some people terminate their messages with elaborate signatures, including simple ASCII cartoons, quotations from greater and lesser authorities, political statements, and disclaimers of all kinds (e.g., The ABC Corporation is not responsible for my opinions; it cannot even comprehend them).

MIME—Multipurpose Internet Mail Extensions

In the early days of the ARPANET, email consisted exclusively of text messages written in English and expressed in ASCII. For this environment, RFC 822 did the job completely: it specified the headers but left the content entirely up to the users. Nowadays, on the worldwide Internet, this approach is no longer adequate. The problems include sending and receiving

1. Messages in languages with accents (e.g., French and German).
2. Messages in nonLatin alphabets (e.g., Hebrew and Russian).
3. Messages in languages without alphabets (e.g., Chinese and Japanese).
4. Messages not containing text at all (e.g., audio and video).

A solution was proposed in RFC 1341 and updated in RFC 1521. This solution, called **MIME (Multipurpose Internet Mail Extensions)** is now widely used. We will now describe it. For additional information about MIME, see RFC 1521 or (Rose, 1993).

The basic idea of MIME is to continue to use the RFC 822 format, but to add structure to the message body and define encoding rules for non-ASCII messages. By not deviating from 822, MIME messages can be sent using the existing mail programs and protocols. All that has to be changed are the sending and receiving programs, which users can do for themselves.

MIME defines five new message headers, as shown in Fig. 7-44. The first of these simply tells the user agent receiving the message that it is dealing with a MIME message, and which version of MIME it uses. Any message not containing a *MIME-Version:* header is assumed to be an English plaintext message, and is processed as such.

Header	Meaning
MIME-Version:	Identifies the MIME version
Content-Description:	Human-readable string telling what is in the message
Content-Id:	Unique identifier
Content-Transfer-Encoding:	How the body is wrapped for transmission
Content-Type:	Nature of the message

Fig. 7-44. RFC 822 headers added by MIME.

The *Content-Description:* header is an ASCII string telling what is in the message. This header is needed so the recipient will know whether it is worth decoding and reading the message. If the string says: "Photo of Barbara's gerbil" and the person getting the message is not a big gerbil fan, the message will probably be discarded rather than decoded into a high-resolution color photograph.