The firewall in this configuration has two components: two routers that do packet filtering and an application gateway. Simpler configurations also exist, but the advantage of this design is that every packet must transit two filters and an application gateway to go in or out. No other route exists. Readers who think that one security checkpoint is enough clearly have not made an international flight on a scheduled airline recently.

Each **packet filter** is a standard router equipped with some extra functionality. The extra functionality allows every incoming or outgoing packet to be inspected. Packets meeting some criterion are forwarded normally. Those that fail the test are dropped.

In Fig. 5-43, most likely the packet filter on the inside LAN checks outgoing packets and the one on the outside LAN checks incoming packets. Packets crossing the first hurdle go to the application gateway for further examination. The point of putting the two packet filters on different LANs is to ensure that no packet gets in or out without having to pass through the application gateway: there is no path around it.

Packet filters are typically driven by tables configured by the system administrator. These tables list sources and destinations that are acceptable, sources and destinations that are blocked, and default rules about what to do with packets coming from or going to other machines.

In the common case of a UNIX setting, a source or destination consists of an IP address and a port. Ports indicate which service is desired. For example, port 23 is for Telnet, port 79 is for Finger, and port 119 is for USENET news. A company could block incoming packets for all IP addresses combined with one of these ports. In this way, no one outside the company could log in via Telnet, or look up people using the Finger daemon. Furthermore, the company would be spared from having employees spend all day reading USENET news.

Blocking outgoing packets is trickier because although most sites stick to the standard port naming conventions, they are not forced to do so. Furthermore, for some important services, such as FTP (File Transfer Protocol), port numbers are assigned dynamically. In addition, although blocking TCP connections is difficult, blocking UDP packets is even harder because so little is known a priori about what they will do. Many packet filters simply ban UDP traffic altogether.

The second half of the firewall mechanism is the **application gateway**. Rather than just looking at raw packets, the gateway operates at the application level. A mail gateway, for example, can be set up to examine each message going in or coming out. For each one it makes a decision to transmit or discard it based on header fields, message size, or even the content (e.g., at a military installation, the presence of words like "nuclear" or "bomb" might cause some special action to be taken).

Installations are free to set up one or more application gateways for specific applications, but it is not uncommon for suspicious organizations to permit email in and out, and perhaps use of the World Wide Web, but ban everything else as

too dicey. Combined with encryption and packet filtering, this arrangement offers a limited amount of security at the cost of some inconvenience.

One final note concerns wireless communication and firewalls. It is easy to design a system that is logically completely secure, but which, in practice, leaks like a sieve. This situation can occur if some of the machines are wireless and use radio communication, which passes right over the firewall in both directions.

5.5. THE NETWORK LAYER IN THE INTERNET

At the network layer, the Internet can be viewed as a collection of subnetworks or **Autonomous Systems** (**ASes**) that are connected together. There is no real structure, but several major backbones exist. These are constructed from high-bandwidth lines and fast routers. Attached to the backbones are regional (midlevel) networks, and attached to these regional networks are the LANs at many universities, companies, and Internet service providers. A sketch of this quasihierarchical organization is given in Fig. 5-44.

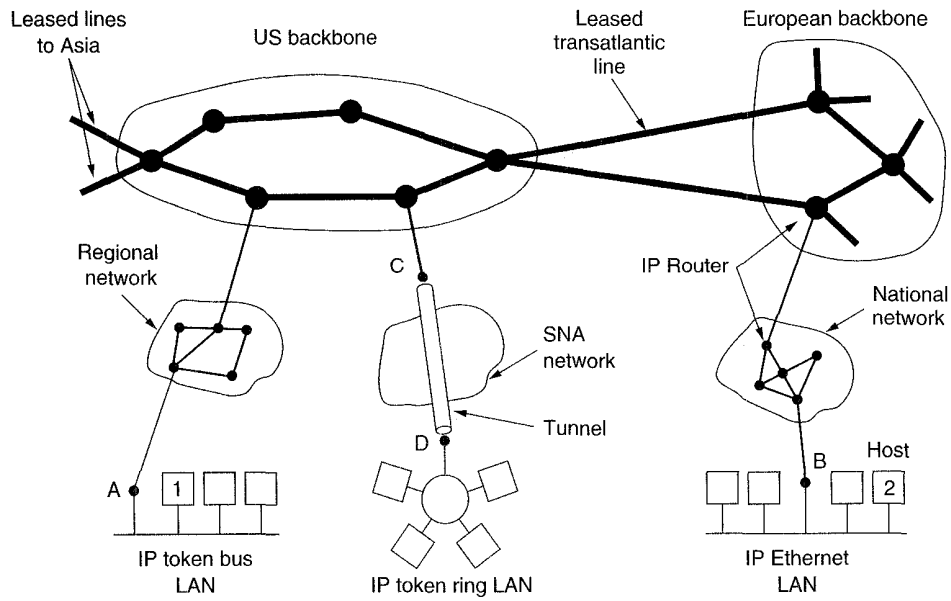


Fig. 5-44. The Internet is an interconnected collection of many networks.

The glue that holds the Internet together is the network layer protocol, **IP** (**Internet Protocol**). Unlike most older network layer protocols, it was designed from the beginning with internetworking in mind. A good way to think of the network layer is this. Its job is to provide a best-efforts way to transport datagrams

from source to destination, without regard to whether or not these machines are on the same network, or whether or not there are other networks in between them.

Communication in the Internet works as follows. The transport layer takes data streams and breaks them up into datagrams. In theory, datagrams can be up to 64 Kbytes each, but in practice they are usually around 1500 bytes. Each datagram is transmitted through the Internet, possibly being fragmented into smaller units as it goes. When all the pieces finally get to the destination machine, they are reassembled by the network layer into the original datagram. This datagram is then handed to the transport layer, which inserts it into the receiving process' input stream.

5.5.1. The IP Protocol

An appropriate place to start our study of the network layer in the Internet is the format of the IP datagrams themselves. An IP datagram consists of a header part and a text part. The header has a 20-byte fixed part and a variable length optional part. The header format is shown in Fig. 5-45. It is transmitted in big endian order: from left to right, with the high-order bit of the *Version* field going first. (The SPARC is big endian; the Pentium is little endian.) On little endian machines, software conversion is required on both transmission and reception.

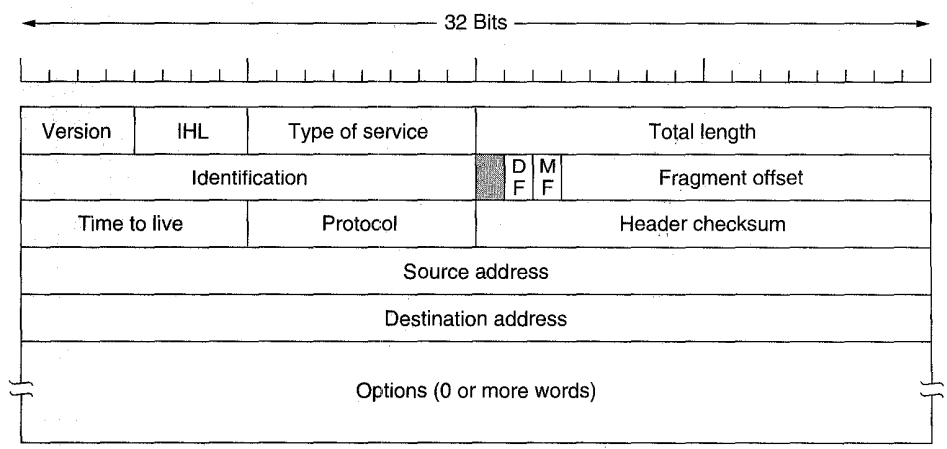


Fig. 5-45. The IP (Internet Protocol) header.

The *Version* field keeps track of which version of the protocol the datagram belongs to. By including the version in each datagram, it becomes possible to have the transition between versions take months, or even years, with some machines running the old version and others running the new one.

Since the header length is not constant, a field in the header, *IHL*, is provided to tell how long the header is, in 32-bit words. The minimum value is 5, which

applies when no options are present. The maximum value of this 4-bit field is 15, which limits the header to 60 bytes, and thus the options field to 40 bytes. For some options, such as one that records the route a packet has taken, 40 bytes is far too small, making the option useless.

The *Type of service* field allows the host to tell the subnet what kind of service it wants. Various combinations of reliability and speed are possible. For digitized voice, fast delivery beats accurate delivery. For file transfer, error-free transmission is more important than fast transmission.

The field itself contains (from left to right), a three-bit *Precedence* field, three flags, *D*, *T*, and *R*, and 2 unused bits. The *Precedence* field is a priority, from 0 (normal) to 7 (network control packet). The three flag bits allow the host to specify what it cares most about from the set {Delay, Throughput, Reliability}. In theory, these fields allow routers to make choices between, for example, a satellite link with high throughput and high delay or a leased line with low throughput and low delay. In practice, current routers ignore the *Type of Service* field altogether.

The *Total length* includes everything in the datagram—both header and data. The maximum length is 65,535 bytes. At present, this upper limit is tolerable, but with future gigabit networks larger datagrams will be needed.

The *Identification* field is needed to allow the destination host to determine which datagram a newly arrived fragment belongs to. All the fragments of a datagram contain the same *Identification* value.

Next comes an unused bit and then two 1-bit fields. *DF* stands for Don't Fragment. It is an order to the routers not to fragment the datagram because the destination is incapable of putting the pieces back together again. For example, when a computer boots, its ROM might ask for a memory image to be sent to it as a single datagram. By marking the datagram with the *DF* bit, the sender knows it will arrive in one piece, even if this means that the datagram must avoid a small-packet network on the best path and take a suboptimal route. All machines are required to accept fragments of 576 bytes or less.

MF stands for More Fragments. All fragments except the last one have this bit set. It is needed to know when all fragments of a datagram have arrived.

The *Fragment offset* tells where in the current datagram this fragment belongs. All fragments except the last one in a datagram must be a multiple of 8 bytes, the elementary fragment unit. Since 13 bits are provided, there is a maximum of 8192 fragments per datagram, giving a maximum datagram length of 65,536 bytes, one more than the *Total length* field.

The *Time to live* field is a counter used to limit packet lifetimes. It is supposed to count time in seconds, allowing a maximum lifetime of 255 sec. It must be decremented on each hop and is supposed to be decremented multiple times when queued for a long time in a router. In practice, it just counts hops. When it hits zero, the packet is discarded and a warning packet is sent back to the source host. This feature prevents datagrams for wandering around forever, something that otherwise might happen if the routing tables ever become corrupted.

When the network layer has assembled a complete datagram, it needs to know what to do with it. The *Protocol* field tells it which transport process to give it to. TCP is one possibility, but so are UDP and some others. The numbering of protocols is global across the entire Internet and is defined in RFC 1700.

The *Header checksum* verifies the header only. Such a checksum is useful for detecting errors generated by bad memory words inside a router. The algorithm is to add up all the 16-bit halfwords as they arrive, using one's complement arithmetic and then take the one's complement of the result. For purposes of this algorithm, the *Header checksum* is assumed to be zero upon arrival. This algorithm is more robust than using a normal add. Note that the *Header checksum* must be recomputed at each hop, because at least one field always changes (the *Time to live* field), but tricks can be used to speed up the computation.

The *Source address* and *Destination address* indicate the network number and host number. We will discuss Internet addresses in the next section. The *Options* field was designed to provide an escape to allow subsequent versions of the protocol to include information not present in the original design, to permit experimenters to try out new ideas, and to avoid allocating header bits to information that is rarely needed. The options are variable length. Each begins with a 1-byte code identifying the option. Some options are followed by a 1-byte option length field, and then one or more data bytes. The *Options* field is padded out to a multiple of four bytes. Currently five options are defined, as listed in Fig. 5-46, but not all routers support all of them.

Option	Description
Security	Specifies how secret the datagram is
Strict source routing	Gives the complete path to be followed
Loose source routing	Gives a list of routers not to be missed
Record route	Makes each router append its IP address
Timestamp	Makes each router append its address and timestamp

Fig. 5-46. IP options.

The *Security* option tells how secret the information is. In theory, a military router might use this field to specify not to route through certain countries the military considers to be "bad guys." In practice, all routers ignore it, so its only practical function is to help spies find the good stuff more easily.

The *Strict source routing* option gives the complete path from source to destination as a sequence of IP addresses. The datagram is required to follow that exact route. It is most useful for system managers to send emergency packets when the routing tables are corrupted, or for making timing measurements.

The *Loose source routing* option requires the packet to traverse the list of routers specified, and in the order specified, but it is allowed to pass through other

routers on the way. Normally, this option would only provide a few routers, to force a particular path. For example, to force a packet from London to Sydney to go west instead of east, this option might specify routers in New York, Los Angeles, and Honolulu. This option is most useful when political or economic considerations dictate passing through or avoiding certain countries.

The *Record route* option tells the routers along the path to append their IP address to the option field. This allows system managers to track down bugs in the routing algorithms ("Why are packets from Houston to Dallas all visiting Tokyo first?") When the ARPANET was first set up, no packet ever passed through more than nine routers, so 40 bytes of option was ample. As mentioned above, now it is too small.

Finally, the *Timestamp* option is like the *Record route* option, except that in addition to recording its 32-bit IP address, each router also records a 32-bit timestamp. This option, too, is mostly for debugging routing algorithms.

5.5.2. IP Addresses

Every host and router on the Internet has an IP address, which encodes its network number and host number. The combination is unique: no two machines have the same IP address. All IP addresses are 32 bits long and are used in the *Source address* and *Destination address* fields of IP packets. The formats used for IP address are shown in Fig. 5-47. Those machines connected to multiple networks have a different IP address on each network.

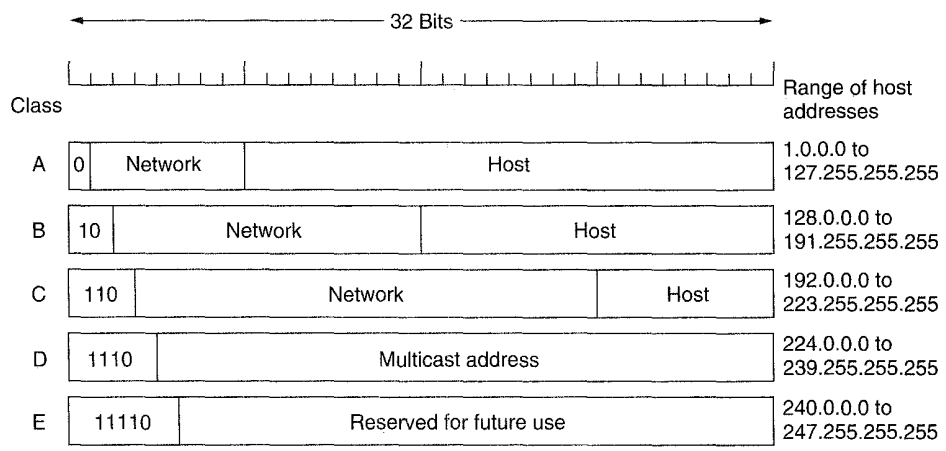


Fig. 5-47. IP address formats.

The class A, B, C, and D formats allow for up to 126 networks with 16 million hosts each, 16,382 networks with up to 64K hosts, 2 million networks, (e.g.,

LANs), with up to 254 hosts each, and multicast, in which a datagram is directed to multiple hosts. Addresses beginning with 11110 are reserved for future use. Tens of thousands of networks are now connected to the Internet, and the number doubles every year. Network numbers are assigned by the **NIC** (**Network Information Center**) to avoid conflicts.

Network addresses, which are 32-bit numbers, are usually written in **dotted decimal notation**. In this format, each of the 4 bytes is written in decimal, from 0 to 255. For example, the hexadecimal address C0290614 is written as 192.41.6.20. The lowest IP address is 0.0.0.0 and the highest is 255.255.255.255.

The values 0 and −1 have special meanings, as shown in Fig. 5-48. The value 0 means this network or this host. The value of −1 is used as a broadcast address to mean all hosts on the indicated network.

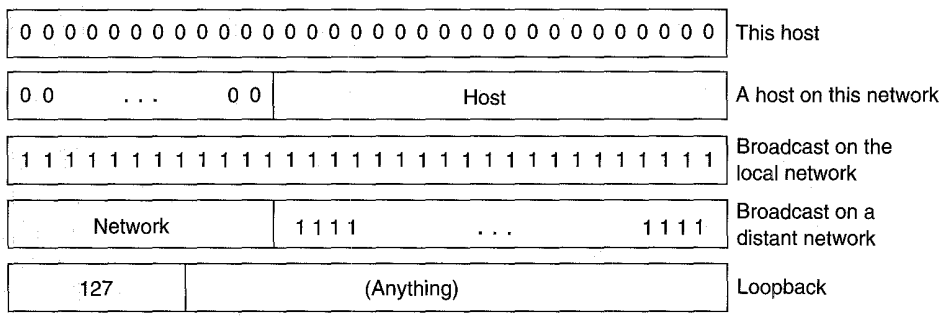


Fig. 5-48. Special IP addresses.

The IP address 0.0.0.0 is used by hosts when they are being booted but is not used afterward. IP addresses with 0 as network number refer to the current network. These addresses allow machines to refer to their own network without knowing its number (but they have to know its class to know how many 0s to include). The address consisting of all 1s allows broadcasting on the local network, typically a LAN. The addresses with a proper network number and all 1s in the host field allow machines to send broadcast packets to distant LANs anywhere in the Internet. Finally, all addresses of the form 127.xx.yy.zz are reserved for loopback testing. Packets sent to that address are not put out onto the wire; they are processed locally and treated as incoming packets. This allows packets to be sent to the local network without the sender knowing its number. This feature is also used for debugging network software.

5.5.3. Subnets

As we have seen, all the hosts in a network must have the same network number. This property of IP addressing can cause problems as networks grow. For example, consider a company that starts out with one class C LAN on the

Internet. As time goes on, it might acquire more than 254 machines, and thus need a second class C address. Alternatively, it might acquire a second LAN of a different type and want a separate IP address for it (the LANs could be bridged to form a single IP network, but bridges have their own problems). Eventually, it might end up with many LANs, each with its own router and each with its own class C network number.

As the number of distinct local networks grows, managing them can become a serious headache. Every time a new network is installed the system administrator has to contact NIC to get a new network number. Then this number must be announced worldwide. Furthermore, moving a machine from one LAN to another requires it to change its IP address, which in turn may mean modifying its configuration files and also announcing the new IP address to the world. If some other machine is given the newly-released IP address, that machine will get email and other data intended for the original machine until the address has propagated all over the world.

The solution to these problems is to allow a network to be split into several parts for internal use but still act like a single network to the outside world. In the Internet literature, these parts are called **subnets**. As we mentioned in Chap. 1, this usage conflicts with "subnet" to mean the set of all routers and communication lines in a network. Hopefully it will be clear from the context which meaning is intended. In this section, the new definition will be the one used. If our growing company started up with a class B address instead of a class C address, it could start out just numbering the hosts from 1 to 254. When the second LAN arrived, it could decide, for example, to split the 16-bit host number into a 6-bit subnet number and a 10-bit host number, as shown in Fig. 5-49. This split allows 62 LANs (0 and −1 are reserved), each with up to 1022 hosts.

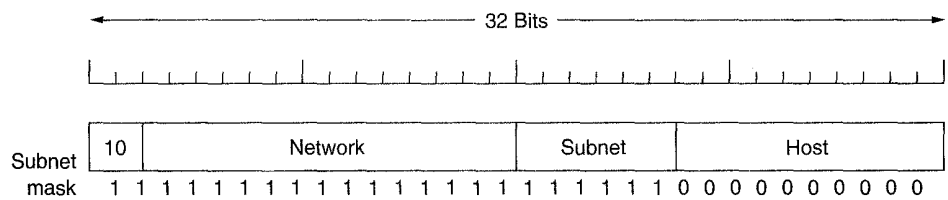


Fig. 5-49. One of the ways to subnet a class B network.

Outside the network, the subnetting is not visible, so allocating a new subnet does not require contacting NIC or changing any external databases. In this example, the first subnet might use IP addresses starting at 130.50.4.1, the second subnet might start at 130.50.8.1, and so on.

To see how subnets work, it is necessary to explain how IP packets are processed at a router. Each router has a table listing some number of (network, 0) IP addresses and some number of (this-network, host) IP addresses. The first kind

tells how to get to distant networks. The second kind tells how to get to local hosts. Associated with each table is the network interface to use to reach the destination, and certain other information.

When an IP packet arrives, its destination address is looked up in the routing table. If the packet is for a distant network, it is forwarded to the next router on the interface given in the table. If it is a local host (e.g., on the router's LAN), it is sent directly to the destination. If the network is not present, the packet is forwarded to a default router with more extensive tables. This algorithm means that each router only has to keep track of other networks and local hosts, not (network, host) pairs, greatly reducing the size of the routing table.

When subnetting is introduced, the routing tables are changed, adding entries of the form (this-network, subnet, 0) and (this-network, this-subnet, host). Thus a router on subnet *k* knows how to get to all the other subnets and also how to get to all the hosts on subnet *k*. It does not have to know the details about hosts on other subnets. In fact, all that needs to be changed is to have each router do a Boolean AND with the network's **subnet mask** (see Fig. 5-49) to get rid of the host number and look up the resulting address in its tables (after determining which network class it is). For example, a packet addressed to 130.50.15.6 and arriving at a router on subnet 5 is ANDed with the subnet mask of Fig. 5-49 to give the address 130.50.12.0. This address is looked up in the routing tables to find out how to get to hosts on subnet 3. The router on subnet 5 is thus spared the work of keeping track of the data link addresses of hosts other than those on subnet 5. Subnetting thus reduces router table space by creating a three-level hierarchy.

5.5.4. Internet Control Protocols

In addition to IP, which is used for data transfer, the Internet has several control protocols used in the network layer, including ICMP, ARP, RARP, and BOOTP. In this section we will look at each of these in turn.

The Internet Control Message Protocol

The operation of the Internet is monitored closely by the routers. When something unexpected occurs, the event is reported by the **ICMP (Internet Control Message Protocol)**, which is also used to test the Internet. About a dozen types of ICMP messages are defined. The most important ones are listed in Fig. 5-50. Each ICMP message type is encapsulated in an IP packet.

The DESTINATION UNREACHABLE message is used when the subnet or a router cannot locate the destination, or a packet with the *DF* bit cannot be delivered because a "small-packet" network stands in the way.

The TIME EXCEEDED message is sent when a packet is dropped due to its counter reaching zero. This event is a symptom that packets are looping, that there is enormous congestion, or that the timer values are being set too low.

Message type	Description
Destination unreachable	Packet could not be delivered
Time exceeded	Time to live field hit 0
Parameter problem	Invalid header field
Source quench	Choke packet
Redirect	Teach a router about geography
Echo request	Ask a machine if it is alive
Echo reply	Yes, I am alive
Timestamp request	Same as Echo request, but with timestamp
Timestamp reply	Same as Echo reply, but with timestamp

Fig. 5-50. The principal ICMP message types.

The PARAMETER PROBLEM message indicates that an illegal value has been detected in a header field. This problem indicates a bug in the sending host's IP software, or possibly in the software of a router transited.

The SOURCE QUENCH message was formerly used to throttle hosts that were sending too many packets. When a host received this message, it was expected to slow down. It is rarely used any more because when congestion occurs, these packets tend to add more fuel to the fire. Congestion control in the Internet is now done largely in the transport layer and will be studied in detail in Chap. 6.

The REDIRECT message is used when a router notices that a packet seems to be routed wrong. It is used by the router to tell the sending host about the probable error.

The ECHO REQUEST and ECHO REPLY messages are used to see if a given destination is reachable and alive. Upon receiving the ECHO message, the destination is expected to send an ECHO REPLY message back. The TIMESTAMP REQUEST and TIMESTAMP REPLY messages are similar, except that the arrival time of the message and the departure time of the reply are recorded in the reply. This facility is used to measure network performance.

In addition to these messages, there are four others that deal with Internet addressing, to allow hosts to discover their network numbers and to handle the case of multiple LANs sharing a single IP address. ICMP is defined in RFC 792.

The Address Resolution Protocol

Although every machine on the Internet has one (or more) IP addresses, these cannot actually be used for sending packets because the data link layer hardware does not understand Internet addresses. Nowadays, most hosts are attached to a

LAN by an interface board that only understands LAN addresses. For example, every Ethernet board ever manufactured comes equipped with a 48-bit Ethernet address. Manufacturers of Ethernet boards request a block of addresses from a central authority to ensure that no two boards have the same address (to avoid conflicts should the two boards ever appear on the same LAN). The boards send and receive frames based on 48-bit Ethernet addresses. They know nothing at all about 32-bit IP addresses.

The question now arises: How do IP addresses get mapped onto data link layer addresses, such as Ethernet? To explain how this works, let us use the example of Fig. 5-51, in which a small university with several class C networks is illustrated. Here we have two Ethernets, one in the Computer Science department, with IP address 192.31.65.0 and one in Electrical Engineering, with IP address 192.31.63.0. These are connected by a campus FDDI ring with IP address 192.31.60.0. Each machine on an Ethernet has a unique Ethernet address, labeled *E1* through *E6*, and each machine on the FDDI ring has an FDDI address, labeled *F1* through *F3*.

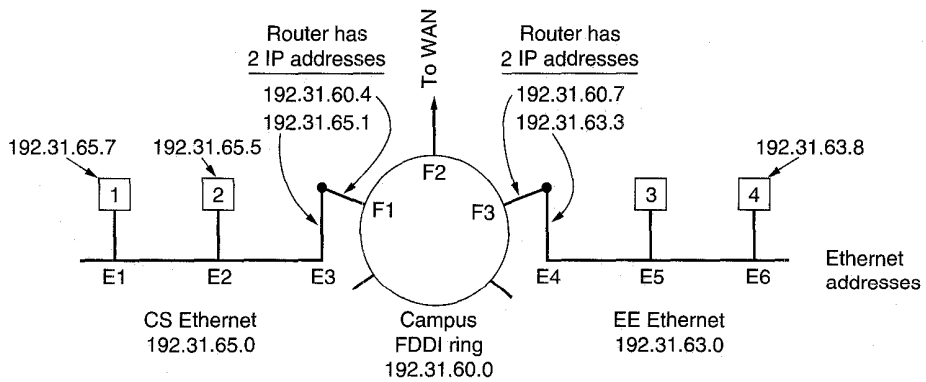


Fig. 5-51. Three interconnected class C networks: two Ethernets and an FDDI ring.

Let us start out by seeing how a user on host 1 sends a packet to a user on host 2. Let us assume the sender knows the name of the intended receiver, possibly something like *mary@eagle.cs.uni.edu*. The first step is to find the IP address for host 2, known as *eagle.cs.uni.edu*. This lookup is performed by the Domain Name System, which we will study in Chap. 7. For the moment, we will just assume that DNS returns the IP address for host 2 (192.31.65.5).

The upper layer software on host 1 now builds a packet with 192.31.65.5 in the *Destination address* field and gives it to the IP software to transmit. The IP software can look at the address and see that the destination is on its own network, but it needs a way to find the destination's Ethernet address. One solution is to have a configuration file somewhere in the system that maps IP addresses onto

Ethernet addresses. This solution is certainly possible, but for organizations with thousands of machines, keeping these files up to date is an error-prone, time-consuming job.

A better solution is for host 1 to output a broadcast packet onto the Ethernet asking: "Who owns IP address 192.31.65.5?" The broadcast will arrive at every machine on Ethernet 192.31.65.0, and each one will check its IP address. Host 2 alone will respond with its Ethernet address (*E2*). In this way host 1 learns that IP address 192.31.65.5 is on the host with Ethernet address *E2*. The protocol for asking this question and getting the reply is called **ARP (Address Resolution Protocol)**. Almost every machine on the Internet runs it. It is defined in RFC 826.

The advantage of using ARP over configuration files is the simplicity. The system manager does not have to do much except assign each machine an IP address and decide about subnet masks. ARP does the rest.

At this point, the IP software on host 1 builds an Ethernet frame addressed to *E2*, puts the IP packet (addressed to 192.31.65.5) in the payload field, and dumps it onto the Ethernet. The Ethernet board of host 2 detects this frame, recognizes it as a frame for itself, scoops it up, and causes an interrupt. The Ethernet driver extracts the IP packet from the payload and passes it to the IP software, which sees that it is correctly addressed, and processes it.

Various optimizations are possible to make ARP more efficient. To start with, once a machine has run ARP, it caches the result in case it needs to contact the same machine shortly. Next time it will find the mapping in its own cache, thus eliminating the need for a second broadcast. In many cases host 2 will need to send back a reply, forcing it, too, to run ARP to determine the sender's Ethernet address. This ARP broadcast can be avoided by having host 1 include its IP to Ethernet mapping in the ARP packet. When ARP broadcast arrives at host 2, the pair (192.31.65.7, E1) is entered into host 2's ARP cache for future use. In fact, all machines on the Ethernet can enter this mapping into their ARP caches.

Yet another optimization is to have every machine broadcast its mapping when it boots. This broadcast is generally done in the form of an ARP looking for its own IP address. There should not be a response, but a side effect of the broadcast is to make any entry in everyone's ARP cache. If a response does arrive, two machines have been assigned the same IP address. The new one should inform the system manager and not boot.

To allow mappings to change, for example, when an Ethernet board breaks and is replaced with a new one (and thus a new Ethernet address), entries in the ARP cache should time out after a few minutes.

Now let us look at Fig. 5-51 again, only this time host 1 wants to send a packet to host 6 (192.31.63.8). Using ARP will fail because host 4 will not see the broadcast (routers do not forward Ethernet-level broadcasts). There are two solutions. First, the CS router could be configured to respond to ARP requests for network 192.31.63.0 (and possibly other local networks). In this case, host 1 will make an ARP cache entry of (192.31.63.8, E3) and happily send all traffic for host

4 to the local router. This solution is called **proxy ARP**. The second solution is to have host 1 immediately see that the destination is on a remote network and just send all such traffic to a default Ethernet address that handles all remote traffic, in this case *E3*. This solution does not require having the CS router know which remote networks it is serving.

Either way, what happens is that host 1 packs the IP packet into the payload field of an Ethernet frame addressed to *E3*. When the CS router gets the Ethernet frame, it removes the IP packet from the payload field and looks up the IP address in its routing tables. It discovers that packets for network 192.31.63.0 are supposed to go to router 192.31.60.7. If it does not already know the FDDI address of 192.31.60.7, it broadcasts an ARP packet onto the ring and learns that its ring address is *F3*. It then inserts the packet into the payload field of an FDDI frame addressed to *F3* and puts it on the ring.

At the EE router, the FDDI driver removes the packet from the payload field and gives it to the IP software, which sees that it needs to send the packet to 192.31.63.8. If this IP address is not in its ARP cache, it broadcasts an ARP request on the EE Ethernet and learns that the destination address is *E6* so it builds an Ethernet frame addressed to *E6*, puts the packet in the payload field, and sends it over the Ethernet. When the Ethernet frame arrives at host 4, the packet is extracted from the frame and passed to the IP software for processing.

Going from host 1 to a distant network over a WAN works essentially the same way, except that this time the CS router's tables tell it to use the WAN router whose FDDI address is *F2*.

The Reverse Address Resolution Protocol

ARP solves the problem of finding out which Ethernet address corresponds to a given IP address. Sometimes the reverse problem has to solved: Given an Ethernet address, what is the corresponding IP address? In particular, this problem occurs when booting a diskless workstation. Such a machine will normally get the binary image of its operating system from a remote file server. But how does it learn its IP address?

The solution is to use the **RARP (Reverse Address Resolution Protocol)** (defined in RFC 903). This protocol allows a newly-booted workstation to broadcast its Ethernet address and say: "My 48-bit Ethernet address is 14.04.05.18.01.25. Does anyone out there know my IP address?" The RARP server sees this request, looks up the Ethernet address in its configuration files, and sends back the corresponding IP address.

Using RARP is better than embedding an IP address in the memory image because it allows the same image to be used on all machines. If the IP address were buried inside the image, each workstation would need its own image.

A disadvantage of RARP is that it uses a destination address of all 1s (limited broadcasting) to reach the RARP server. However, such broadcasts are not

forwarded by routers, so a RARP server is needed on each network. To get around this problem, an alternative bootstrap protocol called **BOOTP** has been invented (see RFCs 951, 1048, and 1084). Unlike RARP, it uses UDP messages, which are forwarded over routers. It also provides a diskless workstation with additional information, including the IP address of the file server holding the memory image, the IP address of the default router, and the subnet mask to use. BOOTP is described in RFC 951.

5.5.5. The Interior Gateway Routing Protocol: OSPF

As we mentioned earlier, the Internet is made up of a large number of autonomous systems. Each AS is operated by a different organization and can use its own routing algorithm inside. For example, the internal networks of companies *X*, *Y*, and *Z* would usually be seen as three ASes if all three were on the Internet. All three may use different routing algorithms internally. Nevertheless, having standards, even for internal routing, simplifies the implementation at the boundaries between ASes and allows reuse of code. In this section we will study routing within an AS. In the next one, we will look at routing between ASes. A routing algorithm within an AS is called an **interior gateway protocol**; an algorithm for routing between ASes is called an **exterior gateway protocol**.

The original Internet interior gateway protocol was a distance vector protocol (RIP) based on the Bellman-Ford algorithm. It worked well in small systems, but less well as ASes got larger. It also suffered from the count-to-infinity problem and generally slow convergence, so it was replaced in May 1979 by a link state protocol. In 1988, the Internet Engineering Task Force began work on a successor. That successor, called **OSPF** (**Open Shortest Path First**) became a standard in 1990. Many router vendors are now supporting it, and it will become the main interior gateway protocol in the near future. Below we will give a sketch of how OSPF works. For the complete story, see RFC 1247.

Given the long experience with other routing protocols, the group designing the new protocol had a long list of requirements that had to be met. First, the algorithm had to be published in the open literature, hence the "O" in OSPF. A proprietary solution owned by one company would not do. Second, the new protocol had to support a variety of distance metrics, including physical distance, delay, and so on. Third, it had to be a dynamic algorithm, one that adapted to changes in the topology automatically and quickly.

Fourth, and new for OSPF, it had to support routing based on type of service. The new protocol had to be able to route real-time traffic one way and other traffic a different way. The IP protocol has a *Type of Service* field, but no existing routing protocol used it.

Fifth, and related to the above, the new protocol had to do load balancing, splitting the load over multiple lines. Most previous protocols sent all packets

over the best route. The second-best route was not used at all. In many cases, splitting the load over multiple lines gives better performance.

Sixth, support for hierarchical systems was needed. By 1988, the Internet had grown so large that no router could be expected to know the entire topology. The new routing protocol had to be designed so that no router would have to.

Seventh, some modicum of security was required to prevent fun-loving students from spoofing routers by sending them false routing information. Finally, provision was needed for dealing with routers that were connected to the Internet via a tunnel. Previous protocols did not handle this well.

OSPF supports three kinds of connections and networks:

1. Point-to-point lines between exactly two routers.
2. Multiaccess networks with broadcasting (e.g., most LANs).
3. Multiaccess networks without broadcasting (e.g., most packet-switched WANs).

A **multiaccess** network is one that can have multiple routers on it, each of which can directly communicate with all the others. All LANs and WANs have this property. Figure 5-52(a) shows an AS containing all three kinds of networks. Note that hosts do not generally play a role in OSPF.

OSPF works by abstracting the collection of actual networks, routers, and lines into a directed graph in which each arc is assigned a cost (distance, delay, etc.). It then computes the shortest path based on the weights on the arcs. A serial connection between two routers is represented by a pair of arcs, one in each direction. Their weights may be different. A multiaccess network is represented by a node for the network itself plus a node for each router. The arcs from the network node to the routers have weight 0 and are omitted from the graph.

Figure 5-52(b) shows the graph representation of the network of Fig. 5-52(a). What OSPF fundamentally does is represent the actual network as a graph like this and then compute the shortest path from every router to every other router.

Many of the ASes in the Internet are themselves large and nontrivial to manage. OSPF allows them to be divided up into numbered **areas**, where an area is a network or a set of contiguous networks. Areas do not overlap but need not be exhaustive, that is, some routers may belong to no area. An area is a generalization of a subnet. Outside an area, its topology and details are not visible.

Every AS has a **backbone** area, called area 0. All areas are connected to the backbone, possibly by tunnels, so it is possible to go from any area in the AS to any other area in the AS via the backbone. A tunnel is represented in the graph as an arc and has a cost. Each router that is connected to two or more areas is part of the backbone. As with other areas, the topology of the backbone is not visible outside the backbone.

Within an area, each router has the same link state database and runs the same shortest path algorithm. Its main job is to calculate the shortest path from itself to

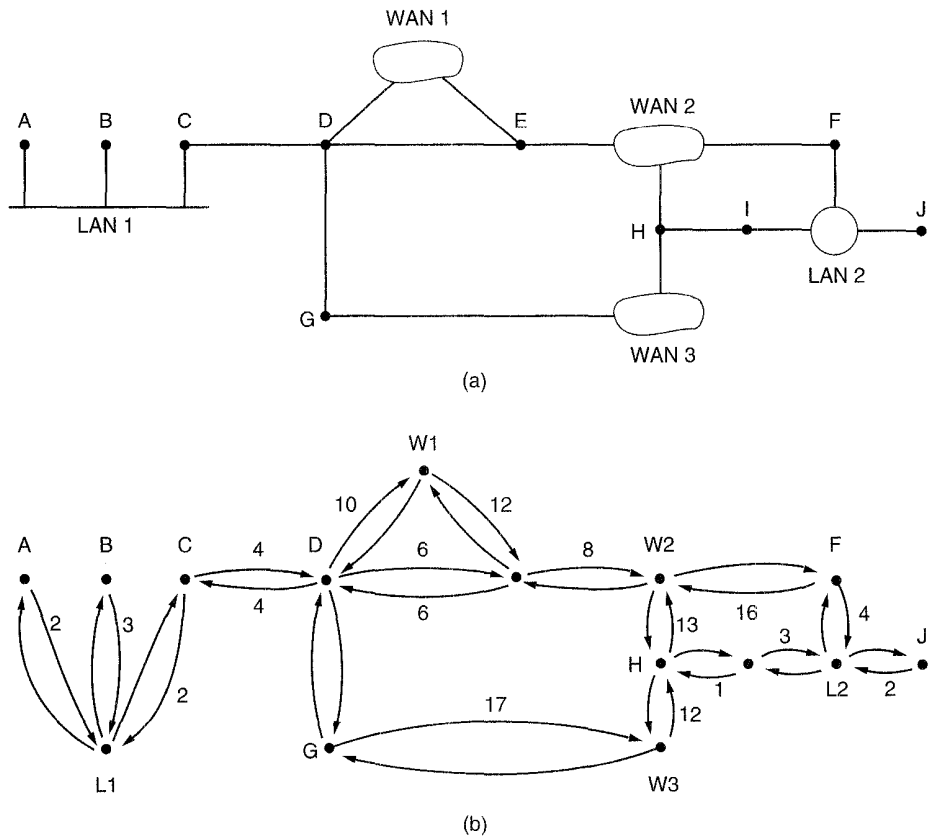


Fig. 5-52. (a) An autonomous system. (b) A graph representation of (a).

every other router in the area, including the router that is connected to the backbone, of which there must be at least one. A router that connects to two areas needs the databases for both areas and must run the shortest path algorithm for each one separately.

The way OSPF handles type of service routing is to have multiple graphs, one labeled with the costs when delay is the metric, one labeled with the costs when throughput is the metric, and one labeled with the costs when reliability is the metric. Although this triples the computation needed, it allows separate routes for optimizing delay, throughput, and reliability.

During normal operation, three kinds of routes may be needed: intra-area, interarea, and interAS. Intra-area routes are the easiest, since the source router already knows the shortest path to the destination router. Interarea routing always proceeds in three steps: go from the source to the backbone; go across the backbone to the destination area; go to the destination. This algorithm forces a star

configuration on OSPF with the backbone being the hub and the other areas being spokes. Packets are routed from source to destination "as is." They are not encapsulated or tunneled, unless going to an area whose only connection to the backbone is a tunnel. Figure 5-53 shows part of the Internet with ASes and areas.

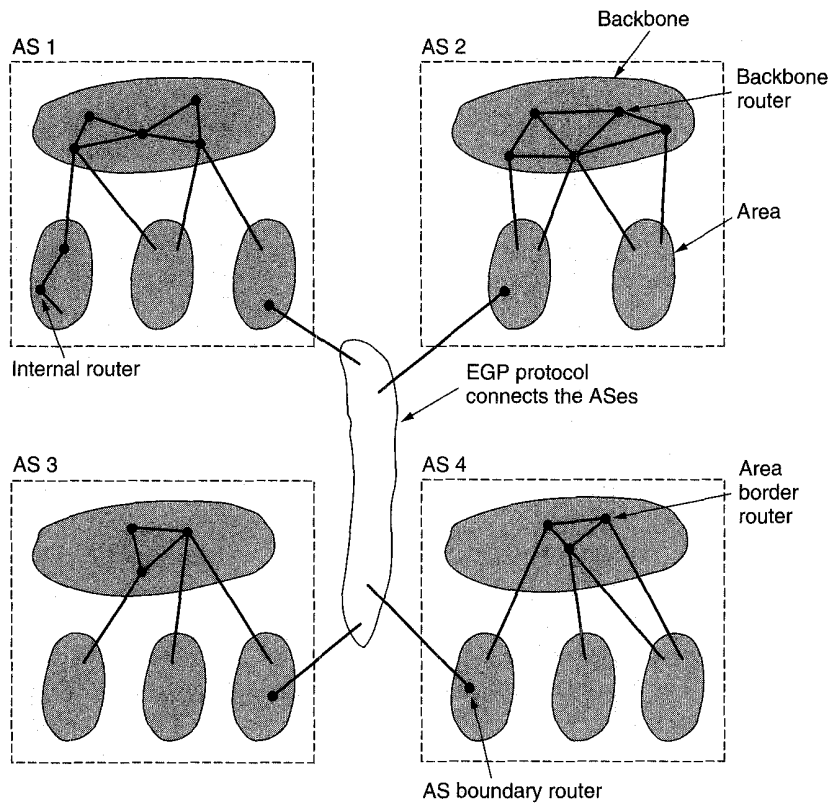


Fig. 5-53. The relation between ASes, backbones, and areas in OSPF.

OSPF distinguishes four classes of routers:

1. Internal routers are wholly within one area.
2. Area border routers connect two or more areas.
3. Backbone routers are on the backbone.
4. AS boundary routers talk to routers in other ASes.

These classes are allowed to overlap. For example, all the border routers are automatically part of the backbone. In addition, a router that is in the backbone

but not part of any other area is also an internal router. Examples of all four classes of routers are illustrated in Fig. 5-53.

When a router boots, it sends HELLO messages on all of its point-to-point lines and multicasts them on LANs to the group consisting of all the other routers. On WANs, it needs some configuration information to know who to contact. From the responses, each router learns who its neighbors are.

OSPF works by exchanging information between **adjacent** routers, which is not the same as between neighboring routers. In particular, it is inefficient to have every router on a LAN talk to every other router on the LAN. To avoid this situation, one router is elected as the **designated router**. It is said to be adjacent to all the other routers, and exchanges information with them. Neighboring routers that are not adjacent do not exchange information with each other. A backup designated router is always kept up to date to ease the transition should the primary designated router crash.

During normal operation, each router periodically floods LINK STATE UPDATE messages to each of its adjacent routers. This message gives its state and provides the costs used in the topological database. The flooding messages are acknowledged, to make them reliable. Each message has a sequence number, so a router can see whether an incoming LINK STATE UPDATE is older or newer than what it currently has. Routers also send these messages when a line goes up or down or its cost changes.

DATABASE DESCRIPTION messages give the sequence numbers of all the link state entries currently held by the sender. By comparing its own values with those of the sender, the receiver can determine who has the most recent values. These messages are used when a line is brought up.

Either partner can request link state information from the other one using LINK STATE REQUEST messages. The net result of this algorithm is that each pair of adjacent routers checks to see who has the most recent data, and new information is spread throughout the area this way. All these messages are sent as raw IP packets. The five kinds of messages are summarized in Fig. 5-54.

Message type	Description
Hello	Used to discover who the neighbors are
Link state update	Provides the sender's costs to its neighbors
Link state ack	Acknowledges link state update
Database description	Announces which updates the sender has
Link state request	Requests information from the partner

Fig. 5-54. The five types of OSPF messages.

Finally, we can put all the pieces together. Using flooding, each router informs all the other routers in its area of its neighbors and costs. This

information allows each router to construct the graph for its area(s) and compute the shortest path. The backbone area does this too. In addition, the backbone routers accept information from the area border routers in order to compute the best route from each backbone router to every other router. This information is propagated back to the area border routers, which advertise it within their areas. Using this information, a router about to send an interarea packet can select the best exit router to the backbone.

5.5.6. The Exterior Gateway Routing Protocol: BGP

Within a single AS, the recommended routing protocol on the Internet is OSPF (although it is certainly not the only one in use). Between ASes, a different protocol, **BGP** (**Border Gateway Protocol**), is used. A different protocol is needed between ASes because the goals of an interior gateway protocol and an exterior gateway protocol are not the same. All an interior gateway protocol has to do is move packets as efficiently as possible from the source to the destination. It does not have to worry about politics.

Exterior gateway protocol routers have to worry about politics a great deal. For example, a corporate AS might want the ability to send packets to any Internet site and receive packets from any Internet site. However, it might be unwilling to carry transit packets originating in a foreign AS and ending in a different foreign AS, even if its own AS was on the shortest path between the two foreign ASes ("That's their problem, not ours"). On the other hand, it might be willing to carry transit traffic for its neighbors, or even for specific other ASes that paid it for this service. Telephone companies, for example, might be happy to act as a carrier for their customers, but not for others. Exterior gateway protocols in general, and BGP in particular, have been designed to allow many kinds of routing policies to be enforced in the interAS traffic.

Typical policies involve political, security, or economic considerations. A few examples of routing constraints are

1. No transit traffic through certain ASes.
2. Never put Iraq on a route starting at the Pentagon.
3. Do not use the United States to get from British Columbia to Ontario.
4. Only transit Albania if there is no alternative to the destination.
5. Traffic starting or ending at IBM® should not transit Microsoft®.

Policies are manually configured into each BGP router. They are not part of the protocol itself.

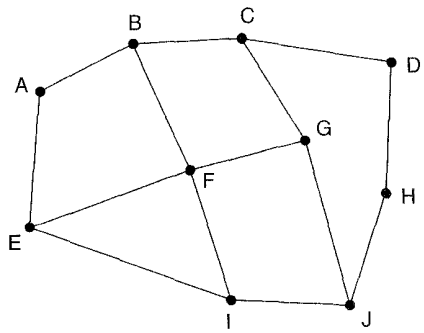
From the point of view of a BGP router, the world consists of other BGP routers and the lines connecting them. Two BGP routers are considered connected if they share a common network. Given BGP's special interest in transit

traffic, networks are grouped into one of three categories. The first category is the **stub networks**, which have only one connection to the BGP graph. These cannot be used for transit traffic because there is no one on the other side. Then come the **multiconnected networks**. These could be used for transit traffic, except that they refuse. Finally, there are the **transit networks**, such as backbones, which are willing to handle third-party packets, possibly with some restrictions.

Pairs of BGP routers communicate with each other by establishing TCP connections. Operating this way provides reliable communication and hides all the details of the network being passed through.

BGP is fundamentally a distance vector protocol, but quite different from most others such as RIP. Instead of maintaining just the cost to each destination, each BGP router keeps track of the exact path used. Similarly, instead of periodically giving each neighbor its estimated cost to each possible destination, each BGP router tells its neighbors the exact path it is using.

As an example, consider the BGP routers shown in Fig. 5-55(a). In particular, consider *F*'s routing table. Suppose that it uses the path *FGCD* to get to *D*. When the neighbors give it routing information, they provide their complete paths, as shown in Fig. 5-55(b) (for simplicity, only destination *D* is shown here).



Information F receives from its neighbors about D

From B: "I use BCD"
From G: "I use GCD"
From I: "I use IFGCD"
From E: "I use EFGCD"

(b)

Fig. 5-55. (a) A set of BGP routers. (b) Information sent to *F*.

After all the paths come in from the neighbors, *F* examines them to see which is the best. It quickly discards the paths from *I* and *E*, since these paths pass through *F* itself. The choice is then between using *B* and *G*. Every BGP router contains a module that examines routes to a given destination and scores them, returning a number for the "distance" to that destination for each route. Any route violating a policy constraint automatically gets a score of infinity. The router then adopts the route with the shortest distance. The scoring function is not part of the BGP protocol and can be any function the system managers want.

BGP easily solves the count-to-infinity problem that plagues other distance vector routing algorithms. For example, suppose *G* crashes or the line *FG* goes

down. *F* then receives routes from its three remaining neighbors. These routes are *BCD*, *IFGCD*, and *EFGCD*. It can immediately see that the two latter routes are pointless, since they pass through *F* itself, so it chooses *FBCD* as its new route. Other distance vector algorithms often make the wrong choice because they cannot tell which of their neighbors have independent routes to the destination, and which do not. The current definition of BGP is in RFC 1654. Additional useful information can be found in RFC 1268.

5.5.7. Internet Multicasting

Normal IP communication is between one sender and one receiver. However, for some applications it is useful for a process to be able to send to a large number of receivers simultaneously. Examples are updating replicated, distributed databases, transmitting stock quotes to multiple brokers, and handling digital conference (i.e., multiparty) telephone calls.

IP supports multicasting, using class D addresses. Each class D address identifies a group of hosts. Twenty-eight bits are available for identifying groups, so over 250 million groups can exist at the same time. When a process sends a packet to a class D address, a best-efforts attempt is made to deliver it to all the members of the group addressed, but no guarantees are given. Some members may not get the packet.

Two kinds of group addresses are supported: permanent addresses and temporary ones. A permanent group is always there and does not have to be set up. Each permanent group has a permanent group address. Some examples of permanent group addresses are

224.0.0.1 All systems on a LAN
224.0.0.2 All routers on a LAN
224.0.0.5 All OSPF routers on a LAN
224.0.0.6 All designated OSPF routers on a LAN

Temporary groups must be created before they can be used. A process can ask its host to join a specific group. It can also ask its host to leave the group. When the last process on a host leaves a group, that group is no longer present on the host. Each host keeps track of which groups its processes currently belong to.

Multicasting is implemented by special multicast routers, which may or may not be colocated with the standard routers. About once a minute, each multicast router sends a hardware (i.e., data link layer) multicast to the hosts on its LAN (address 224.0.0.1) asking them to report back on the groups their processes currently belong to. Each host sends back responses for all the class D addresses it is interested in.

These query and response packets use a protocol called **IGMP** (**Internet Group Management Protocol**), which is vaguely analogous to ICMP. It has only two kinds of packets: query and response, each with a simple fixed format

containing some control information in the first word of the payload field and a class D address in the second word. It is described in RFC 1112.

Multicast routing is done using spanning trees. Each multicast router exchanges information with its neighbors using a modified distance vector protocol in order for each one to construct a spanning tree per group covering all group members. Various optimizations are used to prune the tree to eliminate routers and networks not interested in particular groups. The protocol makes heavy use of tunneling to avoid bothering nodes not in a spanning tree.

5.5.8. Mobile IP

Many users of the Internet have portable computers and want to stay connected to the Internet when they visit a distant Internet site and even on the road in between. Unfortunately, the IP addressing system makes working far from home easier said than done. In this section we will examine the problem and the solution. A more detailed description is given in (Johnson, 1995).

The real villain is the addressing scheme itself. Every IP address contains three fields: the class, the network number, and the host number. For example, consider the machine with IP address 160.80.40.20. The 160.80 gives the class (B) and network number (8272); the 40.20 is the host number (10260). Routers all over the world have routing tables telling which line to use to get to network 160.80. Whenever a packet comes in with a destination IP address of the form 160.80.xxx.yyy, it goes out on that line.

If all of a sudden, the machine with that address is carted off to some distant site, the packets for it will continue to be routed to its home LAN (or router). The owner will no longer get email, and so on. Giving the machine a new IP address corresponding to its new location is unattractive because large numbers of people, programs, and databases would have to be informed of the change.

Another approach is to have the routers use complete IP addresses for routing, instead of just the class and network. However, this strategy would require each router to have millions of table entries, at astronomical cost to the Internet.

When people began demanding the ability to have mobile hosts, the IETF set up a Working Group to find a solution. The Working Group quickly formulated a number of goals considered desirable in any solution. The major ones were

1. Each mobile host must be able to use its home IP address anywhere.
2. Software changes to the fixed hosts were not permitted.
3. Changes to the router software and tables were not permitted.
4. Most packets for mobile hosts should not make detours on the way.
5. No overhead should be incurred when a mobile host is at home.

The solution chosen was the one described in Sec. 5.2.8. To review it briefly, every site that wants to allow its users to roam has to create a home agent. Every

site that wants to allow visitors has to create a foreign agent. When a mobile host shows up at a foreign site, it contacts the foreign host there and registers. The foreign host then contacts the user's home agent and gives it a **care-of address**, normally the foreign agent's own IP address.

When a packet arrives at the user's home LAN, it comes in at some router attached to the LAN. The router then tries to locate the host in the usual way, by broadcasting an ARP packet asking, for example: "What is the Ethernet address of 160.80.40.20?" The home agent responds to this query by giving its own Ethernet address. The router then sends packets for 160.80.40.20 to the home agent. It, in turn, tunnels them to the care-of address by encapsulating them in the payload field of an IP packet addressed to the foreign agent. The foreign agent then decapsulates and delivers them to the data link address of the mobile host. In addition, the home agent gives the care-of address to the sender, so future packets can be tunnelled directly to the foreign agent. This solution meets all the requirements stated above.

One small detail is probably worth mentioning. At the time the mobile host moves, the router probably has its (soon-to-be-invalid) Ethernet address cached. To replace that Ethernet address with the home agent's, a trick called **gratuitious ARP** is used. This is a special, unsolicited message to the router that causes it to replace a specific cache entry, in this case, that of the mobile host about to leave. When the mobile host returns later, the same trick is used to update the router's cache again.

Nothing in the design prevents a mobile host from being its own foreign agent, but that approach only works if the mobile host (in its capacity as foreign agent) is logically connected to the Internet at its current site. Also, it must be able to acquire a (temporary) care-of IP address to use. That IP address must belong to the LAN to which it is currently attached.

The IETF solution for mobile hosts solves a number of other problems not mentioned so far. For example, how are agents located? The solution is for each agent to periodically broadcast its address and the type of services it is willing to provide (e.g., home, foreign, or both). When a mobile host arrives somewhere, it can just listen for these broadcasts, called **advertisements**. Alternatively, it can broadcast a packet announcing its arrival and hope that the local foreign agent responds to it.

Another problem that had to be solved is what to do about impolite mobile hosts that leave without saying goodbye. The solution is to make registration valid only for a fixed time interval. If it is not refreshed periodically, it times out, so the foreign host can clear its tables.

Yet another issue is security. When a home agent gets a message asking it to please forward all of Nora's packets to some IP address, it had better not comply unless it is convinced that Nora is the source of this request, and not somebody trying to impersonate her. Cryptographic authentication protocols are used for this purpose. We will study such protocols in Chap. 7.

A final point addressed by the Working Group relates to levels of mobility. Imagine an airplane with an on-board Ethernet used by the navigation and avionics computers. On this Ethernet is a standard router that talks to the wired Internet on the ground over a radio link. One fine day, some clever marketing executive gets the idea to install Ethernet connectors in all the arm rests so passengers with mobile computers can also plug in.

Now we have two levels of mobility: the aircraft's own computers, which are stationary with respect to the Ethernet, and the passengers' computers, which are mobile with respect to it. In addition, the on-board router is mobile with respect to routers on the ground. Being mobile with respect to a system that is itself mobile can be handled using recursive tunneling.

5.5.9. CIDR—Classless InterDomain Routing

IP has been in heavy use for over a decade. It has worked extremely well, as demonstrated by the exponential growth of the Internet. Unfortunately, IP is rapidly becoming a victim of its own popularity: it is running out of addresses. This looming disaster has sparked a great deal of discussion and controversy within the Internet community about what to do about it. In this section we will describe both the problem and several proposed solutions. A more complete description is given in (Huitema, 1996).

Back in 1987, a few visionaries predicted that some day the Internet might grow to 100,000 networks. Most experts pooh-poohed this as being decades in the future, if ever. The 100,000th network was connected in 1996. The problem, simply stated, is that the Internet is rapidly running out of IP addresses. In principle, over 2 billion addresses exist, but the practice of organizing the address space by classes (see Fig. 5-47), wastes millions of them. In particular, the real villain is the class B network. For most organizations, a class A network, with 16 million addresses is too big, and a class C network, with 256 addresses is too small. A class B network, with 65,536, is just right. In Internet folklore, this situation is known as the **three bears problem** (as in *Goldilocks and the Three Bears*).

In reality, a class B address is far too large for most organizations. Studies have shown that more than half of all class B networks have fewer than 50 hosts. A class C network would have done the job, but no doubt every organization that asked for a class B address thought that one day it would outgrow the 8-bit host field. In retrospect, it might have been better to have had class C networks use 10 bits instead of eight for the host number, allowing 1022 hosts per network. Had this been the case, most organizations would have probably settled for a class C network, and there would have been half a million of them (versus only 16,384 class B networks).

However, then another problem would have emerged more quickly: the routing table explosion. From the point of view of the routers, the IP address space is

a two-level hierarchy, with network numbers and host numbers. Routers do not have to know about all the hosts, but they do have to know about all the networks. If half a million class C networks were in use, every router in the entire Internet would need a table with half a million entries, one per network, telling which line to use to get to that network, as well as other information.

The actual physical storage of half a million entry tables is probably doable, although expensive for critical routers that keep the tables in static RAM on I/O boards. A more serious problem is that the complexity of various algorithms relating to management of the tables grows faster than linear. Worse yet, much of the existing router software and firmware was designed at a time when the Internet had 1000 connected networks and 10,000 networks seemed decades away. Design choices made then often are far from optimal now.

In addition, various routing algorithms require each router to transmit its tables periodically. The larger the tables, the more likely some parts will get lost underway, leading to incomplete data at the other end and possibly routing instabilities.

The routing table problem could have been solved by going to a deeper hierarchy. For example, having each IP address contain a country, state, city, network, and host field might work. Then each router would only need to know how to get to each country, the states or provinces in its own country, the cities in its state or province, and the networks in its city. Unfortunately, this solution would require considerably more than 32 bits for IP addresses and would use addresses inefficiently (Liechtenstein would have as many bits as the United States).

In short, most solutions solve one problem but create a new one. One solution that is now being implemented and which will give the Internet a bit of extra breathing room is **CIDR (Classless InterDomain Routing)**. The basic idea behind CIDR, which is described in RFC 1519, is to allocate the remaining class C networks, of which there are almost two million, in variable-sized blocks. If a site needs, say, 2000 addresses, it is given a block of 2048 addresses (eight contiguous class C networks), and not a full class B address. Similarly, a site needing 8000 addresses gets 8192 addresses (32 contiguous class C networks).

In addition to using blocks of contiguous class C networks as units, the allocation rules for the class C addresses were also changed in RFC 1519. The world was partitioned into four zones, and each one given a portion of the class C address space. The allocation was as follows:

Addresses 194.0.0.0 to 195.255.255.255 are for Europe
Addresses 198.0.0.0 to 199.255.255.255 are for North America
Addresses 200.0.0.0 to 201.255.255.255 are for Central and South America
Addresses 202.0.0.0 to 203.255.255.255 are for Asia and the Pacific

In this way, each region was given about 32 million addresses to allocate, with another 320 million class C addresses from 204.0.0.0 through 223.255.255.255

held in reserve for the future. The advantage of this allocation is that now any router outside of Europe that gets a packet addressed to 194.xx.yy.zz or 195.xx.yy.zz can just send it to its standard European gateway. In effect 32 million addresses have now been compressed into one routing table entry. Similarly for the other regions.

Of course, once a 194.xx.yy.zz packet gets to Europe, more detailed routing tables are needed. One possibility is to have 131,072 entries for networks 194.0.0.xx through 195.255.255.xx, but this is precisely this routing table explosion that we are trying to avoid. Instead, each routing table entry is extended by giving it a 32-bit mask. When a packet comes in, its destination address is first extracted. Then (conceptually) the routing table is scanned entry by entry, masking the destination address and comparing it to the table entry looking for a match.

To make this comparison process clearer, let us consider an example. Suppose that Cambridge University needs 2048 addresses and is assigned the addresses 194.24.0.0 through 194.24.7.255, along with mask 255.255.248.0. Next, Oxford University asks for 4096 addresses. Since a block of 4096 addresses must lie on a 4096-byte boundary, they cannot be given addresses starting at 194.8.0.0. Instead they get 194.24.16.0 through 194.24.31.255 along with mask 255.255.240.0. Now the University of Edinburgh asks for 1024 addresses and is assigned addresses 194.24.8.0 through 194.24.11.255 and mask 255.255.252.0.

The routing tables all over Europe are now updated with three entries, each one containing a base address and a mask. These entries (in binary) are:

Address	Mask
11000010 00011000 00000000 00000000	11111111 11111111 11111000 00000000
11000010 00011000 00010000 00000000	11111111 11111111 11110000 00000000
11000010 00011000 00001000 00000000	11111111 11111111 11111100 00000000

Now consider what happens when a packet comes in addressed to 194.24.17.4, which in binary is

11000010 00011000 00010001 00000100

First it is Boolean ANDed with the Cambridge mask to get

11000010 00011000 00010000 00000000

This value does not match the Cambridge base address, so the original address is next ANDed with the Oxford mask to get

11000010 00011000 00010000 00000000

This value does match the Oxford mask, so the packet is sent to the Oxford router. In practice, the router entries are not tried sequentially; indexing tricks are used to speed up the search. Also, it is possible for two entries to match, in which case the one whose mask has the most 1 bits wins. Finally, the same idea can be applied to all addresses, not just the new class C addresses, so with CIDR, the old

class A, B, and C networks are no longer used for routing. This is why CIDR is called classless routing. CIDR is described in more detail in (Ford et al., 1993; and Huitema, 1995).

5.5.10. IPv6

While CIDR may buy a few more years' time, everyone realizes that the days of IP in its current form (IPv4) are numbered. In addition to these technical problems, there is another issue looming in the background. Up until recently, the Internet has been used largely by universities, high-tech industry, and the government (especially the Dept. of Defense). With the explosion of interest in the Internet starting in the mid 1990s, it is likely that in the next millenium, it will be used by a much larger group of people, especially people with different requirements. For one thing, millions of people with wireless portables may use it to keep in contact with their home bases. For another, with the impending convergence of the computer, communication, and entertainment industries, it may not be long before every television set in the world is an Internet node, producing a billion machines being used for video on demand. Under these circumstances, it became apparent that IP had to evolve and become more flexible.

Seeing these problems on the horizon, in 1990, IETF started work on a new version of IP, one which would never run out of addresses, would solve a variety of other problems, and be more flexible and efficient as well. Its major goals were to

1. Support billions of hosts, even with inefficient address space allocation.
2. Reduce the size of the routing tables.
3. Simplify the protocol, to allow routers to process packets faster.
4. Provide better security (authentication and privacy) than current IP.
5. Pay more attention to type of service, particularly for real-time data.
6. Aid multicasting by allowing scopes to be specified.
7. Make it possible for a host to roam without changing its address.
8. Allow the protocol to evolve in the future.
9. Permit the old and new protocols to coexist for years.

To find a protocol that met all these requirements, IETF issued a call for proposals and discussion in RFC 1550. Twenty-one responses were received, not all of them full proposals. By December 1992, seven serious proposals were on the table. They ranged from making minor patches to IP, to throwing it out altogether and replacing with a completely different protocol.

One proposal was to run TCP over CLNP, which, with its 160-bit addresses would have provided enough address space forever and would have unified two major network layer protocols. However, many people felt that this would have been an admission that something in the OSI world was actually done right, a statement considered Politically Incorrect in Internet circles. CLNP was patterned closely on IP, so the two are not really that different. In fact, the protocol ultimately chosen differs from IP far more than CLNP does. Another strike against CLNP was its poor support for service types, something required to transmit multimedia efficiently.

Three of the better proposals were published in *IEEE Network* (Deering, 1993; Francis, 1993; and Katz and Ford, 1993). After much discussion, revision, and jockeying for position, a modified combined version of the Deering and Francis proposals, by now called **SIPP** (**Simple Internet Protocol Plus**) was selected and given the designation **IPv6** (IPv5 was already in use for an experimental real-time stream protocol).

IPv6 meets the goals fairly well. It maintains the good features of IP, discards or deemphasizes the bad ones, and adds new ones where needed. In general, IPv6 is not compatible with IPv4, but it is compatible with all the other Internet protocols, including TCP, UDP, ICMP, IGMP, OSPF, BGP, and DNS, sometimes with small modifications being required (mostly to deal with longer addresses). The main features of IPv6 are discussed below. More information about it can be found in RFC 1883 through RFC 1887.

First and foremost, IPv6 has longer addresses than IPv4. They are 16 bytes long, which solves the problem that IPv6 was set out to solve: provide an effectively unlimited supply of Internet addresses. We will have more to say about addresses shortly.

The second major improvement of IPv6 is the simplification of the header. It contains only 7 fields (versus 13 in IPv4). This change allows routers to process packets faster and thus improve throughout. We will discuss the header shortly, too.

The third major improvement was better support for options. This change was essential with the new header because fields that previously were required are now optional. In addition, the way options are represented is different, making it simple for routers to skip over options not intended for them. This feature speeds up packet processing time.

A fourth area in which IPv6 represents a big advance is in security. IETF had its fill of newspaper stories about precocious 12-year-olds using their personal computers to break into banks and military bases all over the Internet. There was a strong feeling that something had to be done to improve security. Authentication and privacy are key features of the new IP.

Finally, more attention has been paid to type of service than in the past. IPv4 actually has an 8-bit field devoted to this matter, but with the expected growth in multimedia traffic in the future, much more is needed.

The Main IPv6 Header

The IPv6 header is shown in Fig. 5-56. The *Version* field is always 6 for IPv6 (and 4 for IPv4). During the transition period from IPv4, which will probably take a decade, routers will be able to examine this field to tell what kind of packet they have. As an aside, making this test wastes a few instructions in the critical path, so many implementations are likely to try to avoid it by using some field in the data link header to distinguish IPv4 packets from IPv6 packets. In this way, packets can be passed to the correct network layer handler directly. However, having the data link layer be aware of network packet types completely violates the design principle that each layer should not be aware of the meaning of the bits given to it from the layer above. The discussions between the "Do it right" and "Make it fast" camps will no doubt be lengthy and vigorous.

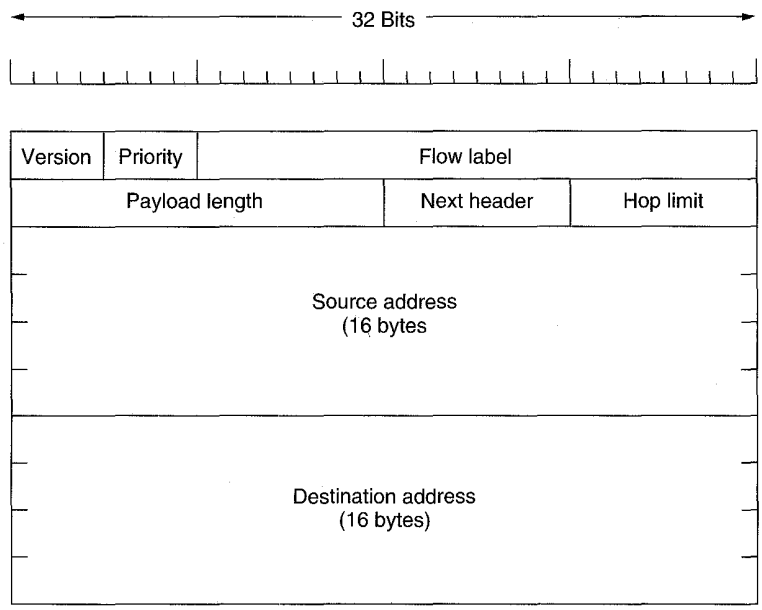


Fig. 5-56. The IPv6 fixed header (required).

The *Priority* field is used to distinguish between packets whose sources can be flow controlled and those that cannot. Values 0 through 7 are for transmissions that are capable of slowing down in the event of congestion. Values 8 through 15 are for real-time traffic whose sending rate is constant, even if all the packets are being lost. Audio and video fall into the latter category. This distinction allows routers to deal with packets better in the event of congestion. Within each group, lower-numbered packets are less important than higher-numbered ones. The IPv6 standard suggests, for example, to use 1 for news, 4 for FTP, and 6 for Telnet

connections, since delaying a news packet for a few seconds is not noticeable, but delaying a Telnet packet certainly is.

The *Flow label* field is still experimental but will be used to allow a source and destination to set up a pseudoconnection with particular properties and requirements. For example, a stream of packets from one process on a certain source host to a certain process on a certain destination host might have stringent delay requirements and thus need reserved bandwidth. The flow can be set up in advance and given an identifier. When a packet with a nonzero *Flow label* shows up, all the routers can look it up in internal tables to see what kind of special treatment it requires. In effect, flows are an attempt to have it both ways: the flexibility of a datagram subnet and the guarantees of a virtual circuit subnet.

Each flow is designated by the source address, destination address, and flow number, so many flows may be active at the same time between a given pair of IP addresses. Also, in this way, even if two flows coming from different hosts but with the same flow number pass through the same router, the router will be able to tell them apart using the source and destination addresses. It is expected that flow numbers will be chosen randomly, rather than assigned sequentially starting at 1, to make it easy for routers to hash them.

The *Payload length* field tells how many bytes follow the 40-byte header of Fig. 5-56. The name was changed from the IPv4 *Total length* field because the meaning was changed slightly: the 40 header bytes are no longer counted as part of the length as they used to be.

The *Next header* field lets the cat out of the bag. The reason the header could be simplified is that there can be additional (optional) extension headers. This field tells which of the (currently) six extension headers, if any, follows this one. If this header is the last IP header, the *Next header* field tells which transport protocol handler (e.g., TCP, UDP) to pass the packet to.

The *Hop limit* field is used to keep packets from living forever. It is, in practice, the same as the *Time to live* field in IPv4, namely, a field that is decremented on each hop. In theory, in IPv4 it was a time in seconds, but no router used it that way, so the name was changed to reflect the way it is actually used.

Next come the *Source address* and *Destination address* fields. Deering's original proposal, SIP, used 8-byte addresses, but during the review process many people felt that with 8-byte addresses IPv6 would run out of addresses within a few decades, whereas with 16-byte addresses it would never run out. Other people argued that 16 bytes was overkill, whereas still others favored using 20-byte addresses to be compatible with the OSI datagram protocol. Another faction wanted variable-sized addresses. After much discussion, it was decided that fixed-length 16-byte addresses were the best compromise.

The IPv6 address space is divided up as shown in Fig. 5-57. Addresses beginning with 80 zeros are reserved for IPv4 addresses. Two variants are supported, distinguished by the next 16 bits. These variants relate to how IPv6 packets will be tunneled over the existing IPv4 infrastructure.

Prefix (binary)	Usage	Fraction
0000 0000	Reserved (including IPv4)	1/256
0000 0001	Unassigned	1/256
0000 001	OSI NSAP addresses	1/128
0000 010	Novell NetWare IPX addresses	1/128
0000 011	Unassigned	1/128
0000 1	Unassigned	1/32
0001	Unassigned	1/16
001	Unassigned	1/8
010	Provider-based addresses	1/8
011	Unassigned	1/8
100	Geographic-based addresses	1/8
101	Unassigned	1/8
110	Unassigned	1/8
1110	Unassigned	1/16
1111 0	Unassigned	1/32
1111 10	Unassigned	1/64
1111 110	Unassigned	1/128
1111 1110 0	Unassigned	1/512
1111 1110 10	Link local use addresses	1/1024
1111 1110 11	Site local use addresses	1/1024
1111 1111	Multicast	1/256

Fig. 5-57. IPv6 addresses

The use of separate prefixes for provider-based and geographic-based addresses is a compromise between two different visions of the future of the Internet. Provider-based addresses make sense if you think that in the future there will be some number of companies providing Internet service to customers, analogous to AT&T, MCI, Sprint, British Telecom, and so on providing telephone service now. Each of these companies will be given some fraction of the address space. The first 5 bits following the 010 prefix are used to indicate which registry to look the provider up in. Currently three registries are operating, for North America, Europe, and Asia. Up to 29 new registries can be added later.

Each registry is free to divide up the remaining 15 bytes as it sees fit. It is expected that many of them will use a 3-byte provider number, giving about 16

million providers, in order to allow large companies to act as their own provider. Another possibility is to use 1 byte to indicate national providers and let them do further allocation. In this manner, additional levels of hierarchy can be introduced as needed.

The geographic model is the same as the current Internet, in which providers do not play a large role. In this way, IPv6 can handle both kinds of addresses.

The link and site local addresses have only a local significance. They can be reused at each organization without conflict. They cannot be propagated outside organizational boundaries, making them well suited to organizations that currently use firewalls to wall themselves off from the rest of the Internet.

Multicast addresses have a 4-bit flag field and a 4-bit scope field following the prefix, then a 112-bit group identifier. One of the flag bits distinguishes permanent from transient groups. The scope field allows a multicast to be limited to the current link, site, organization, or planet. These four scopes are spread out over the 16 values to allow new scopes to be added later. For example, the planetary scope is 14, so code 15 is available to allow future expansion of the Internet to other planets, solar systems, and galaxies.

In addition to supporting the standard unicast (point-to-point) and multicast addresses, IPv6 also supports a new kind of addressing: anycast. **Anycasting** is like multicasting in that the destination is a group of addresses, but instead of trying to deliver the packet to all of them, it tries to deliver it to just one, usually the nearest one. For example, when contacting a group of cooperating file servers, a client can use anycast to reach the nearest one, without having to know which one that is. Anycasting uses regular unicast addresses. It is up to the routing system to choose the lucky host that gets the packet.

A new notation has been devised for writing 16-byte addresses. They are written as eight groups of four hexadecimal digits with colons between the groups, like this:

8000:0000:0000:0000:0123:4567:89AB:CDEF

Since many addresses will have many zeros inside them, three optimizations have been authorized. First, leading zeros within a group can be omitted, so 0123 can be written as 123. Second, one or more groups of 16 zeros can be replaced by a pair of colons. Thus the above address now becomes

8000::123:4567:89AB:CDEF

Finally, IPv4 addresses can be written as a pair of colons and an old dotted decimal number, for example

::192.31.20.46

Perhaps it is unnecessary to be so explicit about it, but there are a lot of 16-byte addresses. Specifically, there are 2^{128} of them, which is approximately

3×10^{38}. If the entire earth, land and water, were covered with computers, IPv6 would allow 7×10^{23} IP addresses per square meter. Students of chemistry will notice that this number is larger than Avogadro's number. While it was not the intention to give every molecule on the surface of the earth its own IP address, we are not that far off.

In practice, the address space will not be used efficiently, just as the telephone number address space is not (the area code for Manhattan, 212, is nearly full, but that for Wyoming, 307, is nearly empty). In RFC 1715, Huitema calculated that using the allocation of telephone numbers as a guide, even in the most pessimistic scenario, there will still be well over 1000 IP addresses per square meter of the earth's surface (land and water). In any likely scenario, there will be trillions of them per square meter. In short, it seems unlikely that we will run out in the foreseeable future. It is also worth noting that only 28 percent of the address space has been allocated so far. The other 72 percent is available for future purposes not yet thought of.

It is instructive to compare the IPv4 header (Fig. 5-45) with the IPv6 header (Fig. 5-56) to see what has been left out in IPv6. The *IHL* field is gone because the IPv6 header has a fixed length. The *Protocol* field was taken out because the *Next header* field tells what follows the last IP header (e.g., a UDP or TCP segment).

All the fields relating to fragmentation were removed because IPv6 takes a different approach to fragmentation. To start with, all IPv6 conformant hosts and routers must support packets of 576 bytes. This rule makes fragmentation less likely to occur in the first place. In addition, when a host sends an IPv6 packet that is too large, instead of fragmenting it, the router that is unable to forward it sends back an error message. This message tells the host to break up all future packets to that destination. Having the host send packets that are the right size in the first place is ultimately much more efficient than having the routers fragment them on the fly.

Finally, the *Checksum* field is gone because calculating it greatly reduces performance. With the reliable networks now used, combined with the fact that the data link layer and transport layers normally have their own checksums, the value of yet another checksum was not worth the performance price it extracted. Removing all these features has resulted in a lean and mean network layer protocol. Thus the goal of IPv6—a fast, yet flexible, protocol with plenty of address space—has been met by this design.

Extension Headers

Nevertheless, some of the missing fields are occasionally still needed so IPv6 has introduced the concept of an (optional) **extension header**. These headers can be supplied to provide extra information, but encoded in an efficient way. Six

kinds of extension headers are defined at present, as listed in Fig. 5-58. Each one is optional, but if more than one is present, they must appear directly after the fixed header, and preferably in the order listed.

Extension header	Description
Hop-by-hop options	Miscellaneous information for routers
Routing	Full or partial route to follow
Fragmentation	Management of datagram fragments
Authentication	Verification of the sender's identity
Encrypted security payload	Information about the encrypted contents
Destination options	Additional information for the destination

Fig. 5-58. IPv6 extension headers.

Some of the headers have a fixed format; others contain a variable number of variable-length fields. For these, each item is encoded as a (Type, Length, Value) tuple. The *Type* is a 1-byte field telling which option this is. The *Type* values have been chosen so that the first 2 bits tell routers that do not know how to process the option what to do. The choices are: skip the option, discard the packet, discard the packet and send back an ICMP packet, and the same as the previous one, except do not send ICMP packets for multicast addresses (to prevent one bad multicast packet from generating millions of ICMP reports).

The *Length* is also a 1-byte field. It tells how long the value is (0 to 255 bytes). The *Value* is any information required, up to 255 bytes.

The hop-by-hop header is used for information that all routers along the path must examine. So far, one option has been defined: support of datagrams exceeding 64K. The format of this header is shown in Fig. 5-59.

Next header	0	194	0
Jumbo payload length			

Fig. 5-59. The hop-by-hop extension header for large datagrams (jumbograms).

As with all extension headers, this one starts out with a byte telling what kind of header comes next. This byte is followed by one telling how long the hop-by-hop header is in bytes, excluding the first 8 bytes, which are mandatory. The next 2 bytes indicate that this option defines the datagram size (code 194) as a 4-byte number. The last 4 bytes give the size of the datagram. Sizes less than 65,536 are not permitted and will result in the first router discarding the packet and sending back an ICMP error message. Datagrams using this header extension are called

jumbograms. The use of jumbograms is important for supercomputer applications that must transfer gigabytes of data efficiently across the Internet.

The routing header lists one or more routers that must be visited on the way to the destination. Both strict routing (the full path is supplied) and loose routing (only selected routers are supplied) are available, but they are combined. The format of the routing header is shown in Fig. 5-60.

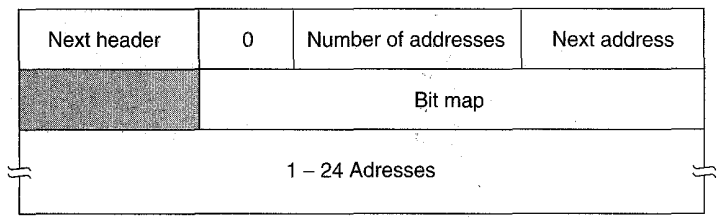


Fig. 5-60. The extension header for routing.

The first 4 bytes of the routing extension header contain four 1-byte integers: the next header type, the routing type (currently 0), the number of addresses present in this header (1 to 24), and the index of the next address to visit. The latter field starts at 0 and is incremented as each address is visited.

Then comes a reserved byte followed by a bit map with bits for each of the 24 potential IPv6 addresses following it. These bits tell whether each address must be visited directly after the one before it (strict source routing), or whether other routers may come in between (loose source routing).

The fragment header deals with fragmentation similarly to the way IPv4 does. The header holds the datagram identifier, fragment number, and a bit telling whether more fragments will follow. In IPv6, unlike in IPv4, only the source host can fragment a packet. Routers along the way may not do this. Although this change is a major philosophical break with the past, it simplifies the routers' work and makes routing go faster. As mentioned above, if a router is confronted with a packet that is too big, it discards the packet and sends an ICMP packet back to the source. This information allows the source host to fragment the packet into smaller pieces using this header and try again.

The authentication header provides a mechanism by which the receiver of a packet can be sure of who sent it. With IPv4, no such guarantee is present. The encrypted security payload makes it possible to encrypt the contents of a packet so that only the intended recipient can read it. These headers use cryptographic techniques to accomplish their missions. We will give brief descriptions below, but readers not already familiar with modern cryptography may not understand the full description now. They should come back after having read Chap. 7 (in particular, Sec. 7.1), which treats cryptographic protocols.

When a sender and receiver wish to communicate securely, they must first agree on one or more secret keys that only they know. How they do this is outside

the scope of IPv6. Each of these keys is assigned a unique 32-bit key number. The key numbers are global, so that if Alice is using key 4 to talk to Bob, she cannot also have a key 4 to talk to Carol. Associated with each key number are other parameters, such as key lifetime, and so on.

To send an authenticated message, the sender first constructs a packet consisting of all the IP headers and the payload and then replaces the fields that change underway (e.g., *Hop limit*) with zeros. The packet is then padded out with zeros to a multiple of 16 bytes. Similarly, the secret key to be used is also padded out with zeros to a multiple of 16 bytes. Now a cryptographic checksum is computed on the concatenation of the padded secret key, the padded packet, and the padded secret key again. Users may define their own cryptographic checksum algorithms, but cryptographically unsophisticated users should use the default algorithm, MD5.

Now we come to the role of the authentication header. Basically, it contains three parts. The first part consists of 4 bytes holding the next header number, the length of the authentication header, and 16 zero bits. Then comes the 32-bit key number. Finally, the MD5 (or other) checksum is included.

The receiver then uses the key number to find the secret key. The padded version of it is then prepended and appended to the padded payload, the variable header fields are zeroed out, and the checksum computed. If it agrees with the checksum included in the authentication header, the receiver can be sure that the packet came from the sender with whom the secret key is shared and also be sure that the packet was not tampered with underway. The properties of MD5 make it computationally infeasible for an intruder to forge the sender's identity or modify the packet in a way that escapes detection.

It is important to note that the payload of an authenticated packet is sent unencrypted. Any router along the way can read what it says. For many applications, secrecy is not really important, just authentication. For example, if a user instructs his bank to pay his telephone bill, there is probably no real need for secrecy, but there is a very real need for the bank to be absolutely sure it knows who sent the packet containing the payment order.

For packets that must be sent secretly, the encrypted security payload extension header is used. It starts out with a 32-bit key number, followed by the encrypted payload. The encryption algorithm is up to the sender and receiver, but DES in cipher block chaining mode is the default. When DES-CBC is used, the payload field starts out with the initialization vector (a multiple of 4 bytes), then the payload, then padding out to multiple of 8 bytes. If both encryption and authentication are desired, both headers are needed.

The destination options header is intended for fields that need only be interpreted at the destination host. In the initial version of IPv6, the only options defined are null options for padding this header out to a multiple of 8 bytes, so initially it will not be used. It was included to make sure that new routing and host software can handle it, in case someone thinks of a destination option some day.

Controversies

Given the open design process and the strongly-held opinions of many of the people involved, it should come as no surprise that many choices made for IPv6 were highly controversial. We will summarize a few of these below. For all the gory details, see (Huitema, 1996).

We have already mentioned the argument about the address length. The result was a compromise: 16-byte fixed-length addresses.

Another fight developed over the length of the *Hop limit* field. One camp felt strongly that limiting the maximum number of hops to 255 (implicit in using an 8-bit field) was a gross mistake. After all, paths of 32 hops are common now, and 10 years from now much longer paths may be common. These people argued that using a huge address size was farsighted but using a tiny hop count was shortsighted. In their view, the greatest sin a computer scientist can commit is to provide too few bits somewhere.

The response was that arguments could be made to increase every field, leading to a bloated header. Also, the function of the *Hop limit* field is to keep packets from wandering around for a long time and 65,535 hops is far too long. Finally, as the Internet grows, more and more long-distance links will be built, making it possible to get from any country to any other country in half a dozen hops at most. If it takes more than 125 hops to get from the source and destination to their respective international gateways, something is wrong with the national backbones. The 8-bitters won this one.

Another hot potato was the maximum packet size. The supercomputer community wanted packets in excess of 64 KB. When a supercomputer gets started transferring, it really means business and does not want to be interrupted every 64 KB. The argument against large packets is that if a 1-MB packet hits a 1.5-Mbps T1 line, that packet will tie the line up for over 5 seconds, producing a very noticeable delay for interactive users sharing the line. A compromise was reached here: normal packets are limited to 64 KB, but the hop-by-hop extension header can be used to permit jumbograms.

A third hot topic was removing the IPv4 checksum. Some people likened this move to removing the brakes from a car. Doing so makes the car lighter so it can go faster, but if an unexpected event happens, you have a problem.

The argument against checksums was that any application that really cares about data integrity has to have a transport layer checksum anyway, so having another one in IP (in addition to the data link layer checksum) is overkill. Furthermore, experience showed that computing the IP checksum was a major expense in IPv4. The antichecksum camp won this one, and IPv6 does not have a checksum.

Mobile hosts were also a point of contention. If a portable computer flies halfway around the world, can it continue operating at the destination with the same IPv6 address, or does it have to use a scheme with home agents and foreign

agents? Mobile hosts also introduce asymmetries into the routing system. It may well be the case that a small mobile computer can easily hear the powerful signal put out by a large stationary router, but the stationary router cannot hear the feeble signal put out by the mobile host. Consequently, some people wanted to build explicit support for mobile hosts into IPv6. That effort failed when no consensus could be found for any specific proposal.

Probably the biggest battle was about security. Everyone agreed it was needed. The war was about where and how. First where. The argument for putting it in the network layer is that it then becomes a standard service that all applications can use without any advance planning. The argument against it is that really secure applications generally want nothing less than end-to-end encryption, where the source application does the encryption and the destination application undoes it. With anything less, the user is at the mercy of potentially buggy network layer implementations over which he has no control. The response to this argument is that these applications can just refrain from using the IP security features and do the job themselves. The rejoinder to that is that the people who do not trust the network to do it right, do not want to have to pay the price of slow, bulky IP implementations that have this capability, even if it is disabled.

Another aspect of where to put security relates to the fact that many (but not all) countries have stringent export laws concerning cryptography. Some, notably France and Iraq, also greatly restrict its use domestically, so that people cannot have secrets from the police. As a result, any IP implementation that used a cryptographic system strong enough to be of much value could not be exported from the United States (and many other countries) to customers worldwide. Having to maintain two sets of software, one for domestic use and one for export, is something most computer vendors vigorously oppose.

One potential solution is for all vendors to move their cryptography shops to a country that does not regulate cryptography, such as Finland or Switzerland. Strong cryptographic software could be designed and manufactured there and then shipped legally to all countries except France and Iraq. The problem with this approach is that designing part of the router software in one country and part in another can lead to integration problems.

The final controversy concerning security relates to the choice of the default algorithms that all implementations must support. While MD5 was thought to be relatively secure, recent advances in cryptography may weaken it. No serious cryptographer believes that DES is secure against attacks by major governments, but it is probably good enough to foil even the most precocious 12-year-olds for the time being. The compromise was thus to mandate security in IPv6, use a state-of-the-art checksum algorithm for good authentication and a weakish algorithm for secrecy but give users the option of replacing these algorithms with their own.

One point on which there was no controversy is that no one expects the IPv4 Internet to be turned off on a Sunday morning and come back up as an IPv6

Internet Monday morning. Instead, isolated "islands" of IPv6 will be converted, initially communicating via tunnels. As the IPv6 islands grow, they will merge into bigger islands. Eventually, all the islands will merge, and the Internet will be fully converted. Given the massive investment in IPv4 routers currently deployed, the conversion process will probably take a decade. For this reason, an enormous amount of effort has gone into making sure that this transition will be as painless as possible.

5.6. THE NETWORK LAYER IN ATM NETWORKS

The layers of the ATM model (see Fig. 1-30) do not map onto the OSI layers especially well, which leads to ambiguities. The OSI data link layer deals with framing and transfer protocols between two machines on the same physical wire (or fiber). Data link layer protocols are single-hop protocols. They do not deal with end-to-end connections because switching and routing do not occur in the data link layer. About this there is no doubt.

The lowest layer that goes from source to destination, and thus involves routing and switching (i.e., is multihop), is the network layer. The ATM layer deals with moving cells from source to destination and definitely involves routing algorithms and protocols within the ATM switches. It also deals with global addressing. Thus functionally, the ATM layer performs the work expected of the network layer. The ATM layer is not guaranteed to be 100 percent reliable, but that is not required for a network layer protocol.

Also, the ATM layer resembles layer 3 of X.25, which everyone agrees is a network layer protocol. Depending on bit settings, the X.25 network layer protocol may or may not be reliable, but most implementations treat it as unreliable. Because the ATM layer has the functionality expected of the network layer, does not have the functionality expected of the data link layer, and is quite similar to existing network layer protocols, we will discuss the ATM layer in this chapter.

Confusion arises because many people in the ATM community regard the ATM layer as a data link layer, or when doing LAN emulation, even a physical layer. Many people in the Internet community also regard it as a data link layer because they want to put IP on top of it, and making the ATM layer a data link layer fits well with this idea. (Although following through with this line of reasoning, to the Internet community, *all* networks operate at the data link layer, no matter what their physical characteristics are.)

The only problem is that the ATM layer does not have the characteristics of a data link layer protocol: a single-hop protocol used by machines at the opposite ends of a wire, such as protocols 1 through 6 in Chap. 3. It has the characteristics of a network layer protocol: end-to-end virtual circuits, switching, and routing.

The author is reminded of an old riddle:

Q: How many legs would a mule have if you called the tail a leg?
A: Four. *Calling* the tail a leg does not *make* it a leg.

Suffice it to say, the reader is warned of controversy here, combined with a major dose of raw emotion.

The ATM layer is connection oriented, both in terms of the service it offers and the way it operates internally. The basic element of the ATM layer is the virtual circuit (officially called a **virtual channel**). A virtual circuit is normally a connection from one source to one destination, although multicast connections are also permitted. Virtual circuits are unidirectional, but a pair of circuits can be created at the same time. Both parts of the pair are addressed by the same identifier, so effectively a virtual circuit is full duplex. However, the channel capacity and other properties may be different in the two directions and may even be zero for one of them.

The ATM layer is unusual for a connection-oriented protocol in that it does not provide any acknowledgements. The reason for this design is that ATM was designed for use on fiber optic networks, which are highly reliable. It was thought adequate to leave error control to higher layers. After all, sending acknowledgements in the data link or network layer is really only an optimization. It is always sufficient for the transport layer to send a message and then send the entire message again if it is not acknowledged on time.

Furthermore, ATM networks are often used for real-time traffic, such as audio and video. For this kind of traffic, retransmitting an occasional bad cell is worse than just ignoring it.

Despite its lack of acknowledgements, the ATM layer does provide one hard guarantee: cells sent along a virtual circuit will never arrive out of order. The ATM subnet is permitted to discard cells if congestion occurs but under no conditions may it reorder the cells sent on a single virtual circuit. No ordering guarantees are given about cells sent on *different* virtual circuits, however. For example, if a host sends a cell on virtual circuit 10 and later sends a cell on virtual circuit 20 to the same destination, the second cell may arrive first. If the two cells had been sent on the same virtual circuit, the first one sent would always be the first one to arrive.

The ATM layer supports a two-level connection hierarchy that is visible to the transport layer. Along any transmission path from a given source to a given destination, a group of virtual circuits can be grouped together into what is called a **virtual path**, as depicted in Fig. 5-61. Conceptually, a virtual path is like a bundle of twisted copper pairs: when it is rerouted, all the pairs (virtual circuits) are rerouted together. We will consider the implications of this two-level design in detail later.

5.6.1. Cell Formats

In the ATM layer, two interfaces are distinguished: the **UNI** (**User-Network Interface**) and the **NNI** (**Network-Network Interface**). The former defines the boundary between a host and an ATM network (in many cases, between the

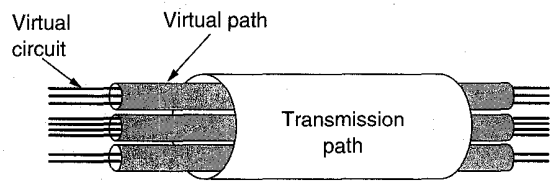


Fig. 5-61. A transmission path can hold multiple virtual paths, each of which can hold multiple virtual circuits.

customer and the carrier). The latter applies to the line between two ATM switches (the ATM term for routers).

In both cases the cells consist of a 5-byte header followed by a 48-byte payload, but the two headers are slightly different. The headers, as defined by the ATM Forum, are illustrated in Fig. 5-62. Cells are transmitted leftmost byte first and leftmost bit within a byte first.

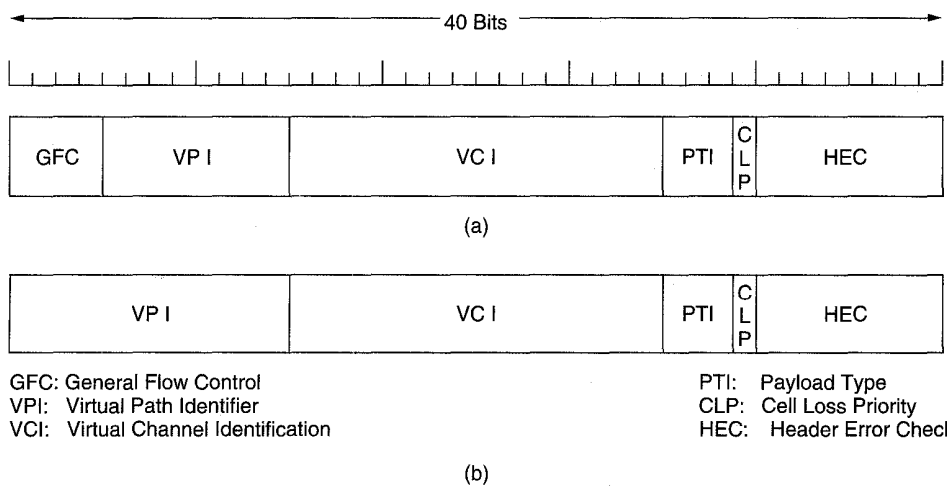


Fig. 5-62. (a) The ATM layer header at the UNI. (b) The ATM layer header at the NNI.

The *GFC* field is present only in cells between a host and the network. It is overwritten by the first switch it reaches, so it does not have end-to-end significance and is not delivered to the destination. It was originally conceived as perhaps having some utility for flow control or priority between hosts and the networks, but no values are defined for it and the network ignores it. The best way to think of it is as a bug in the standard.

The *VPI* field is a small integer selecting a particular virtual path (see Fig. 5-61). Similarly, the *VCI* field selects a particular virtual circuit within the chosen virtual path. Since the *VPI* field has 8 bits (at the UNI) and the *VCI* field has 16

bits, theoretically, a host could have up to 256 VC bundles, each containing up to 65,536 virtual circuits. Actually, slightly fewer of each are available because some *VCI*s are reserved for control functions, such as setting up virtual circuits.

The *PTI* field defines the type of payload the cell contains in accordance with the values given in Fig. 5-63. Here the cell types are user supplied, but the congestion information is network supplied. In other words, a cell sent with *PTI* 000 might arrive with 010 to warn the destination of problems underway.

Payload type	Meaning
000	User data cell, no congestion, cell type 0
001	User data cell, no congestion, cell type 1
010	User data cell, congestion experienced, cell type 0
011	User data cell, congestion experienced, cell type 1
100	Maintenance information between adjacent switches
101	Maintenance information between source and destination switches
110	Resource Management cell (used for ABR congestion control)
111	Reserved for future function

Fig. 5-63. Values of the *PTI* field.

The *CLP* bit can be set by a host to differentiate between high-priority traffic and low-priority traffic. If congestion occurs and cells must be discarded, switches first attempt to discard cells with *CLP* set to 1 before throwing out any set to 0.

Finally, the *HEC* field is a checksum over the header. It does not check the payload. A Hamming code on a 40-bit number only requires 5 bits, so with 8 bits a more sophisticated code can be used. The one chosen can correct all single-bit errors and can detect about 90 percent of all multibit errors. Various studies have shown that the vast majority of errors on optical links are single-bit errors.

Following the header comes 48 bytes of payload. Not all 48 bytes are available to the user, however, since some of the AAL protocols put their headers and trailers inside the payload.

The NNI format is the same as the UNI format, except that the *GFC* field is not present and those 4 bits are used to make the *VPI* field 12 bits instead of 8.

5.6.2. Connection Setup

ATM supports both permanent virtual circuits and switched virtual circuits. The former are always present and can be used at will, like leased lines. The latter have to be established each time they are used, like making phone calls. In this section we will describe how switched virtual circuits are established.

Technically, connection setup is not part of the ATM layer but is handled by the control plane (see Fig. 1-30) using a highly-complex ITU protocol called **Q.2931** (Stiller, 1995). Nevertheless, the logical place to handle setting up a network layer connection is in the network layer, and similar network layer protocols do connection setup here, so we will discuss it here.

Several ways are provided for setting up a connection. The normal way is to first acquire a virtual circuit for signaling and use it. To establish such a circuit, cells containing a request are sent on virtual path 0, virtual circuit 5. If successful, a new virtual circuit is opened on which connection setup requests and replies can be sent and received.

The reason for this two-step setup procedure is that this way the bandwidth reserved for virtual circuit 5 (which is barely used at all) can be kept extremely low. Also, an alternative way is provided to set up virtual circuits. Some carriers may allow users to have permanent virtual paths between predefined destinations, or allow them to set these up dynamically. Once a host has a virtual path to some other host, it can allocate virtual circuits on it itself, without the switches being involved.

Virtual circuit establishment uses the six message types listed in Fig. 5-64. Each message occupies one or more cells and contains the message type, length, and parameters. The messages can be sent by a host to the network or can be sent by the network (usually in response to a message from another host) to a host. Various other status and error reporting messages also exist but are not shown here.

Message	Meaning when sent by host	Meaning when sent by network
SETUP	Please establish a circuit	Incoming call
CALL PROCEEDING	I saw the incoming call	Your call request will be attempted
CONNECT	I accept the incoming call	Your call request was accepted
CONNECT ACK	Thanks for accepting	Thanks for making the call
RELEASE	Please terminate the call	The other side has had enough
RELEASE COMPLETE	Ack for RELEASE	Ack for RELEASE

Fig. 5-64. Messages used for connection establishment and release.

The normal procedure for establishing a call is for a host to send a SETUP message on a special virtual circuit. The network then responds with CALL PROCEEDING to acknowledge receipt of the request. As the SETUP message propagates toward the destination, it is acknowledged at each hop by CALL PROCEEDING.

When the SETUP message finally arrives, the destination host can respond with CONNECT to accept the call. The network then sends a CONNECT ACK message to indicate that it has received the CONNECT message. As the CONNECT message

propagates back toward the originator, each switch receiving it acknowledges it with a CONNECT ACK message. This sequence of events is shown in Fig. 5-65(a).

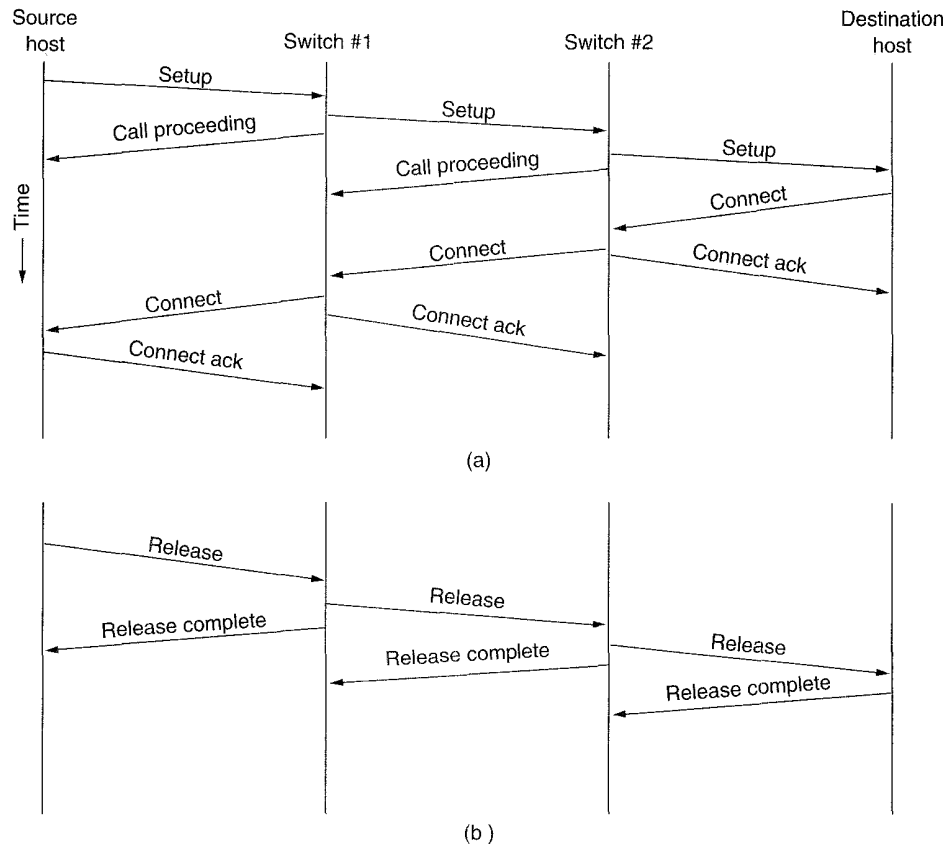


Fig. 5-65. (a) Connection setup in an ATM network. (b) Connection release.

The sequence for terminating a virtual circuit is simple. The host wishing to hang up just sends a RELEASE message, which propagates to the other end and causes the circuit to be released. Each hop along the way, the message is acknowledged, as shown in Fig. 5-65(b).

ATM networks allow multicast channels to be set up. A multicast channel has one sender and more than one receiver. These are constructed by setting up a connection to one of the destinations in the usual way. Then the ADD PARTY message is sent to attach a second destination to the virtual circuit returned by the previous call. Additional ADD PARTY messages can be sent afterward to increase the size of the multicast group.

In order to set up a connection to a destination, it is necessary to specify which destination, by including its address in the SETUP message. ATM addresses

come in three forms. The first is 20 bytes long and is based on OSI addresses. The first byte indicates which of three formats the address is in. In the first format, bytes 2 and 3 specify a country, and byte 4 gives the format of the rest of the address, which contains a 3-byte authority, a 2-byte domain, a 2-byte area, and a 6-byte address, plus some other items. In the second format, bytes 2 and 3 designate an international organization instead of a country. The rest of the address is the same as in format 1. Alternatively, a older form of addressing (CCITT E.164) using 15-digit decimal ISDN telephone numbers is also permitted.

5.6.3. Routing and Switching

When a virtual circuit is set up, the SETUP message wends its way through the network from source to destination. The routing algorithm determines the path taken by this message, and thus by the virtual circuit. The ATM standard does not specify any particular routing algorithm, so the carrier is free to choose among the algorithms discussed earlier in this chapter, or to use a different one.

Experience with previous connection-oriented networks, such as X.25, have shown that a considerable amount of computing power in the switches can be expended determining how to convert the virtual circuit information in each cell to the choice of output line. The ATM designers wanted to avoid this fate, so the ATM layer has been designed to make efficient routing possible. In particular, the idea was to route on the *VPI* field, but not the *VCI* field, except at the final hop in each direction, when cells are sent between a switch and a host. Between two switches, only the virtual path was to be used.

Using only the *VPI*s between interior switches has several advantages. To start with, once a virtual path has been established from a source to a destination, any additional virtual circuits along that path can just follow the existing path. No new routing decisions have to be made. It is as though a bundle of twisted pairs has already been pulled from the source to the destination. Setting up a new connection merely requires allocating one of the unused pairs.

Second, routing of individual cells is easier when all virtual circuits for a given path are always in the same bundle. The routing decision only involves looking at a 12-bit number, not a 12-bit number and a 16-bit number. We will describe how cell switching is done below, but even without going into the details, it should be clear that indexing into a table of 2^{12} entries is feasible whereas indexing into a table of 2^{28} entries is not.

Third, basing all routing on virtual paths makes it easier to switch a whole group of virtual circuits. Consider, for example, the hypothetical U.S. ATM backbone illustrated in Fig. 5-66. Normally, virtual circuits from NY to SF pass through Omaha and Denver. However, suppose a disturbance occurs on the Omaha-Denver line. By rerouting the Omaha-Denver virtual path to LA and then SF, all the virtual circuits (potentially up to 65,535 of them) can be switched in one operation instead of potentially thousands of operations.

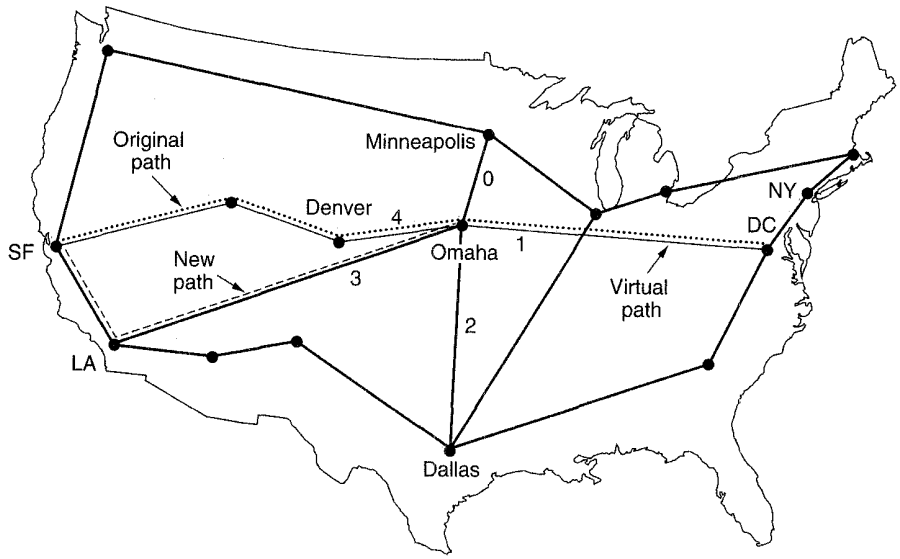


Fig. 5-66. Rerouting a virtual path reroutes all of its virtual circuits.

Finally, virtual paths make it easier for carriers to offer closed user groups (private networks) to corporate customers. A company can set up a network of permanent virtual paths among its various offices, and then allocate virtual circuits within these paths on demand. No calls can come into the private network from outside and no calls can leave the private network, except via special gateways. Many companies like this kind of security.

Whether switches will actually use the *VPI* field for routing as planned, or will, in fact, use the combination of the *VPI* and *VCI* fields (thus negating all the advantages just discussed) remains to be seen. Initial evidence from the field is not encouraging.

Let us now see how cells could be routed within an interior switch (one that is attached only to other switches and not to hosts). To make matters concrete, let us consider the Omaha switch of Fig. 5-66. For each of its five incoming lines, it has a table, *vpi_table*, indexed by incoming *VPI* that tells which of the five outgoing lines to use and what VPI to put in outgoing cells. Let us assume the five lines are numbered from 0 to 4, clockwise starting at Minneapolis. For each outgoing line, the switch maintains a bit map telling which *VPI*s are currently in use on that line.

When the switch is booted, all the entries in all the *vpi_table* structures are marked as not in use. Similarly, all the bit maps are marked to indicate that all *VPI*s are available (except the reserved ones). Now suppose calls come as shown in Fig. 5-67.

As each virtual path (and virtual circuit) is set up, entries are made in the tables. We will assume the virtual circuits are full duplex, so that each one set up

Source	Incoming line	Incoming VPI	Destination	Outgoing line	Outgoing VPI	Path:
NY	1	1	SF	4	1	New
NY	1	2	Denver	4	2	New
LA	3	1	Minneapolis	0	1	New
DC	1	3	LA	3	2	New
NY	1	1	SF	4	1	Old
SF	4	3	DC	1	4	New
DC	1	5	SF	4	4	New
NY	1	2	Denver	4	2	Old
SF	4	5	Minneapolis	0	2	New
NY	1	1	SF	4	1	Old

Fig. 5-67. Some routes through the Omaha switch of Fig. 5-66.

results in two entries, one for the forward traffic from the source and one for the reverse traffic from the destination.

The tables corresponding to the routes of Fig. 5-67 are shown in Fig. 5-68. For example, the first call generates the (4, 1) entry for *VPI* 1 in the DC table because it refers to cells coming in on line 1 with *VPI* 1 and going to SF. However, an entry is also made in the Denver table for *VPI* 1 showing that cells coming in from Denver with *VPI* 1 should go out on line 1 with *VPI* 1. These are cells traveling the other way (from SF to NY) on this virtual path. Note that in some cases two or three virtual circuits are sharing a common path. No new table entries are needed for additional virtual circuits connecting a source and destination that already have a path assigned.

Now we can explain how cells are processed inside a switch. Suppose that a cell arrives on line 1 (DC) with *VPI* 3. The switch hardware or software uses the 3 as an index into the table for line 1 and sees that the cell should go out on line 3 (LA) with *VPI* 2. It overwrites the *VPI* field with a 2 and puts the outgoing line number, 3, somewhere in the cell, for example, in the *HEC* field, since that has to be recomputed later anyway.

Now the question is how to get the cell from its current input buffer to line 3. However, this issue (routing within a switch) was discussed in detail in Chap. 2, and we saw how it was done in knockout and Batcher-banyan switches.

At this point it is straightforward to see how an entire bundle of virtual circuits can be rerouted, as is done in Fig. 5-66. By changing the entry for *VPI* 1 in the DC table from (4, 1) to (3, 3), cells from NY headed for SF will be diverted to LA. Of course, the LA switch has to be informed of this event, so the switch has

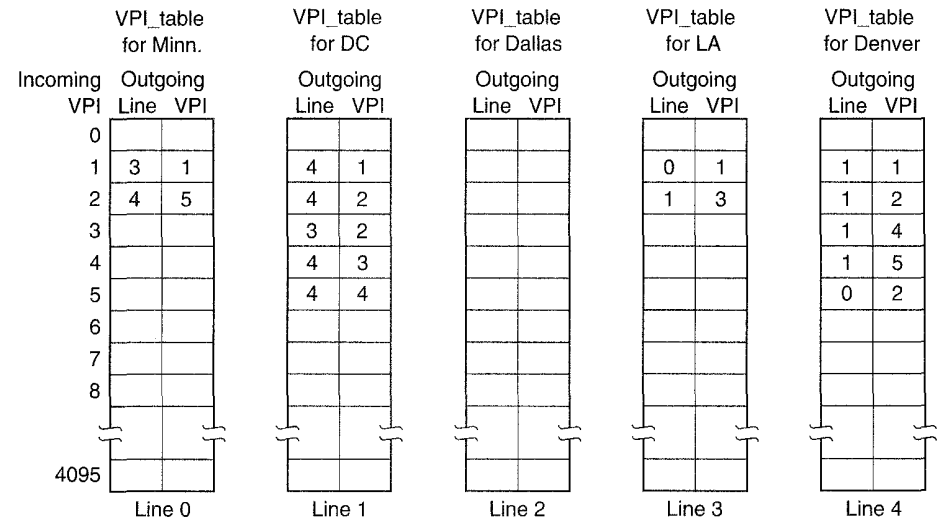


Fig. 5-68. The table entries for the routes of Fig. 5-67.

to generate and send a SETUP message to LA to establish the new path with *VPI* 3. Once this path has been set up, all the virtual circuits from NY to SF are now rerouted via LA, even if there are thousands of them. If virtual paths did not exist, each virtual circuit would have its own table entry and would have to be rerouted separately.

It is worth pointing out explicitly that the discussion above is about ATM in WANs. In a LAN, matters are much simpler. For example, a single virtual path can be used for all virtual circuits.

5.6.4. Service Categories

After a fair amount of trial and error, by version 4.0 of the ATM specification, it was becoming clear what kinds of traffic ATM networks were carrying and what kind of services their customers wanted. Consequently, the standard was modified to explicitly list the service categories commonly used, in order to allow equipment vendors to optimize their adaptor boards and switches for some or all of these categories. The service categories chosen as being important are listed in Fig. 5-69.

The **CBR** (**Constant Bit Rate**) class is intended to emulate a copper wire or optical fiber (only at much greater expense). Bits are put on one end and they come off the other end. No error checking, flow control, or other processing is done. Nevertheless, this class is essential to making a smooth transition between

Class	Description	Example
CBR	Constant bit rate	T1 circuit
RT-VBR	Variable bit rate: real time	Real-time videoconferencing
NRT-VBR	Variable bit rate: non-real time	Multimedia email
ABR	Available bit rate	Browsing the Web
UBR	Unspecified bit rate	Background file transfer

Fig. 5-69. The ATM service categories.

the current telephone system and future B-ISDN systems, since voice-grade PCM channels, T1 circuits, and most of the rest of the telephone system use constant-rate, synchronous bit transmission. With the CBR class, all of this traffic can be carried directly by an ATM system. CBR is also suited to all other interactive (i.e., real-time) audio and video streams.

The next class, **VBR (Variable Bit Rate)**, is divided into two subclasses, for real time and non-real time, respectively. RT-VBR is intended for services that have variable bit rates combined with stringent real-time requirements, such as interactive compressed video (e.g., videoconferencing). Due to the way MPEG and other compression schemes work, with a complete base frame followed by a series of differences between the current frame and the base frame, the transmission rate varies strongly in time (Pancha and El Zarki, 1994). Despite this variation, it is essential that the ATM network not introduce any jitter in the cell arrival pattern, as this will cause the display to appear jerky. In other words, both the average cell delay and the variation in cell delay must be tightly controlled. On the other hand, an occasional lost bit or cell here is tolerable and is best just ignored.

The other VBR subclass is for traffic where timely delivery is important but a certain amount of jitter can be tolerated by the application. For example, multimedia email is typically spooled to the receiver's local disk before being displayed, so any variation in cell delivery times will be eliminated before the email is viewed.

The **ABR (Available Bit Rate)** service category is designed for bursty traffic whose bandwidth range is known roughly. A typical example might be for use in a company that currently connects its offices by a collection of leased lines. Typically, the company has a choice of putting in enough capacity to handle the peak load, which means that some lines are idle part of the day, or putting in just enough capacity for the minimum load, which leads to congestion during the busiest part of the day.

Using ABR service avoids having to make a long term commitment to a fixed bandwidth. With ABR it is possible to say, for example, that the capacity between two points must always be 5 Mbps, but might have peaks up to 10 Mbps.

The system will then guarantee 5 Mbps all the time, and do its best to provide 10 Mbps when needed, but with no promises.

ABR is the only service category in which the network provides rate feedback to the sender, asking it to slow down when congestion occurs. Assuming that the sender complies with such requests, cell loss for ABR traffic is expected to be low. Traveling ABR is a little like flying standby: if there are seats left over (excess capacity), standby passengers are transported without delay. If there is insufficient capacity, they have to wait (unless some of the minimum bandwidth is available).

Finally, we come to **UBR (Unspecified Bit Rate)**, which makes no promises and gives no feedback about congestion. This category is well suited to sending IP packets, since IP also makes no promises about delivery. All UBR cells are accepted, and if there is capacity left over, they will also be delivered. If congestion occurs, UBR cells will be discarded, with no feedback to the sender and no expectation that the sender slows down.

To continue our standby analogy, with UBR, all standby passengers get to board, but if halfway to the destination the pilot sees that fuel is running low, standby passengers are unceremoniously pushed through the emergency exit. To make UBR attractive, carriers are likely to make it cheaper than the other classes. For applications that have no delivery constraints and want to do their own error control and flow control anyway, UBR is a perfectly reasonably choice. File transfer, email, and USENET news are all potential candidates for UBR service because none of these applications have real-time characteristics.

The properties of the various service categories are summarized in Fig. 5-70.

Service characteristic	CBR	RT-VBR	NRT-VBR	ABR	UBR
Bandwidth guarantee	Yes	Yes	Yes	Optional	No
Suitable for real-time traffic	Yes	Yes	No	No	No
Suitable for bursty traffic	No	No	Yes	Yes	Yes
Feedback about congestion	No	No	No	Yes	No

Fig. 5-70. Characteristics of the ATM service categories.

5.6.5. Quality of Service

Quality of service is an important issue for ATM networks, in part because they are used for real-time traffic, such as audio and video. When a virtual circuit is established, both the transport layer (typically a process in the host machine, the "customer") and the ATM network layer (e.g., a network operator, the "carrier") must agree on a contract defining the service. In the case of a public network, this contract may have legal implications. For example, if the carrier agrees not to

lose more than one cell per billion and it loses two cells per billion, the customer's legal staff may get all excited and start running around yelling "breach of contract."

The contract between the customer and the network has three parts:

1. The traffic to be offered.
2. The service agreed upon.
3. The compliance requirements.

It is worth noting that the contract may be different for each direction. For a video-on-demand application, the required bandwidth from the user's remote control to the video server might be 1200 bps. In the other direction it might be 5 Mbps. It should be noted that if the customer and the carrier cannot agree on terms, or the carrier is unable to provide the service desired, the virtual circuit will not be set up.

The first part of the contract is the **traffic descriptor**. It characterizes the load to be offered. The second part of the contract specifies the quality of service desired by the customer and accepted by the carrier. Both the load and the service must be formulated in terms of measurable quantities, so compliance can be objectively determined. Merely saying "moderate load" or "good service" will not do.

To make it possible to have concrete traffic contracts, the ATM standard defines a number of **QoS** (**Quality of Service**) parameters whose values the customer and carrier can negotiate. For each quality of service parameter, the worst case performance for each parameter is specified, and the carrier is required to meet or exceed it. In some cases, the parameter is a minimum; in others it is a maximum. Again here, the quality of service is specified separately for each direction. Some of the more important ones are listed in Fig. 5-71, but not all of them are applicable to all service categories.

The first three parameters specify how fast the user wants to send. **PCR** (**Peak Cell Rate**) is the maximum rate at which the sender is planning to send cells. This parameter may be lower than what the bandwidth of the line permits. If the sender is planning to push out a cell every 4 μsec, its *PCR* is 250,000 cells/sec, even though the actual cell transmission time may be 2.7 μsec.

SCR (**Sustained Cell Rate**) is the expected or required cell rate averaged over a long time interval. For CBR traffic, *SCR* will be equal to *PCR*, but for all the other service categories, it will be substantially lower. The *PCR/SCR* ratio is one measure of the burstiness of the traffic.

MCR (**Minimum Cell Rate**) is the minimum number of cells/sec that the customer considers acceptable. If the carrier is unable to guarantee to provide this much bandwidth it must reject the connection. When ABR service is requested, then the actual bandwidth used must lie between *MCR* and *PCR*, but it may vary

Parameter	Acronym	Meaning
Peak cell rate	PCR	Maximum rate at which cells will be sent
Sustained cell rate	SCR	The long-term average cell rate
Minimum cell rate	MCR	The minimum acceptable cell rate
Cell delay variation tolerance	CDVT	The maximum acceptable cell jitter
Cell loss ratio	CLR	Fraction of cells lost or delivered too late
Cell transfer delay	CTD	How long delivery takes (mean and maximum)
Cell delay variation	CDV	The variance in cell delivery times
Cell error rate	CER	Fraction of cells delivered without error
Severely-errored cell block ratio	SECBR	Fraction of blocks garbled
Cell misinsertion rate	CMR	Fraction of cells delivered to wrong destination

Fig. 5-71. Some of the quality of service parameters.

dynamically during the lifetime of the connection. If the customer and carrier agree to setting MCR to 0, then ABR service becomes similar to UBR service.

CVDT (Cell Variation Delay Tolerance) tells how much variation will be present in cell transmission times. It is specified independently for *PCR* and *SCR*. For a perfect source operating at *PCR*, every cell will appear *exactly* $1/PCR$ after the previous one. No cell will ever be early and no cell will ever be late, not even by a picosecond. For a real source operating at *PCR*, some variation will occur in cell transmission times. The question is: How much variation is acceptable? Can a cell be 1 nsec early? How about 30 seconds? *CDVT* controls the amount of variability acceptable using a leaky bucket algorithm to be described shortly.

The next three parameters describe characteristics of the network and are measured at the receiver. All three are negotiable. **CLR (Cell Loss Ratio)** is straightforward. It measures the fraction of the transmitted cells that are not delivered at all or are delivered so late as to be useless (e.g., for real-time traffic). **CTD (Cell Transfer Delay)** is the average transit time from source to destination. **CDV (Cell Delay Variation)** measures how uniformly the cells are delivered.

The model for *CDT* and *CDV* is shown in Fig. 5-72. Here we see the probability of a cell taking time t to arrive, as a function of t. For a given source, destination, and route through the intermediate switches, some minimum delay always exists due to propagation and switching time. However, not all cells make it in the minimum time; the probability density function usually has a long tail. By choosing a value of *CDT*, the customer and the carrier are, in effect, agreeing, on how late a cell can be delivered and still count as a correctly delivered cell. Normally, *CDV* will be chosen so that, α, the fraction of cells that are rejected for

being too late will be on the order of 10^{-10} or less. *CDV* measures the spread in arrival times. For real-time traffic, this parameter is often more important than *CDT*.

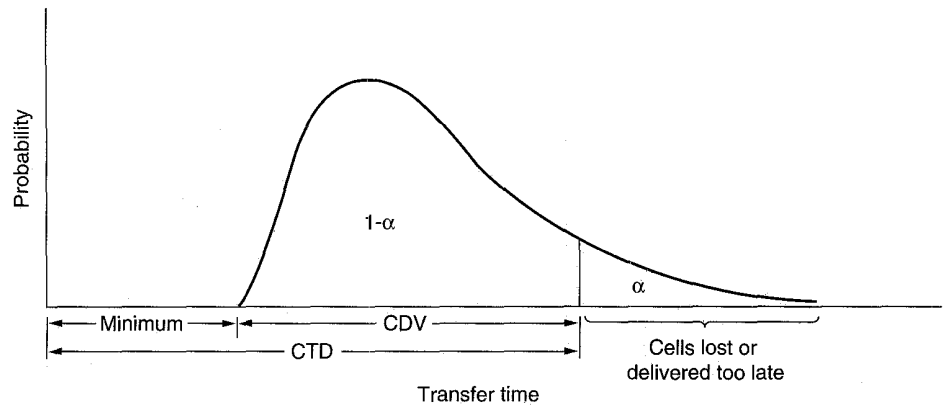


Fig. 5-72. The probability density function for cell arrival times.

The last three QoS parameters specify characteristics of the network. They are generally not negotiable. **CER (Cell Error Ratio)** is the fraction of cells that are delivered with one or more bits wrong. **SECBR (Severely-Errored Cell Block Ratio)** is the fraction of *N*-cell blocks of which *M* or more cells contain an error. Finally, **CMR (Cell Misinsertion Rate)** is the number of cells/sec that are delivered to the wrong destination on account of an undetected error in the header.

The third part of the traffic contract tells what constitutes obeying the rules. If the customer sends one cell too early, does this void the contract? If the carrier fails to meet one of its quality targets for a period of 1 msec, can the customer sue? Effectively, this part of the contract is negotiated between the parties and says how strictly the first two parts will be enforced.

The ATM and Internet quality of service models differ somewhat, which impacts their respective implementations. The ATM model is based strictly on connections, whereas the Internet model uses datagrams plus flows (e.g., RSVP). A comparison of these two models is given in (Crowcroft et al., 1995).

5.6.6. Traffic Shaping and Policing

The mechanism for using and enforcing the quality of service parameters is based (in part) on a specific algorithm, the **Generic Cell Rate Algorithm (GCRA)**, and is illustrated in Fig. 5-73. It works by checking every cell to see if it conforms to the parameters for its virtual circuit.

GCRA has two parameters. These specify the maximum allowed arrival rate (*PCR*) and the amount of variation herein that is tolerable (*CDVT*). The

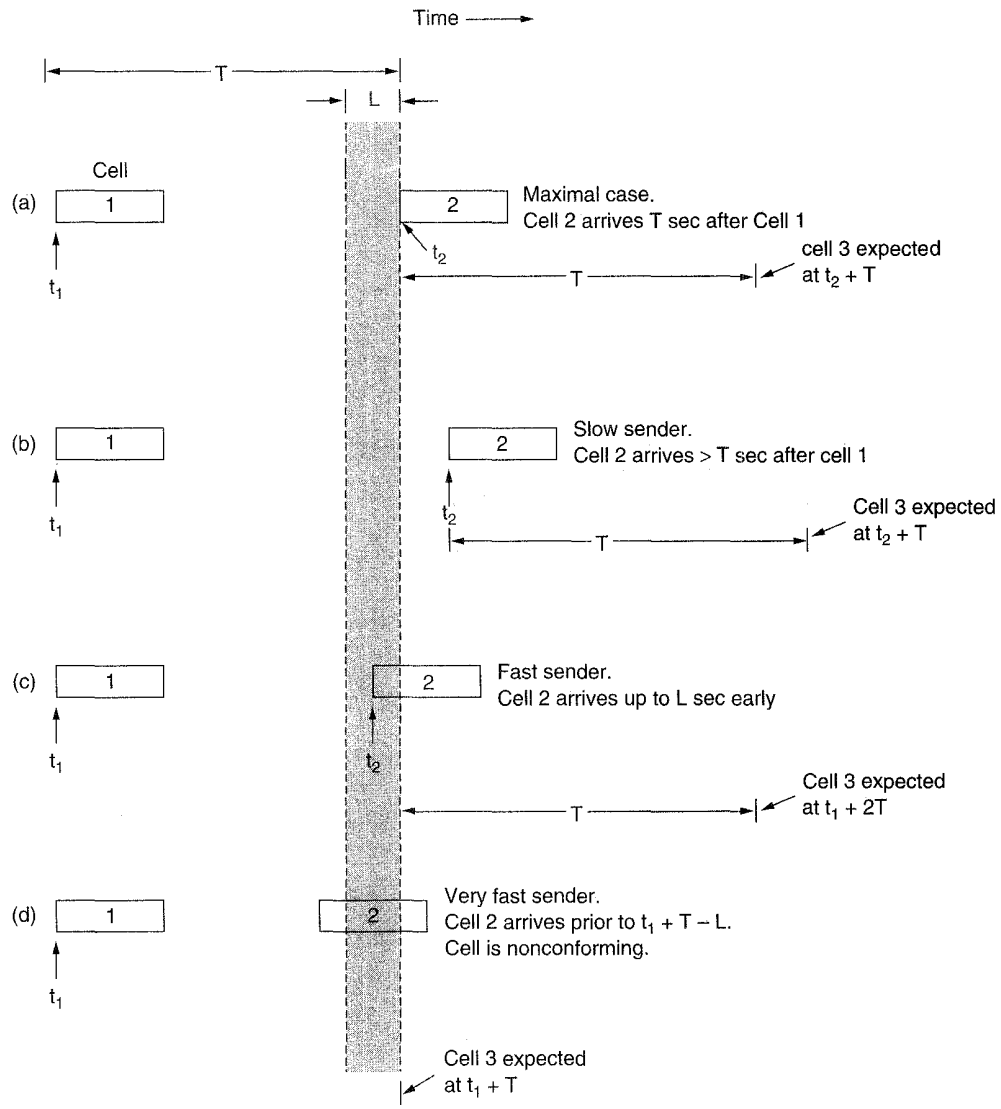


Fig. 5-73. The generic cell rate algorithm.

reciprocal of *PCR*, $T = 1/PCR$, is the minimum cell interarrival time, as shown in Fig. 5-73(a). If the customer agrees not to send more than 100,000 cells/sec, then $T = 10$ μsec. In the maximal case, one cell arrives promptly every 10 μsec. To avoid tiny numbers, we will work in microseconds, but since all the parameters are real numbers, the unit of time does not matter.

A sender is always permitted to space consecutive cells more widely than T, as shown in Fig. 5-73(b). Any cell arriving more than T μsec after the previous one is conforming.

The problem arises with senders that tend to jump the gun, as in Fig. 5-73(c) and (d). If a cell arrives a little early (at or later than $t_1 + T - L$), it is conforming, but the next cell is still expected at $t_1 + 2T$, (not at $t_2 + T$), to prevent the sender from transmitting every cell L μsec early, and thus increasing the peak cell rate.

If a cell arrives more than L μsec early, it is declared as nonconforming. The treatment of nonconforming cells is up to the carrier. Some carriers may simply discard them; others may keep them, but set the *CLP* bit, to mark them as low priority to allow switches to drop nonconforming cells first in the event of congestion. The use of the *CLP* bit may also be different for the different service categories of Fig. 5-69.

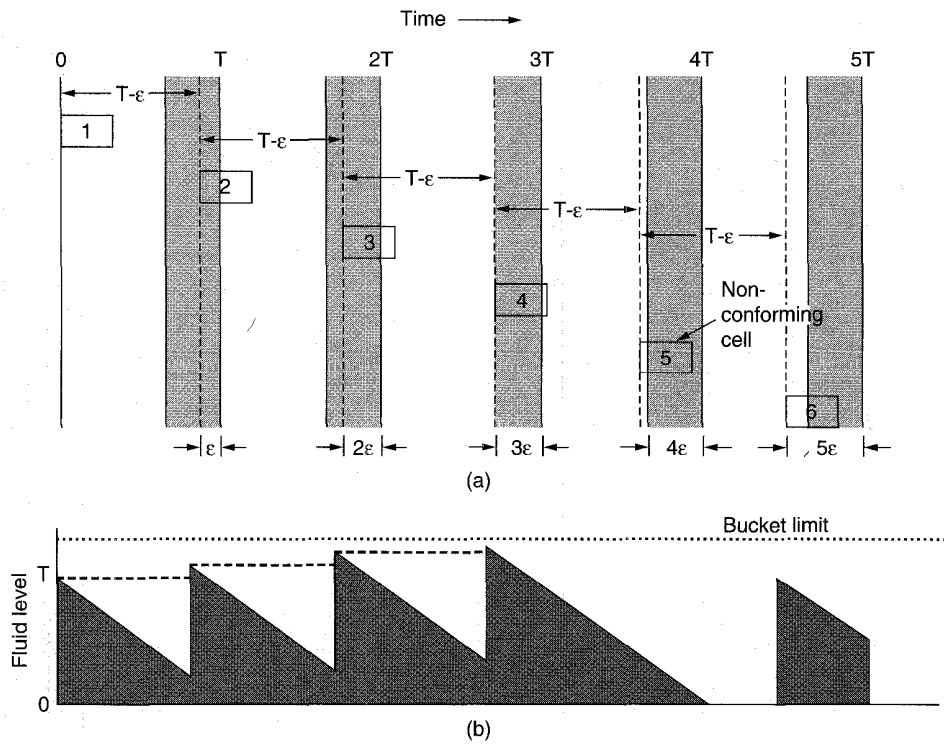


Fig. 5-74. (a) A sender trying to cheat. (b) The same cell arrival pattern, but now viewed in terms of a leaky bucket.

Now let us consider what happens if a sender tries to cheat a little bit, as shown in Fig. 5-74(a). Instead of waiting until time T to send cell 2, the sender

transmits it a wee bit early, at $T - \varepsilon$, where, say, $\varepsilon = 0.3L$. This cell is accepted without problems.

Now the sender transmits cell 3, again at $T - \varepsilon$ after the previous cell, that is, at $T - 2\varepsilon$. Again it is accepted. However, every successive cell inches closer and closer to the fatal $T - L$ boundary. In this case, cell 5 arrives at $T - 4\varepsilon$ ($T - 1.2L$) which is too early, so cell 5 is declared nonconforming and is discarded by the network interface.

When viewed in these terms, the GCRA algorithm is called a **virtual scheduling algorithm**. However, viewed differently, it is equivalent to a leaky bucket algorithm, as depicted in Fig. 5-74(b). Imagine that each conforming cell that arrives pours T units of fluid into a leaky bucket. The bucket leaks fluid at a rate of 1 unit/μsec, so that after T μsec it is all gone. If cells arrive precisely every T μsec, each arriving cell will find the bucket (just) emptied, and will refill it with T units of fluid. Thus the fluid level is raised to T when a cell arrives and is reduced linearly until it gets to zero. This situation is illustrated in Fig. 5-74(b) between 0 and T.

Since fluid drains out linearly in time, at a time t after a cell arrives, the amount of its fluid left is $T - t$. At the time cell 2 arrives, at $T - \varepsilon$, there are still ε units of fluid in the bucket. The addition of the new cell raises this value to $T + \varepsilon$. Similarly, at the time cell 3 arrives, 2ε units are left in the bucket so the new cell raises the fluid level to $T + 2\varepsilon$. When cell 4 arrives, it is raised to $T + 3\varepsilon$.

If this goes on indefinitely, some cell is going to raise the level to above the bucket capacity and thus be rejected. To see which one it is, let us now compute what the bucket capacity is. We want the leaky bucket algorithm to give the same result as Fig. 5-74(a), so we want overflow to occur when a cell arrives L μsec early. If the fluid left requires L μsec to drain out, the amount of fluid must be L, since the drain rate is 1 unit/μsec. Thus we want the bucket capacity to be $L + T$ so that any cell arriving more than L μsec early will be rejected due to bucket overflow. In Fig. 5-74(b), when cell 5 arrives, the addition of T units to the 4ε units of fluid already present raises the bucket level to $T + 4\varepsilon$. Since we are using $\varepsilon = 0.3L$ in this example, the bucket would be raised to $T + 1.2L$ by the addition of cell 5, so the cell is rejected, no new fluid is added, and the bucket eventually empties.

For a given T, if we set L very small, the capacity of the bucket will be hardly more than T, so all cells will have to be sent with a very uniform spacing. However, if we now raise L to a value much greater than T, the bucket can hold multiple cells because $T + L \gg T$. This means that the sender can transmit a burst of cells back-to-back at the peak rate and have them still accepted.

We can easily compute the number of conforming cells, N, that can be transmitted back-to-back at the peak cell rate ($PCR = 1/T$). During a burst of N cells, the total amount of fluid added to the bucket is NT because each cell adds T. However, during the maximum burst, fluid drains out of the bucket at a rate of 1 unit per time interval. Let us call the cell transmission time δ time units. Note

that $\delta \leq T$ because it is entirely possible for a sender on a 155.52 Mbps line to agree to send no more than 100,000 cells/sec, in which case $\delta = 2.73$ μsec and $T = 10$ μsec. During the burst of N cells, the amount of drainage is $(N-1)\delta$ because drainage does not start until the first cell has been entirely transmitted.

From these observations, we set that the net increase in fluid in the bucket during the maximum burst is $NT - (N-1)\delta$. The bucket capacity is $T + L$. Equating these two quantities we get

$$NT - (N-1)\delta = T + L$$

Solving this equation for N we get

$$N = 1 + \frac{L}{T - \delta}$$

However, if this number is not an integer, it must be rounded downward to an integer to prevent bucket overflow. For example, with $PCR = 100{,}000$ cells/sec, $\delta = 2.73$ μsec, and $L = 50$ μsec, seven cells may be sent back-to-back at the 155.52 Mbps rate without filling the bucket. An eighth cell would be nonconforming.

The GCRA is normally specified by giving the parameters T and L. T is just the reciprocal of *PCR*; L is *CDVT*. The GCRA is also used to make sure the mean cell rate does not exceed *SCR* for any substantial period.

In this example we assumed that cells arrive uniformly. In reality, they do not. Nevertheless, the leaky bucket algorithm can also be used here, too. At every cell arrival, a check is made to see if there is room in the bucket for an additional T units of fluid. If there is, the cell is conforming; otherwise it is not.

In addition to providing a rule about which cells are conforming and which ones are not, the GCRA also shapes the traffic to remove some of the burstiness. The smaller *CDVT* is, the greater the smoothing effect, but the greater the chance that cells will be discarded as nonconforming. Some implementations combine the GCRA leaky bucket with a token bucket, to provide additional smoothing.

5.6.7. Congestion Control

Even with traffic shaping, ATM networks do not automatically meet the performance requirements set forth in the traffic contract. For example, congestion at intermediate switches is always a potential problem, especially when over 350,000 cells/sec are pouring in on each line, and a switch can have 100 lines. Consequently, a great deal of thought has gone into the subject of performance and congestion in ATM networks. In this section, we will discuss some of the approaches used. For additional information, see (Eckberg, 1992; Eckberg et al., 1991; Hong and Suda, 1991; Jain, 1995; and Newman, 1994).

ATM networks must deal with both long-term congestion, caused by more traffic coming in than the system can handle, and short-term congestion, caused

by burstiness in the traffic. As a result, several different strategies are used together. The most important of these fall into three categories:

1. Admission control.
2. Resource reservation.
3. Rate-based congestion control.

We will now discuss each of these strategies in turn.

Admission Control

In low-speed networks, it is usually adequate to wait for congestion to occur and then react to it by telling the source of the packets to slow down. In high-speed networks, this approach often works poorly, because in the interval between sending the notification and notification arriving at the source, thousands of additional packets may arrive.

Furthermore, many ATM networks have real-time traffic sources that produce data at an intrinsic rate. Telling such a source to slow down may not work (imagine a new digital telephone with a red light on it; when congestion is signaled, the red line comes on and the speaker is required to talk 25 percent slower).

Consequently, ATM networks emphasize preventing congestion from occurring in the first place. However, for CBR, VBR, and UBR traffic, no dynamic congestion control is present at all, so here an ounce of prevention is worth a pound (actually, more like a metric ton) of cure. A major tool for preventing congestion is admission control. When a host wants a new virtual circuit, it must describe the traffic to be offered and the service expected. The network can then check to see if it is possible to handle this connection without adversely affecting existing connections. Multiple potential routes may have to be examined to find one which can do the job. If no route can be located, the call is rejected.

Denying admission should be done fairly. Is it fair that one couch potato zapping through dozens of television programs can wipe out 100 busy beavers trying to read their email? If no controls are applied, a small number of high-bandwidth users can severely affect many low-bandwidth users. To prevent this, users should be divided into classes based on usage. The probability of service denial should be roughly the same for all classes (possibly by giving each class its own resource pool).

Resource Reservation

Closely related to admission control is the technique of reserving resources in advance, usually at call setup time. Since the traffic descriptor gives the peak cell rate, the network has the possibility of reserving enough bandwidth along the path

to handle that rate. Bandwidth can be reserved by having the SETUP message earmark bandwidth along each line it traverses, making sure, of course, that the total bandwidth earmarked along a line is less than the capacity of that line. If the SETUP message hits a line that is full, it must backtrack and look for an alternative path.

The traffic descriptor can contain not only the peak bandwidth, but also the average bandwidth. If a host wants, for example, a peak bandwidth of 100,000 cells/sec, but an average bandwidth of only 20,000 cells/sec, in principle, five such circuits could be multiplexed onto the same physical trunk. The trouble is that all five connections could be idle for half an hour, then start blasting away at the peak rate, causing massive cell loss. Since VBR traffic can be statistically multiplexed, problems can occur with this service category. Possible solutions are being studied.

Rate-Based Congestion Control

With CBR and VBR traffic, it is generally not possible for the sender to slow down, even in the event of congestion, due to the inherent real-time or semi-real-time nature of the information source. With UBR, nobody cares; if there are too many cells, the extra ones are just dropped.

However, with ABR traffic, it is possible and reasonable for the network to signal one or more senders and ask them to slow down temporarily until the network can recover. It is in the interest of a sender to comply, since the network can always punish it by throwing out its (excess) cells.

How congestion should be detected, signaled, and controlled for ABR traffic was a hot topic during the development of the ATM standard, with vigorous arguments for various proposed solutions. Let us now briefly look at some of the solutions that were quickly rejected before examining the winner.

In one proposal, whenever a sender wished to send a burst of data, it first had to send a special cell reserving the necessary bandwidth. After the acknowledgement came back, the burst could begin. The advantage here is that congestion never occurs because the required bandwidth is always there when it is needed. The ATM Forum rejected this solution due to the potentially long delay before a host may begin to send.

A second proposal had switches sending back choke cells whenever congestion began to occur. Upon receipt of such a cell, a sender was expected to cut back to half its current cell transmission rate. Various schemes were proposed for getting the rate back up again later when the congestion cleared up. This scheme was rejected because choke cells might get lost in the congestion, and because the scheme seemed unfair to small users. For example, consider a switch getting 100-Mbps streams from each of five users, and one 100-kbps stream from another user. Many committee members felt it was inappropriate to tell the 100 kbps user to give up 50 kbps because he was causing too much congestion.

A third proposal used the fact that packet boundaries are marked by a bit in the last cell. The idea here was to discard cells to relieve the congestion but to do this highly selectively. The switch was to scan the incoming cell stream for the end of a packet and then throw out all the cells in the next packet. Of course, this one packet would be transmitted later, but dropping all k cells in one packet ultimately leads to one packet retransmission, which is far better than dropping k random cells, which might lead to k packet retransmissions. This scheme was rejected on fairness grounds because the next end-of-packet mark seen might not belong to the sender overloading the switch. Also the scheme did not need to be standardized. Any switch vendor is free to pick which cells to discard when congestion occurs.

After much discussion, the battle focused on two contenders, a credit-based solution (Kung and Morris, 1995) and rate-based solution (Bonomi and Fendick, 1995). The credit-based solution was essentially a dynamic sliding window protocol. It required each switch to maintain, per virtual circuit, a credit—effectively the number of buffers reserved for that circuit. As long as each transmitted cell had a buffer waiting for it, congestion could never arise.

The argument against it came from the switch vendors. They did not want to do all the accounting to keep track of the credits and did not want to reserve so many buffers in advance. The amount of overhead and waste required was thought to be too much, so ultimately, the rate-based congestion control scheme was adopted. It works like this.

The basic model is that after every k data cells, each sender transmits a special **RM (Resource Management)** cell. This cell travels along the same path as the data cells, but is treated specially by the switches along the way. When it gets to the destination, it is examined, updated, and sent back to the sender. The full path for RM cells is shown in Fig. 5-75.

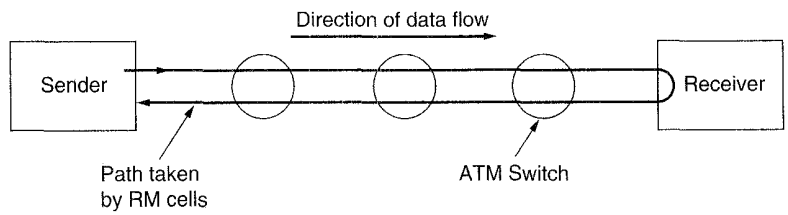


Fig. 5-75. The path taken by RM cells.

In addition, two other congestion control mechanisms are provided. First, overloaded switches can spontaneously generate RM cells and ship them back to the sender. Second, overloaded switches can set the middle *PTI* bit on data cells traveling from the sender to the receiver. Neither of these methods are fully reliable however, since these cells may be lost in the congestion without anyone noticing. In contrast, a lost RM cell will be noticed by the sender when it fails to

return within the expected time interval. As an aside, the *CLP* bit is not used for ABR congestion control.

ABR congestion control is based on the idea that each sender has a current rate, **ACR** (**Actual Cell Rate**) that falls between *MCR* and *PCR*. When congestion occurs, *ACR* is reduced (but not below *MCR*). When congestion is absent, *ACR* is increased (but not above *PCR*). Each RM cell sent contains the rate at which the sender would currently like to transmit, possibly *PCR*, possibly lower. This value is called **ER** (**Explicit Rate**). As the RM cell passes through various switches on the way to the receiver, those that are congested may reduce *ER*. No switch may increase it. Reduction can occur either in the forward direction or in the reverse direction. When the sender gets the RM cell back, it can then see what the minimum acceptable rate is according to all the switches along the path. It can then adjust *ACR*, if need be, to bring it into line with what the slowest switch can handle.

The congestion mechanism using the middle *PTI* bit is integrated into the RM cells by having the receiver include this bit (taken from the last data cell) in each RM cell sent back. The bit cannot be taken from the RM cell itself because all RM cells have this bit set all the time, as shown in Fig. 5-63.

The ATM layer is quite complicated. In this chapter, we have highlighted only a portion of the issues. For additional information, see (De Prycker, 1993; McDysan and Spohn, 1995; Minoli and Vitella, 1994; and La Porta et al., 1994). However, the reader should be warned that all these references discuss the ATM 3 standard, not the ATM 4 standard, which was not finalized until 1996.

5.6.8. ATM LANs

As it becomes increasingly obvious that ITU's original goal of replacing the public switched telephone network by an ATM network is going to take a very long time, attention is shifting to the use of ATM technology to connect existing LANs together. In this approach, an ATM network can function either as a LAN, connecting individual hosts, or as a bridge, connecting multiple LANs. Although both concepts are interesting, they raise some challenging issues that we will discuss below. Additional information about ATM LANs can be found in (Chao et al., 1994; Newman, 1994; Truong et al., 1995).

The major problem that must be solved is how to provide connectionless LAN service over a connection-oriented ATM network. One possible solution is to introduce a connectionless server into the network. Every host initially sets up a connection to this server, and sends all packets to it for forwarding. While simple, this solution does not use the full bandwidth of the ATM network, and the connectionless server can easily become a bottleneck.

An alternative approach, proposed by the ATM Forum, is shown in Fig. 5-76. Here every host has a (potential) ATM virtual circuit to every other host. These virtual circuits can be established and released dynamically as needed, or they can

be permanent virtual circuits. To send a frame, the source host first encapsulates the packet in the payload field of an ATM AAL message and sends it to the destination, the same way frames are shipped over Ethernets, token rings, and other LANs.

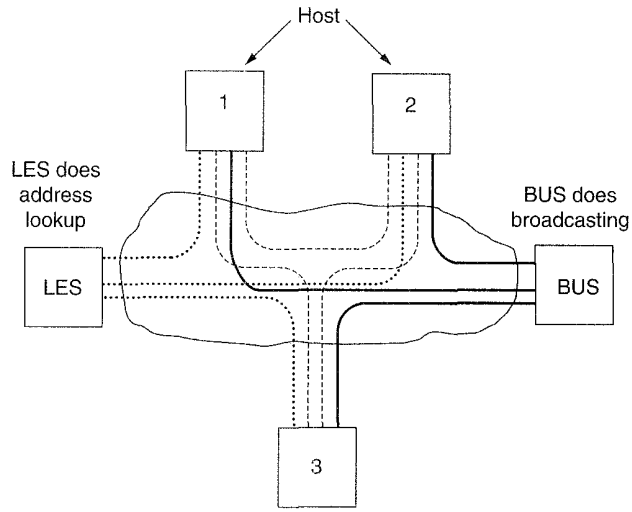


Fig. 5-76. ATM LAN emulation.

The main problem introduced by this scheme is how to tell which IP (or other network layer address) belongs to which virtual circuit. In an 802 LAN, this problem is solved by the ARP protocol, in which a host can broadcast a request such as: "Who has IP address 192.31.20.47?" The host using that address then sends back a point-to-point response, which is cached for later use.

With an ATM LAN, this solution does not work because ATM LANs do not support broadcasting. This problem is solved by introducing a new server, the **LES (LAN Emulation Server)**. To look up a network layer address (e.g., an IP address), a host sends a packet (e.g., an ARP request) to the LES, which then looks up the corresponding ATM address and returns it to the machine requesting it. This address can then be used to send encapsulated packets to the destination.

However, this solution only solves the host location problem. Some programs use broadcasting or multicasting as an essential part of the application. For these applications, the **BUS (Broadcast/Unknown Server)** is introduced. It has connections to all hosts and can simulate broadcasting by sending a packet to all of them, one at a time. Hosts can also speed up delivery of a packet to an unknown host by sending the packet to the BUS for broadcasting and then (in parallel) looking up the address (for future use) using the LES.

A model similar to this one has been adopted by the IETF as the official Internet way to use an ATM network for transporting IP packets. In this model the

LES server is called the **ATMARP** server, but the functionality is essentially the same. Broadcasting and multicasting are not supported in the IETF proposal. The model is described in RFC 1483 and RFC 1577. Another good source of information is (Comer, 1995).

In the IETF method, a set of ATM hosts can be grouped together to form a **logical IP subnet**. Each LIS has its own ATMARP server. In effect, a LIS acts like a virtual LAN. Hosts on the same LIS may exchange IP packets directly, but hosts on different ones are required to go through a router. The reason for having LISes is that every host on a LIS must (potentially) have an open virtual circuit to every other host on its LIS. By restricting the number of hosts per LIS, the number of open virtual circuits can be reduced to a manageable number.

Another use of ATM networks is to use them as bridges to connect existing LANs. In this configuration, only one machine on each LAN needs an ATM connection. Like all transparent bridges, the ATM bridge must listen promiscuously to all LANs to which it is attached, forwarding frames where needed. Since bridges use only MAC addresses (not IP addresses), ATM bridges must build a spanning tree, just as 802 bridges.

In short, while ATM LAN emulation is an interesting idea, there are serious questions about its performance and price, and there is certainly heavy competition from existing LANs and bridges, which are well established and highly optimized. Whether ATM LANs and bridges ever replace 802 LANs and bridges remains to be seen.

5.7. SUMMARY

The network layer provides services to the transport layer. It can be based on either virtual circuits or datagrams. In both cases, its main job is routing packets from the source to the destination. In virtual circuit subnets, a routing decision is made when the virtual circuit is set up. In datagram subnets, it is made on every packet.

Many routing algorithms are used in computer networks. Static algorithms include shortest path routing, flooding, and flow-based routing. Dynamic algorithms include distance vector routing and link state routing. Most actual networks use one of these. Other important routing topics are hierarchical routing, routing for mobile hosts, broadcast routing, and multicast routing.

Subnets can become congested, increasing the delay and lowering the throughput for packets. Network designers attempt to avoid congestion by proper design. Techniques include traffic shaping, flow specifications, and bandwidth reservation. If congestion does occur, it must be dealt with. Choke packets can be sent back, load can be shed, and other methods applied.

Networks differ in various ways, so when multiple networks are connected together problems can occur. Sometimes the problems can be finessed by

tunneling a packet through a hostile network, but if the source and destination networks are different, this approach fails. When different networks have different maximum packet sizes, fragmentation may be called for.

The Internet has a rich variety of protocols related to the network layer. These include the data transport protocol, IP, but also the control protocols ICMP, ARP, and RARP, and the routing protocols OSPF and BGP. The Internet is rapidly running out of IP addresses, so a new version of IP, IPv6, has been developed.

Unlike the datagram-based Internet, ATM networks use virtual circuits inside. These must be set up before data can be transferred and torn down after transmission is completed. Quality of service and congestion control are major issues with ATM networks.

PROBLEMS

1. Give two example applications for which connection-oriented service is appropriate. Now give two examples for which connectionless service is best.
2. Are there any circumstances when a virtual circuit service will (or at least should) deliver packets out of order? Explain.
3. Datagram subnets route each packet as a separate unit, independent of all others. Virtual circuit subnets do not have to do this, since each data packet follows a predetermined route. Does this observation mean that virtual circuit subnets do not need the capability to route isolated packets from an arbitrary source to an arbitrary destination? Explain your answer.
4. Give three examples of protocol parameters that might be negotiated when a connection is set up.
5. Consider the following design problem concerning implementation of virtual circuit service. If virtual circuits are used internal to the subnet, each data packet must have a 3-byte header, and each router must tie up 8 bytes of storage for circuit identification. If datagrams are used internally, 15-byte headers are needed, but no router table space is required. Transmission capacity costs 1 cent per 10^6 bytes, per hop. Router memory can be purchased for 1 cent per byte and is depreciated over two years (business hours only). The statistically average session runs for 1000 sec, in which time 200 packets are transmitted. The mean packet requires four hops. Which implementation is cheaper, and by how much?
6. Assuming that all routers and hosts are working properly and that all software in both is free of all errors, is there any chance, however small, that a packet will be delivered to the wrong destination?

7. Give a simple heuristic for finding two paths through a network from a given source to a given destination that can survive the loss of any communication line (assuming two such paths exist). The routers are considered reliable enough, so it is not necessary to worry about the possibility of router crashes.

8. Consider the subnet of Fig. 5-15(a). Distance vector routing is used, and the following vectors have just come in to router *C*: from *B*: (5, 0, 8, 12, 6, 2); from *D*: (16, 12, 6, 0, 9, 10); and from *E*: (7, 6, 3, 9, 0, 4). The measured delays to *B*, *D*, and *E*, are 6, 3, and 5, respectively. What is *C*'s new routing table? Give both the outgoing line to use and the expected delay.

9. If delays are recorded as 8-bit numbers in a 50-router network, and delay vectors are exchanged twice a second, how much bandwidth per (full-duplex) line is chewed up by the distributed routing algorithm? Assume that each router has three lines to other routers.

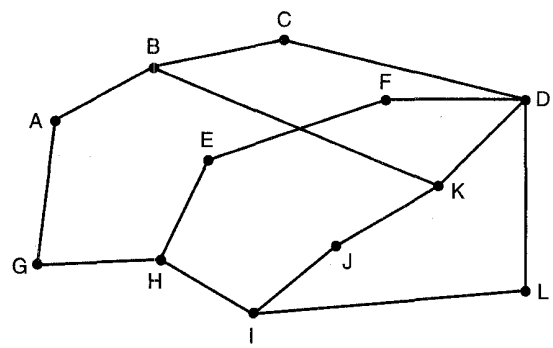
10. In Fig. 5-16 the Boolean OR of the two sets of ACF bits are 111 in every row. Is this just an accident here, or does it hold for all subnets under all circumstances?

11. For hierarchical routing with 4800 routers, what region and cluster sizes should be chosen to minimize the size of the routing table for a three-layer hierarchy?

12. In the text it was stated that when a mobile host is not at home, packets sent to its home LAN are intercepted by its home agent. For an IP network on an 802.3 LAN, how does the home agent accomplish this interception?

13. Looking at the subnet of Fig. 5-5, how many packets are generated by a broadcast from *B*, using
    (a) reverse path forwarding?
    (b) the sink tree?

14. Compute a multicast spanning tree for router *C* in the subnet below for a group with members at routers *A*, *B*, *C*, *D*, *E*, *F*, *I*, and *K*.



15. As a possible congestion control mechanism in a subnet using virtual circuits internally, a router could refrain from acknowledging a received packet until (1) it knows its last transmission along the virtual circuit was received successfully and (2) it has a free buffer. For simplicity, assume that the routers use a stop-and-wait protocol and that each virtual circuit has one buffer dedicated to it for each direction of traffic. If it

takes T sec to transmit a packet (data or acknowledgement) and there are n routers on the path, what is the rate at which packets are delivered to the destination host? Assume that transmission errors are rare, and that the host-router connection is infinitely fast.

16. A datagram subnet allows routers to drop packets whenever they need to. The probability of a router discarding a packet is p. Consider the case of a source host connected to the source router, which is connected to the destination router, and then to the destination host. If either of the routers discards a packet, the source host eventually times out and tries again. If both host-router and router-router lines are counted as hops, what is the mean number of
(a) hops a packet makes per transmission?
(b) transmissions a packet makes?
(c) hops required per received packet?

17. Give an argument why the leaky bucket algorithm should allow just one packet per tick, independent of how large the packet is.

18. The byte-counting variant of the leaky bucket algorithm is used in a particular system. The rule is that one 1024-byte packet, two 512-byte packets, etc. may be sent on each tick. Give a serious restriction of this system that was not mentioned in the text.

19. An ATM network uses a token bucket scheme for traffic shaping. A new token is put into the bucket every 5 μsec. What is the maximum sustainable net data rate (i.e., excluding header bits)?

20. A computer on a 6-Mbps network is regulated by a token bucket. The token bucket is filled at a rate of 1 Mbps. It is initially filled to capacity with 8 megabits. How long can the computer transmit at the full 6 Mbps?

21. Figure 5-27 shows four input characteristics for a proposed flow specification. Imagine that the maximum packet size is 1000 bytes, the token bucket rate is 10 million bytes/sec, the token bucket size is 1 million bytes, and the maximum transmission rate is 50 million bytes/sec. How long can a burst at maximum speed last?

22. A device accepts frames from the Ethernet to which it is attached. It removes the packet inside each frame, adds framing information around it, and transmits it over a leased telephone line (its only connection to the outside world) to an identical device at the other end. This device removes the framing, inserts the packet into a token ring frame, and transmits it to a local host over a token ring LAN. What would you call the device?

23. Is fragmentation needed in concatenated virtual circuit internets, or only in datagram systems?

24. Tunneling through a concatenated virtual circuit subnet is straightforward: the multiprotocol router at one end just sets up a virtual circuit to the other end and passes packets through it. Can tunneling also be used in datagram subnets? If so, how?

25. An IP datagram using the *Strict source routing* option has to be fragmented. Do you think the option is copied into each fragment, or is it sufficient to just put it in the first fragment? Explain your answer.

26. Suppose that instead of using 16 bits for the network part of a class B address, 20 bits had been used. How many class B networks would there have been?

27. Convert the IP address whose hexadecimal representation is C22F1582 to dotted decimal notation.

28. A class B network on the Internet has a subnet mask of 255.255.240.0. What is the maximum number of hosts per subnet?

29. You have just explained the ARP protocol to a friend. When you are all done, he says: "I've got it. ARP provides a service to the network layer, so it is part of the data link layer." What do you say to him?

30. ARP and RARP both map addresses from one space to another. In this respect, they are similar. However, their implementations are fundamentally different. In what major way do they differ?

31. Describe a way to do reassembly of IP fragments at the destination.

32. Most IP datagram reassembly algorithms have a timer to avoid having a lost fragment tie up reassembly buffers forever. Suppose a datagram is fragmented into four fragments. The first three fragments arrrive, but the last one is delayed. Eventually the timer goes off and the three fragments in the receiver's memory are discarded. A little later, the last fragment stumbles in. What should be done with it?

33. Most IP routing protocols use number of hops as the metric to be minimized when doing routing computations. For ATM networks, number of hops is not terribly important. Why not? *Hint*: Take a look at Chap. 2. to see how ATM switches work. Do they use store-and-forward?

34. In both IP and ATM, the checksum covers only the header and not the data. Why do you suppose this design was chosen?

35. A person who lives in Boston travels to Minneapolis, taking her portable computer with her. To her surprise, the LAN at her destination in Minneapolis is a wireless IP LAN, so she does not have to plug in. Is it still necessary to go through the entire business with home agents and foreign agents to make email and other traffic arrive correctly?

36. IPv6 uses 16-byte addresses. If a block of 1 million addresses is allocated every picosecond, how long will the addresses last?

37. The *Protocol* field used in the IPv4 header is not present in the fixed IPv6 header. Why not?

38. When the IPv6 protocol is introduced, does the ARP protocol have to be changed? If so, are the changes conceptual or technical?

39. In Chap. 1, we classified interactions between the network and the hosts using four classes of primitives: *request*, *indication*, *response*, and *confirm*. Classify the SETUP and CONNECT messages of Fig. 5-65 into these categories.

40. A new virtual circuit is being set up in an ATM network. Between the source and destination hosts lie three ATM switches. How many messages (including acknowledgements) will be sent to establish the circuit?

41. The logic used to construct the table of Fig. 5-67 is simple: the lowest unused *VPI* is always assigned to a connection. If a new virtual circuit is requested between NY and Denver, which *VPI* will be assigned to it?

42. In Fig. 5-73(c), if a cell arrives early, the next one is still due at $t_1 + 2T$. Suppose that the rule were different, namely that the next cell was expected at $t_2 + T$, and the sender made maximum use of this rule. What maximum peak cell rate could then be achieved? For $T = 10$ μsec and $L = 2$ μsec, give the original and new peak cell rates, respectively.

43. What is the maximum burst length on an 155.52 Mbps ATM ABR connection whose *PCR* value is 200,000 and whose L value is 25 μsec?

44. Write a program to simulate routing using flooding. Each packet should contain a counter that is decremented on each hop. When the counter gets to zero, the packet is discarded. Time is discrete, with each line handling one packet per time interval. Make three versions of the program: all lines are flooded, all lines except the input line are flooded, and only the (statically chosen) best k lines are flooded. Compare flooding with deterministic routing ($k = 1$) in terms of delay and bandwidth used.

45. Write a program that simulates a computer network using discrete time. The first packet on each router queue makes one hop per time interval. Each router has only a finite number of buffers. If a packet arrives and there is no room for it, it is discarded and not retransmitted. Instead, there is an end-to-end protocol, complete with timeouts and acknowledgement packets, that eventually regenerates the packet from the source router. Plot the throughput of the network as a function of the end-to-end timeout interval, parametrized by error rate.

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.

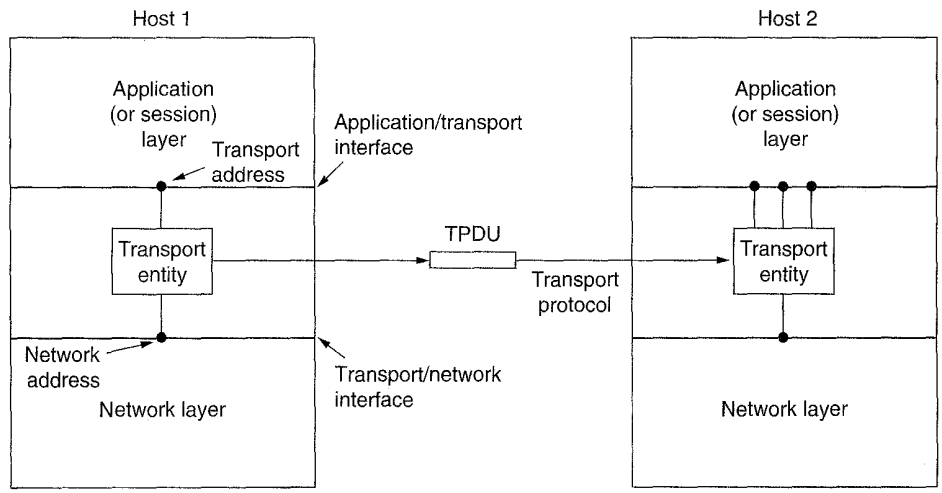


Fig. 6-1. The network, transport, and application layers.

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.

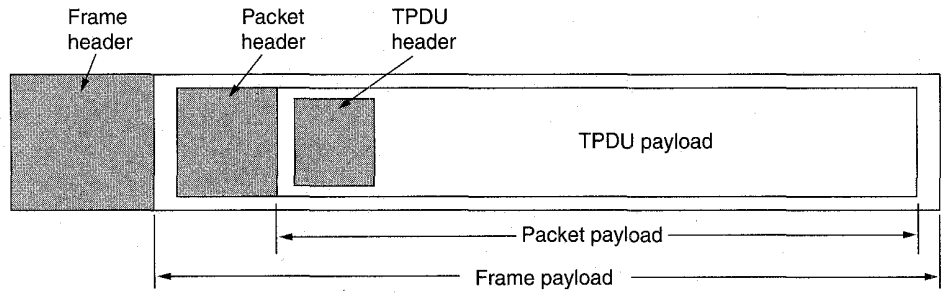


Fig. 6-4. Nesting of TPDUs, packets, and frames.

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.

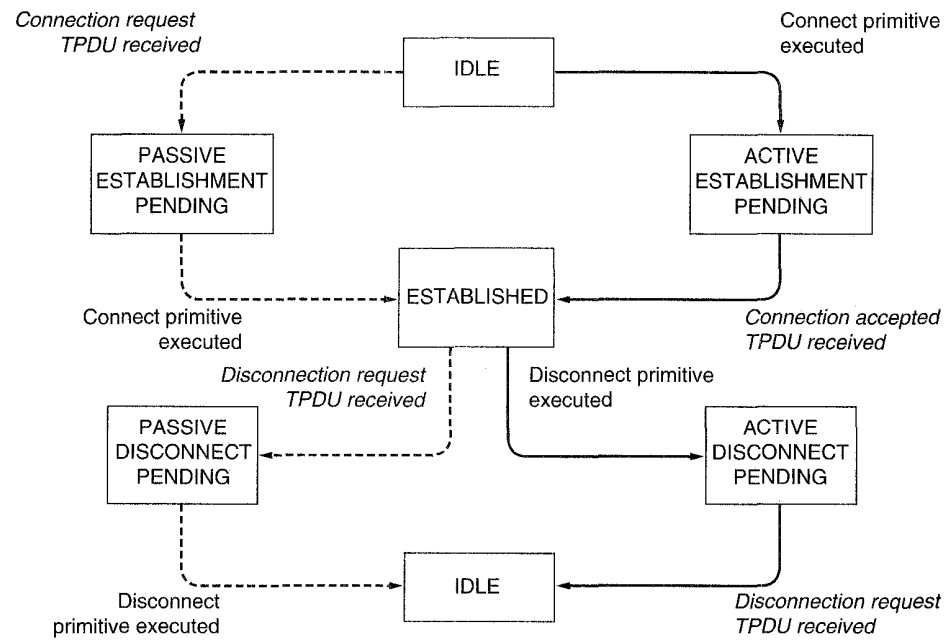


Fig. 6-5. A state diagram for a simple connection management scheme. Transitions labeled in italics are caused by packet arrivals. The solid lines show the client's state sequence. The dashed lines show the server's state sequence.

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.

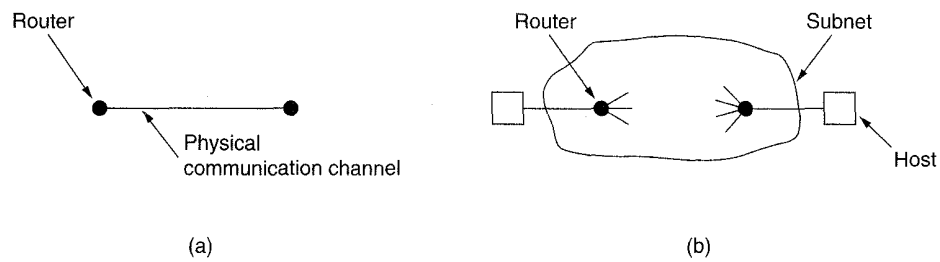


Fig. 6-7. (a) Environment of the data link layer. (b) Environment of the transport layer.

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.

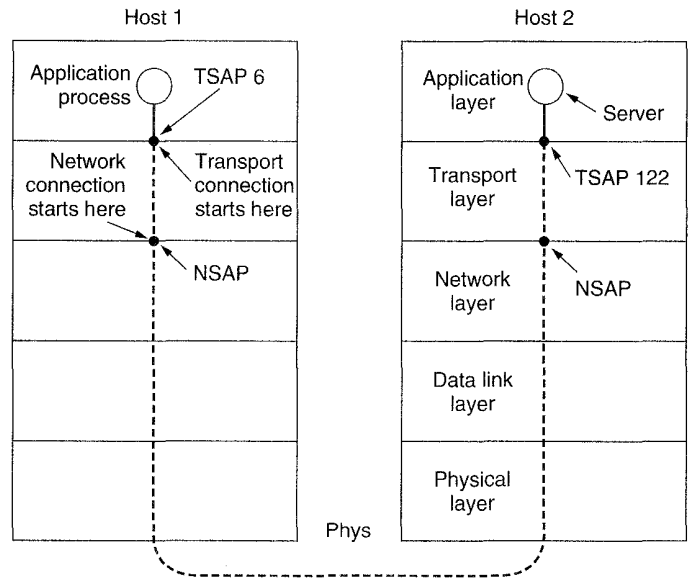


Fig. 6-8. TSAPs, NSAPs, and connections.

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).

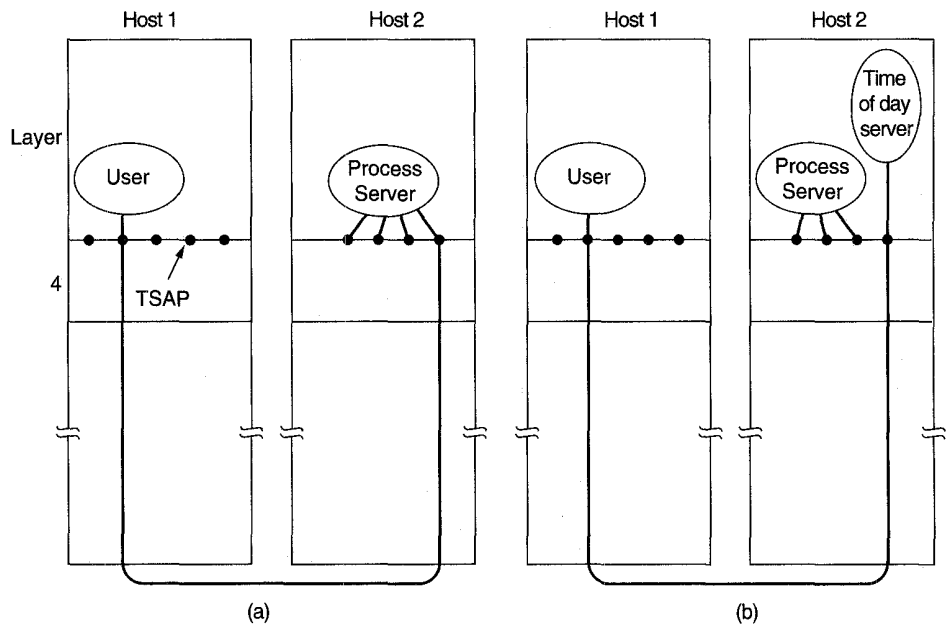


Fig. 6-9. How a user process in host 1 establishes a connection with a time-of-day server in host 2.

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.

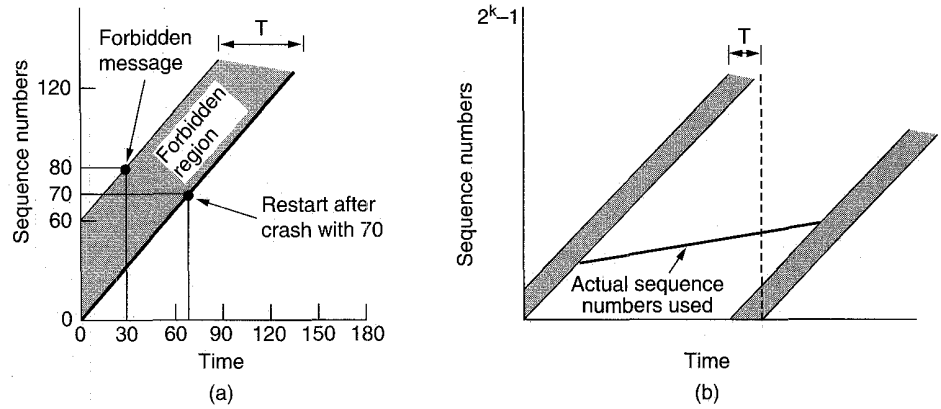


Fig. 6-10. (a) TPDUs may not enter the forbidden region. (b) The resynchronization problem.

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.

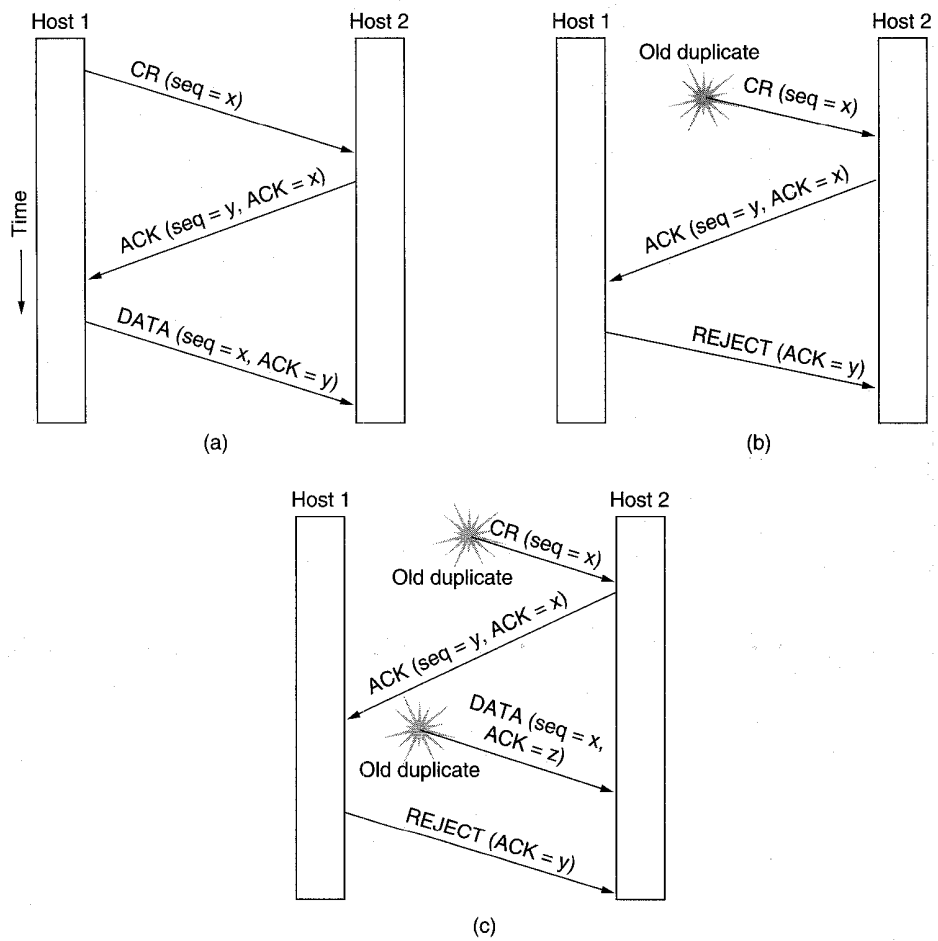


Fig. 6-11. Three protocol scenarios for establishing a connection using a three-way handshake. CR and ACC denote CONNECTION REQUEST and CONNECTION ACCEPTED, respectively. (a) Normal operation. (b) Old duplicate CONNECTION REQUEST appearing out of nowhere. (c) Duplicate CONNECTION REQUEST and duplicate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

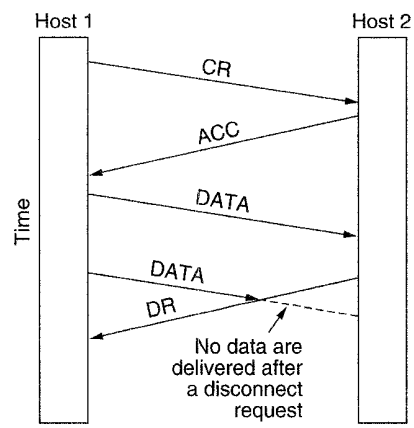


Fig. 6-12. Abrupt disconnection with loss of data.

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.

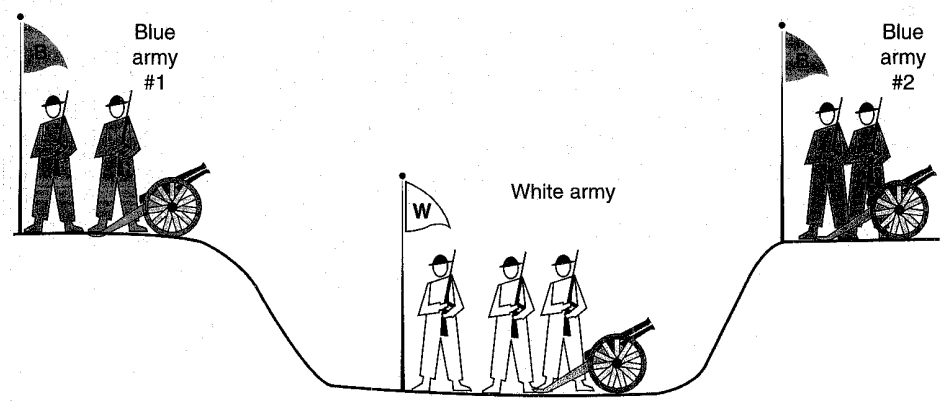


Fig. 6-13. The two-army problem.

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.

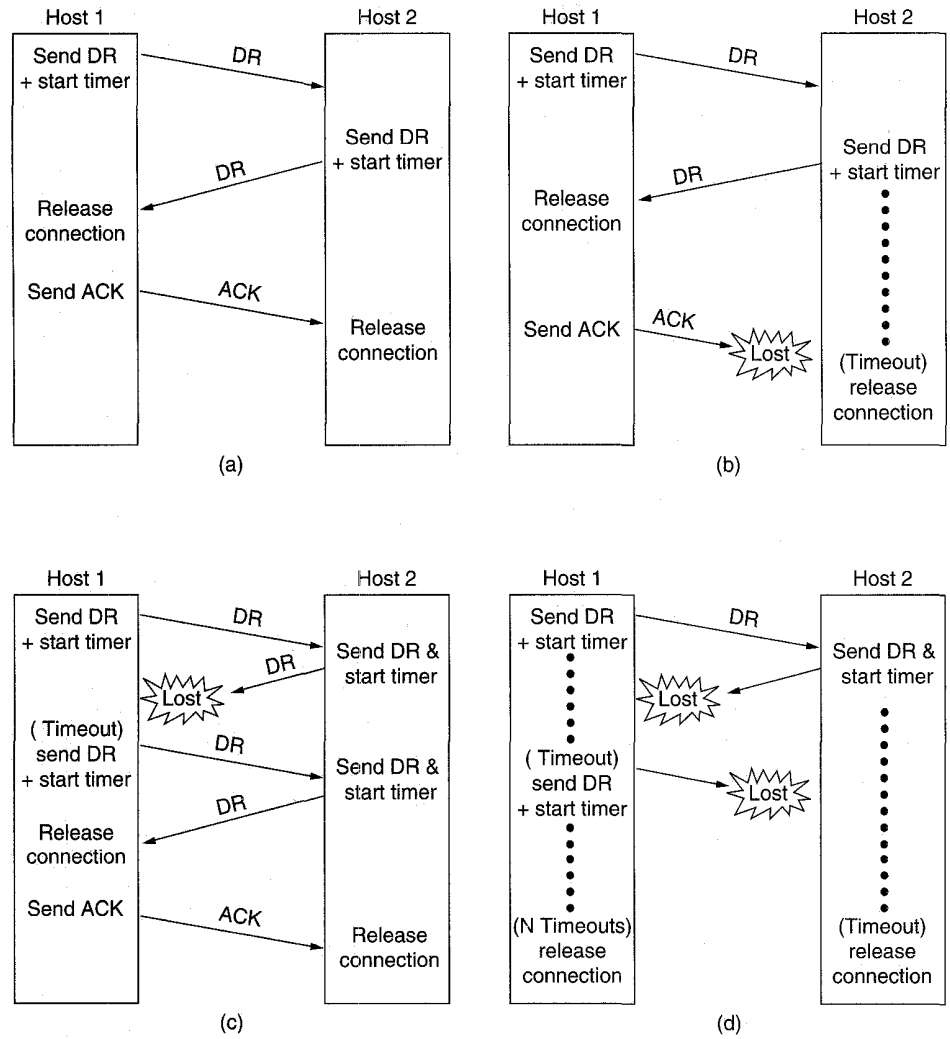


Fig. 6-14. Four protocol scenarios for releasing a connection. (a) Normal case of three-way handshake. (b) Final ACK lost. (c) Response lost. (d) Response lost and subsequent DRs lost.

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.

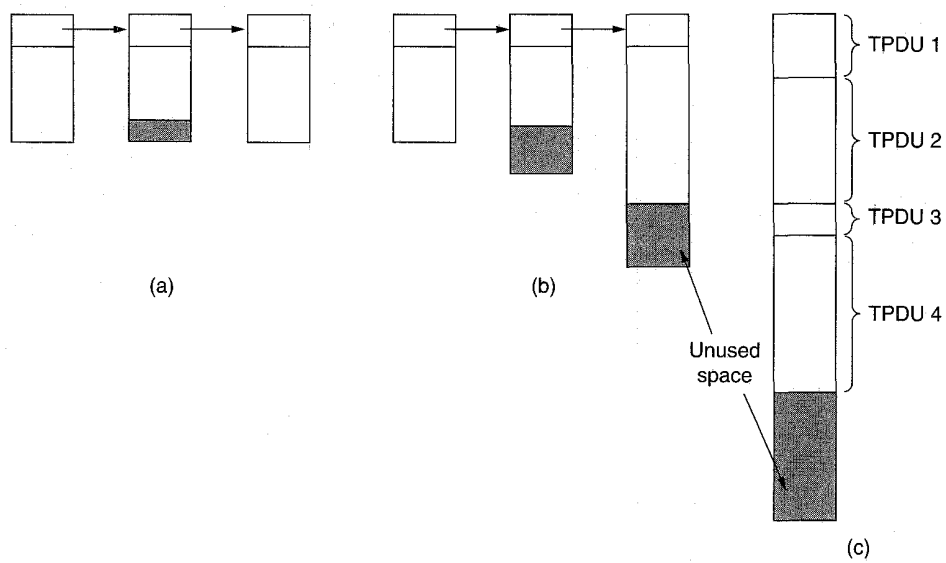


Fig. 6-15. (a) Chained fixed-size buffers. (b) Chained variable-size buffers. (c) One large circular buffer per connection.

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.

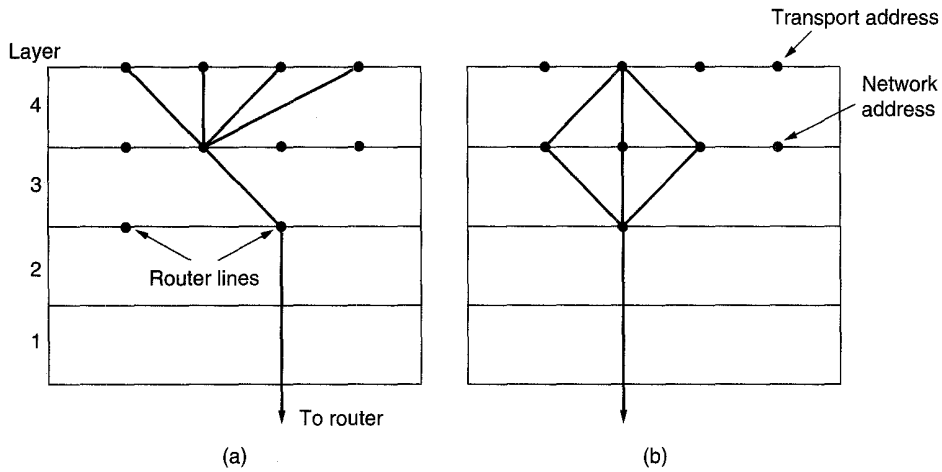


Fig. 6-17. (a) Upward multiplexing. (b) Downward multiplexing.

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

Strategy used by sending host	First ACK, then write			First write, then ACK		
	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

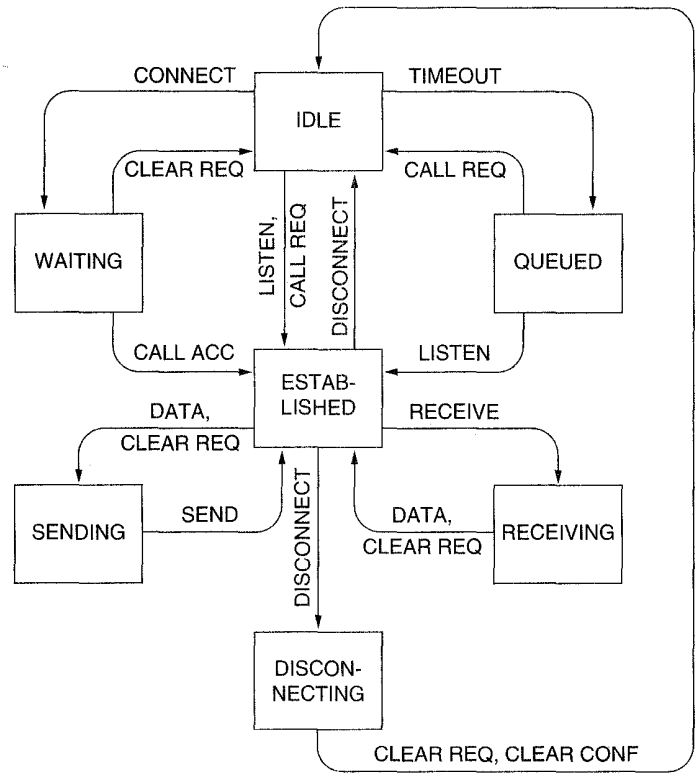


Fig. 6-22. The example protocol in graphical form. Transitions that leave the connection state unchanged have been omitted for simplicity.

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.

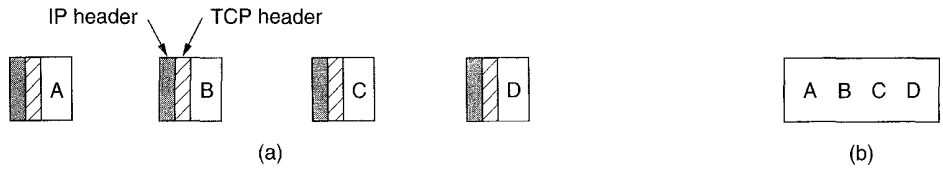


Fig. 6-23. (a) Four 512-byte segments sent as separate IP datagrams. (b) The 2048 bytes of data delivered to the application in a single READ call.

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.

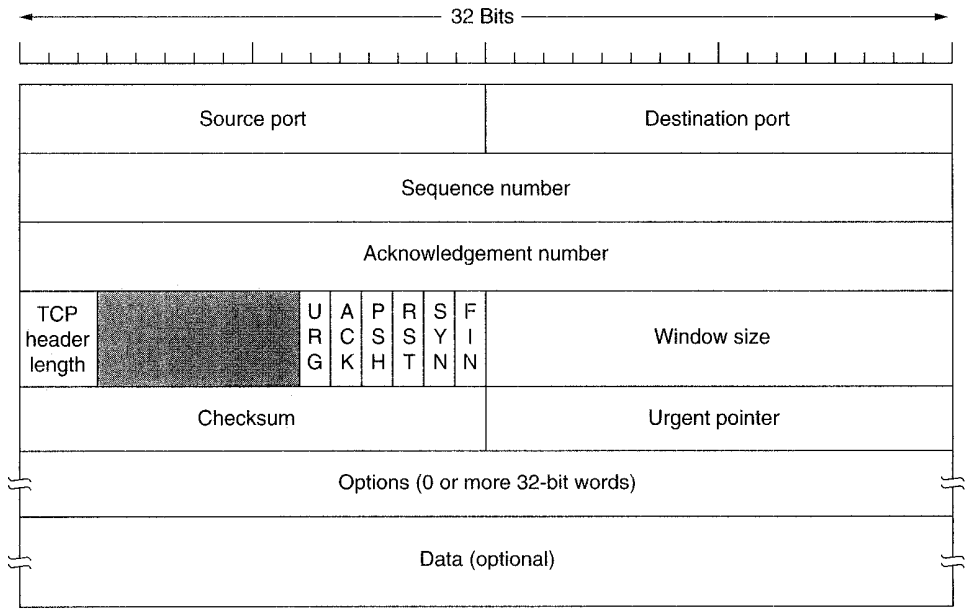


Fig. 6-24. The TCP header.

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.

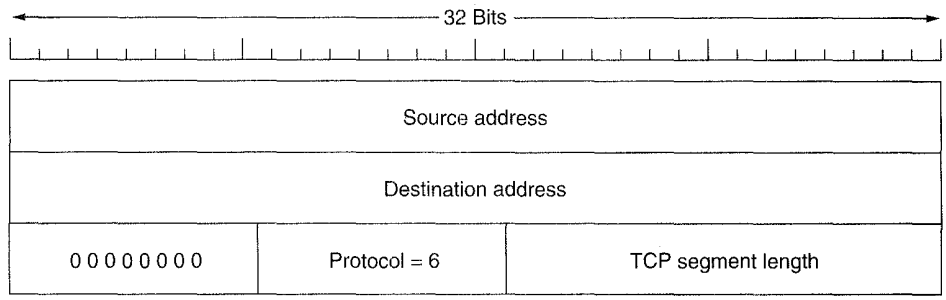


Fig. 6-25. The pseudoheader included in the TCP checksum.

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.

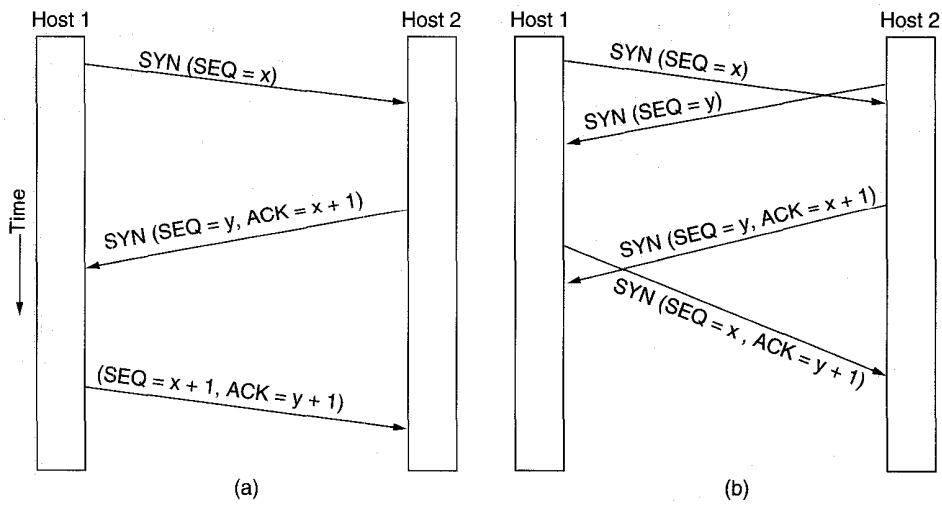


Fig. 6-26. (a) TCP connection establishment in the normal case. (b) Call collision.

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.

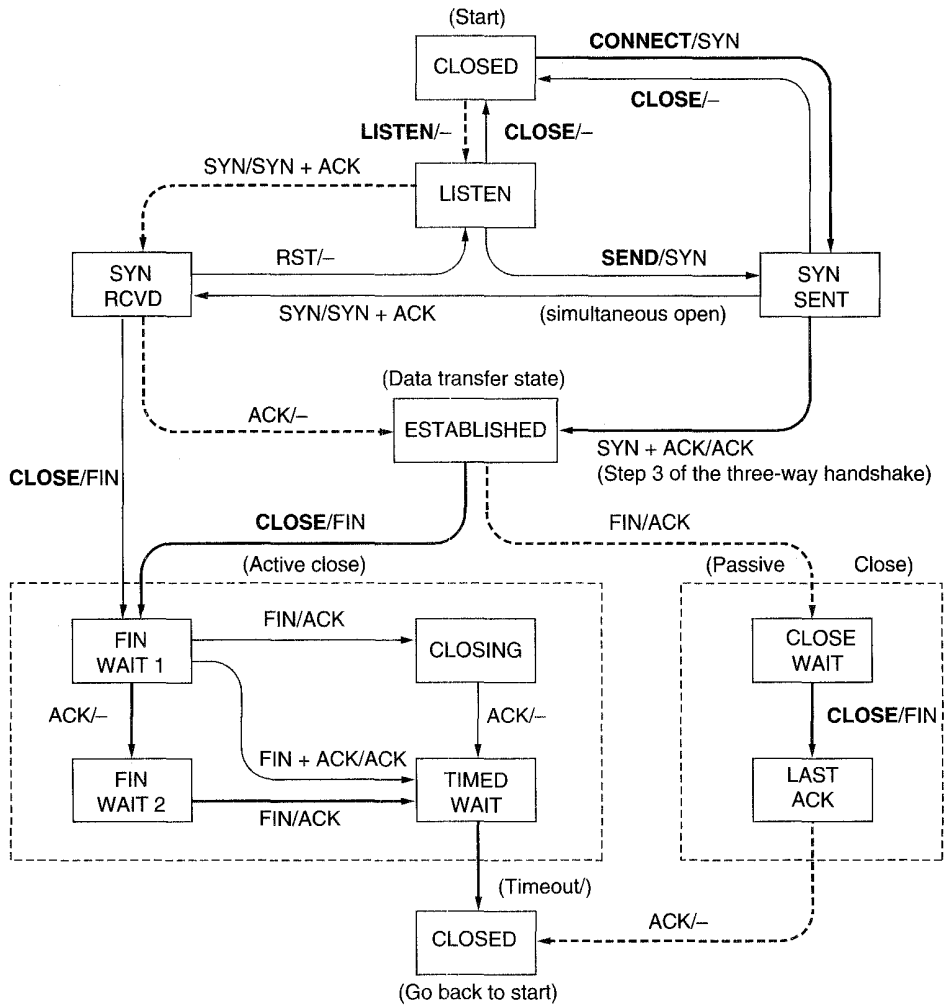


Fig. 6-28. TCP connection management finite state machine. The heavy solid line is the normal path for a client. The heavy dashed line is the normal path for a server. The light lines are unusual events.

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.

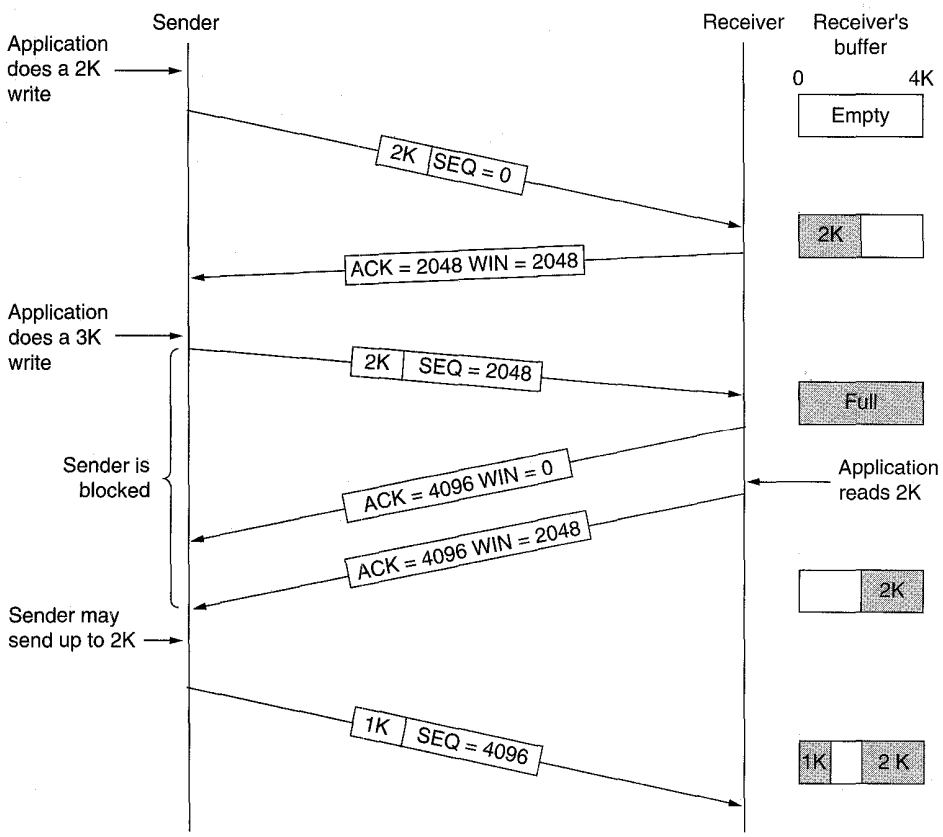


Fig. 6-29. Window management in TCP.

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.

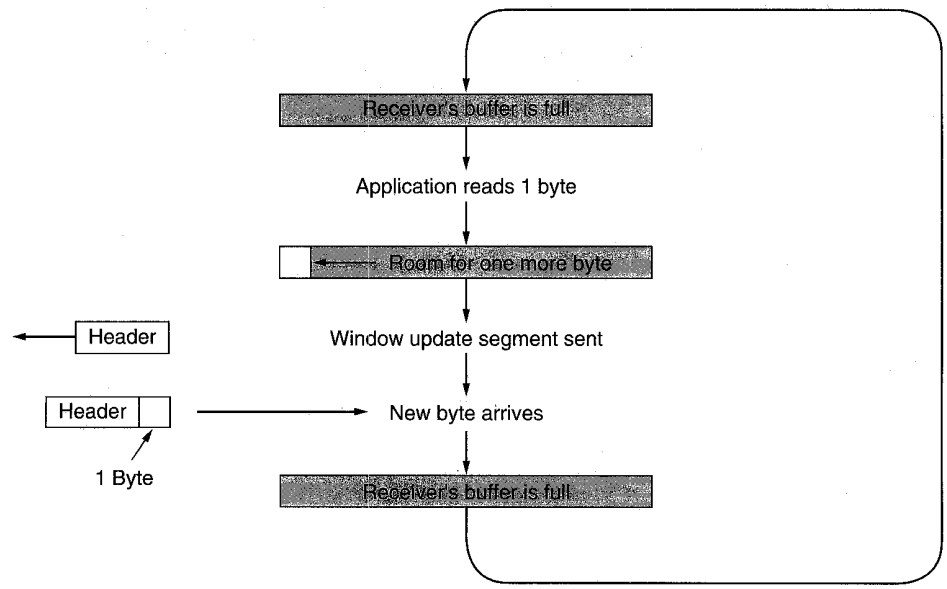


Fig. 6-30. Silly window syndrome.

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).

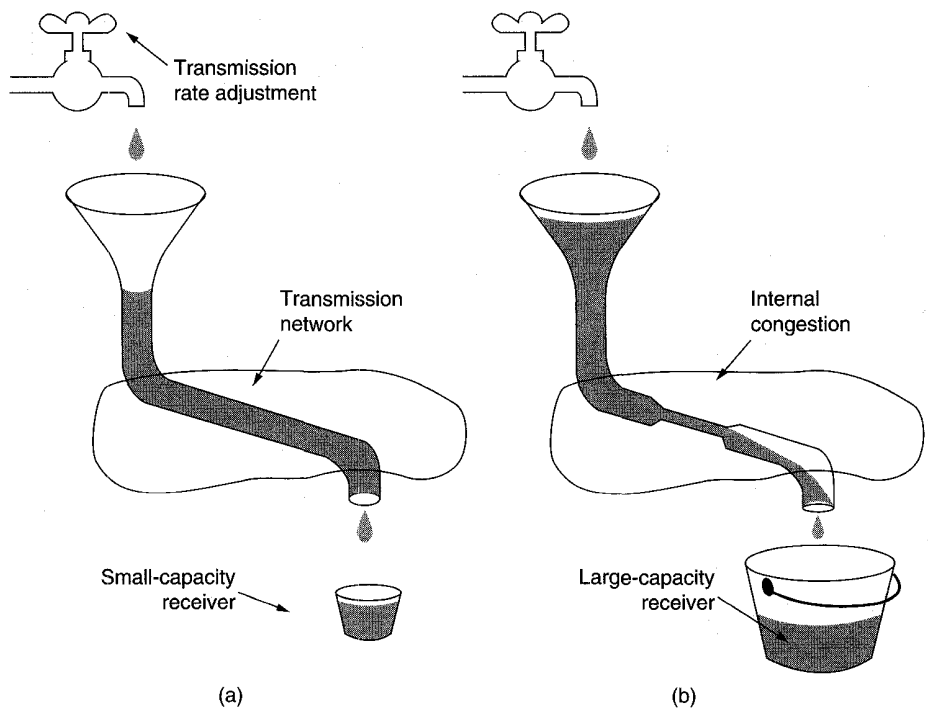


Fig. 6-31. (a) A fast network feeding a low-capacity receiver. (b) A slow network feeding a high-capacity receiver.

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.

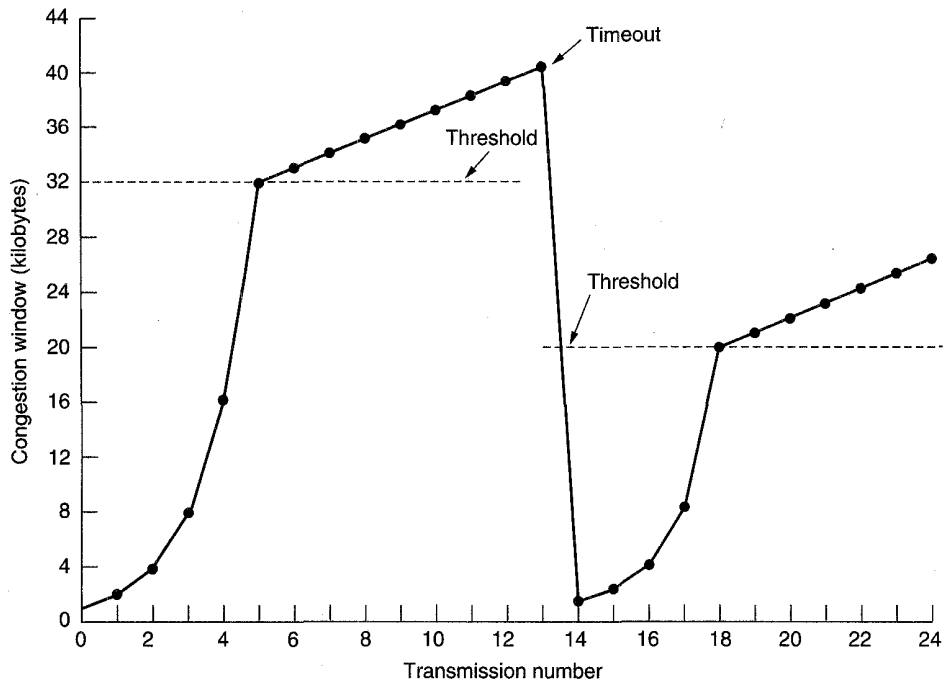


Fig. 6-32. An example of the Internet congestion algorithm.

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.

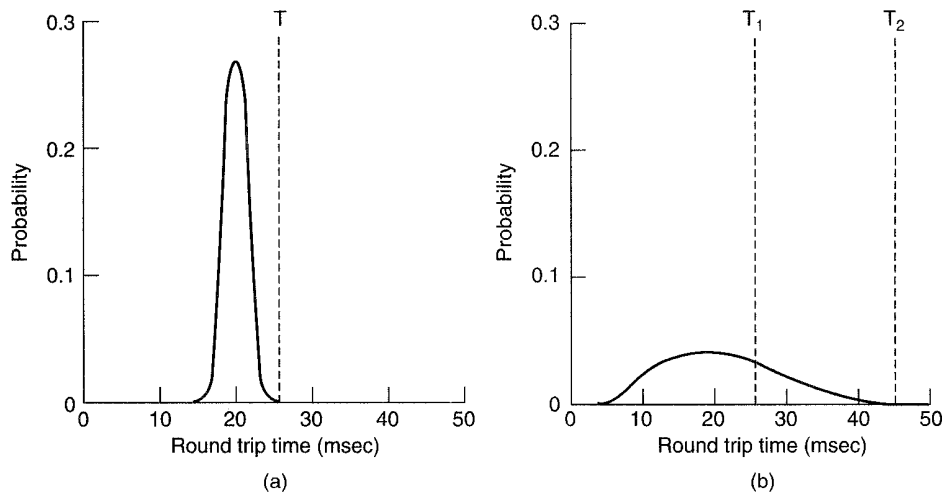


Fig. 6-33. (a) Probability density of acknowledgement arrival times in the data link layer. (b) Probability density of acknowledgement arrival times for TCP.

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 βRTT, 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.

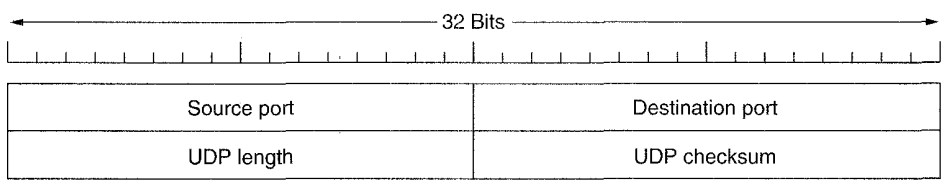


Fig. 6-34. The UDP header.

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.

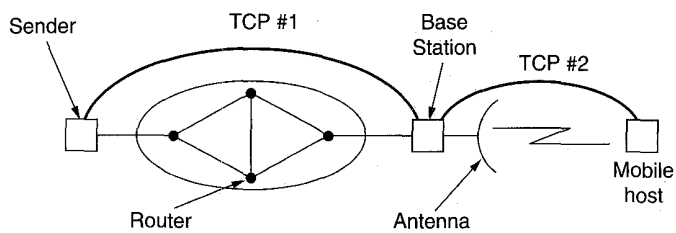


Fig. 6-35. Splitting a TCP connection into two connections.

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

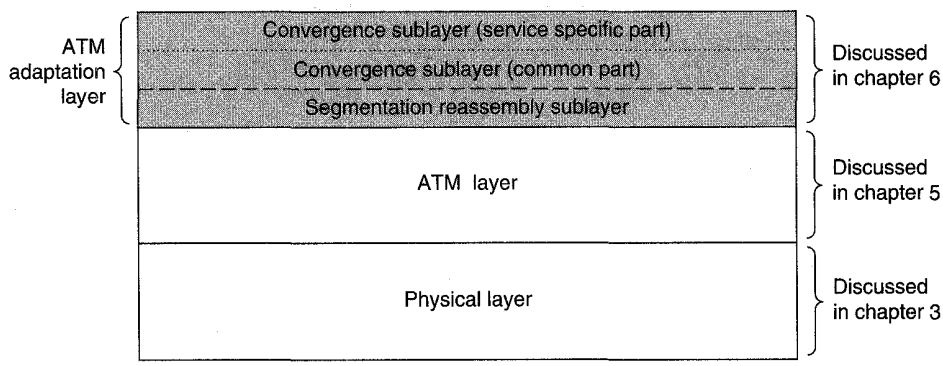


Fig. 6-37. The ATM model showing the ATM adaptation layer and its sublayers.

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,

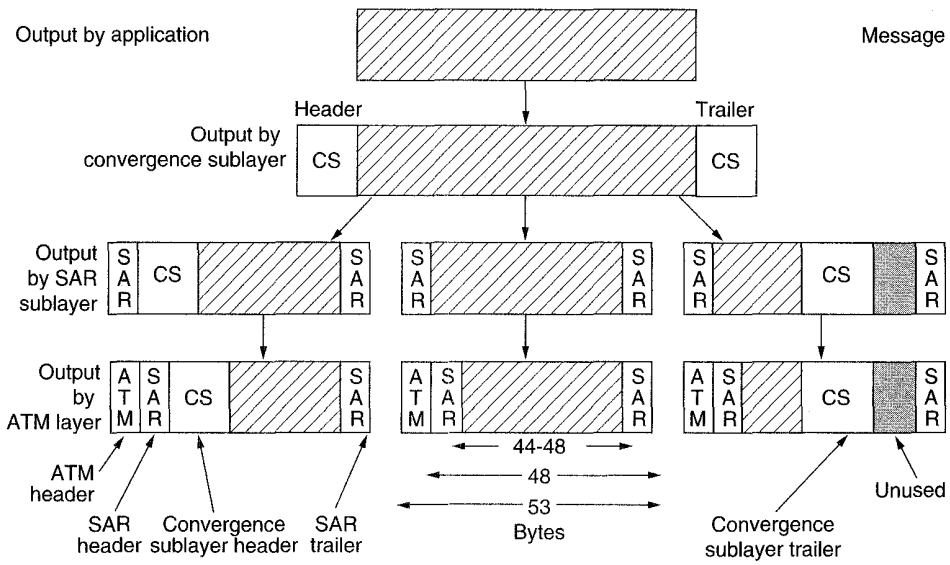


Fig. 6-38. The headers and trailers that can be added to a message in an ATM network.

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

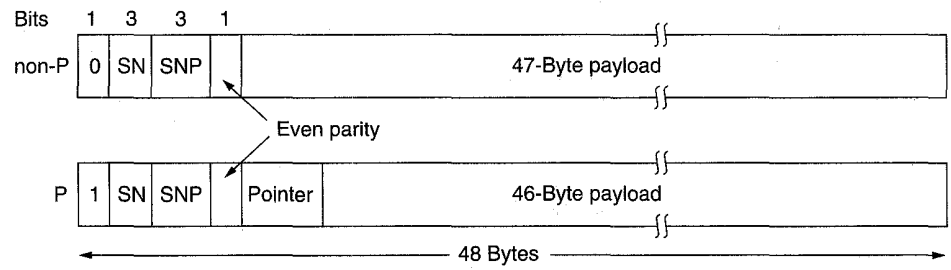


Fig. 6-39. The AAL 1 cell format.

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

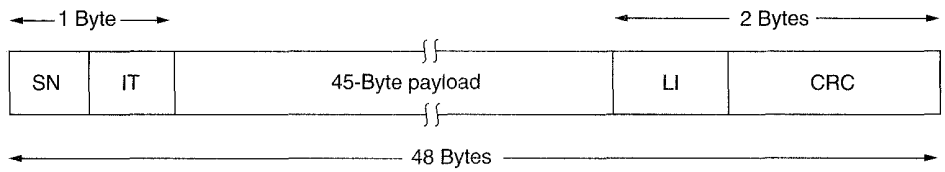


Fig. 6-40. The AAL 2 cell format.

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

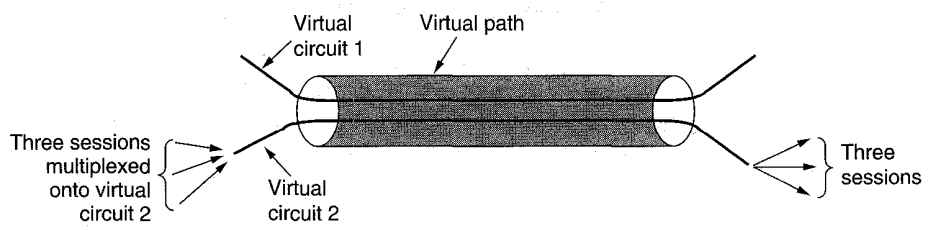


Fig. 6-41. Multiplexing of several sessions onto one virtual circuit.

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.

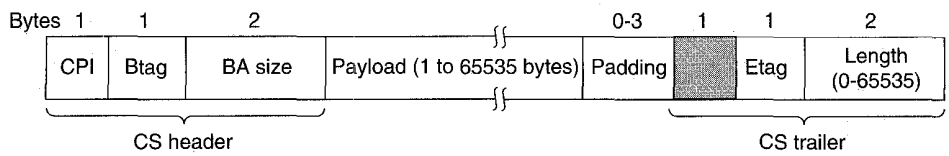


Fig. 6-42. AAL 3/4 convergence sublayer message format.

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.

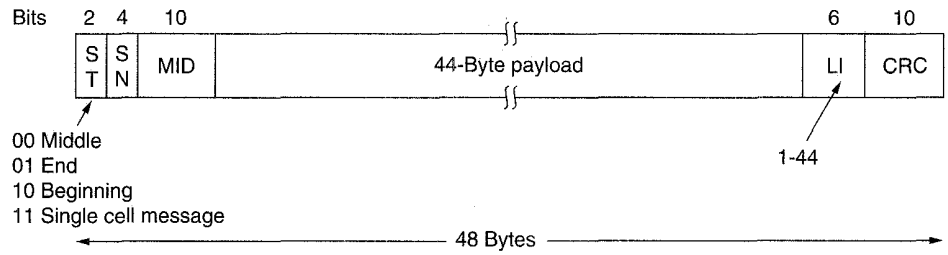


Fig. 6-43. The AAL 3/4 cell format.

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.

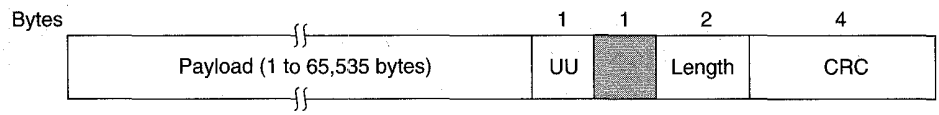


Fig. 6-44. AAL 5 convergence sublayer message format.

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

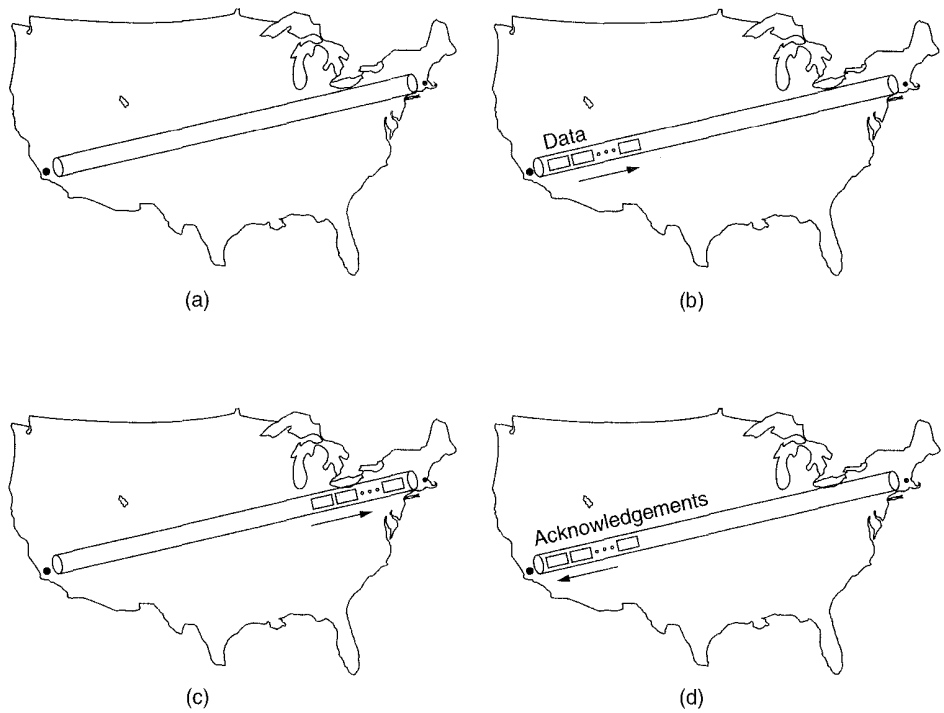


Fig. 6-46. The state of transmitting one megabit from San Diego to Boston. (a) At $t = 0$. (b) After 500 μsec. (c) After 20 msec. (d) After 40 msec.

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).

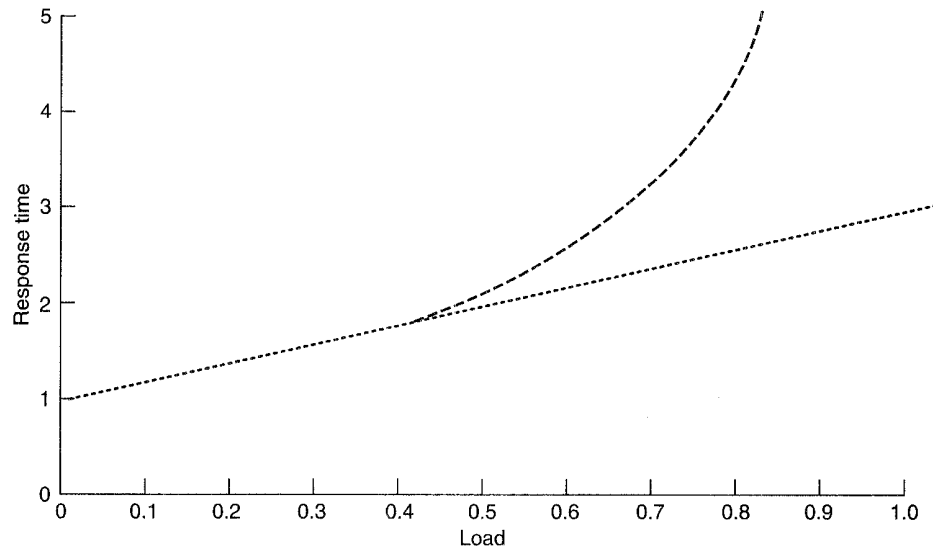


Fig. 6-47. Response as a function of load.

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.

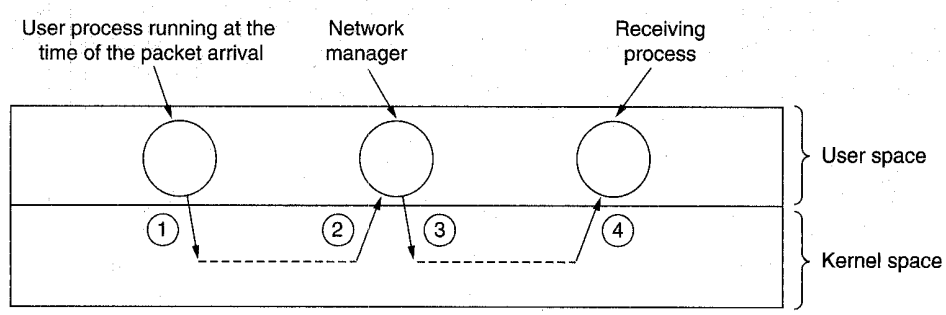


Fig. 6-48. Four context switches to handle one packet with a user-space network manager.

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.

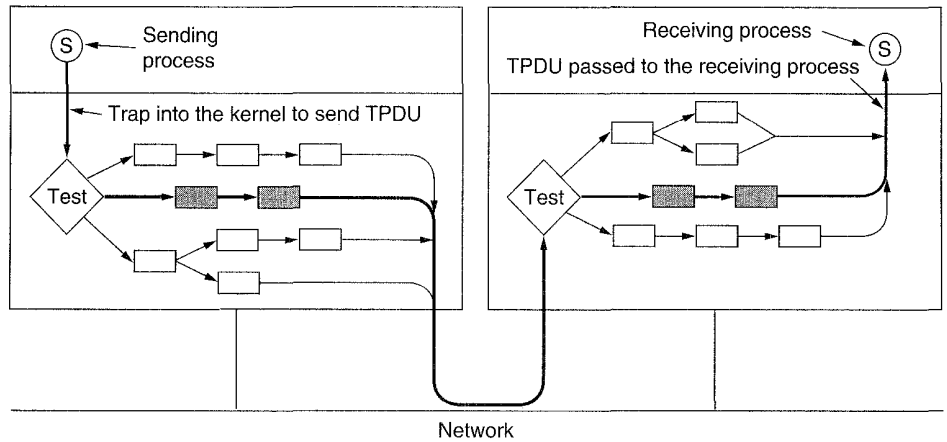


Fig. 6-49. The fast path from sender to receiver is shown with a heavy line. The processing steps on this path are shaded.

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.

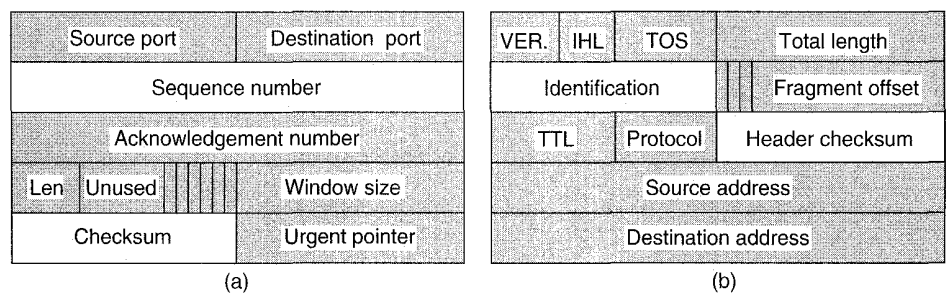


Fig. 6-50. (a) TCP header. (b) IP header. In both cases, the shaded fields are taken from the prototype without change.

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 $T + 15$.

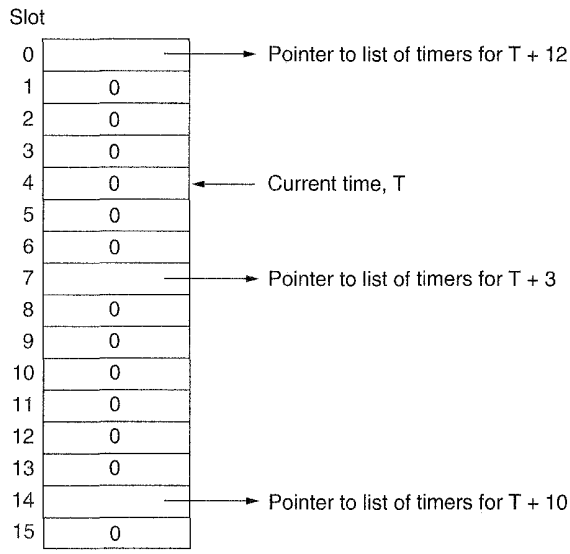


Fig. 6-51. A timing wheel.

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

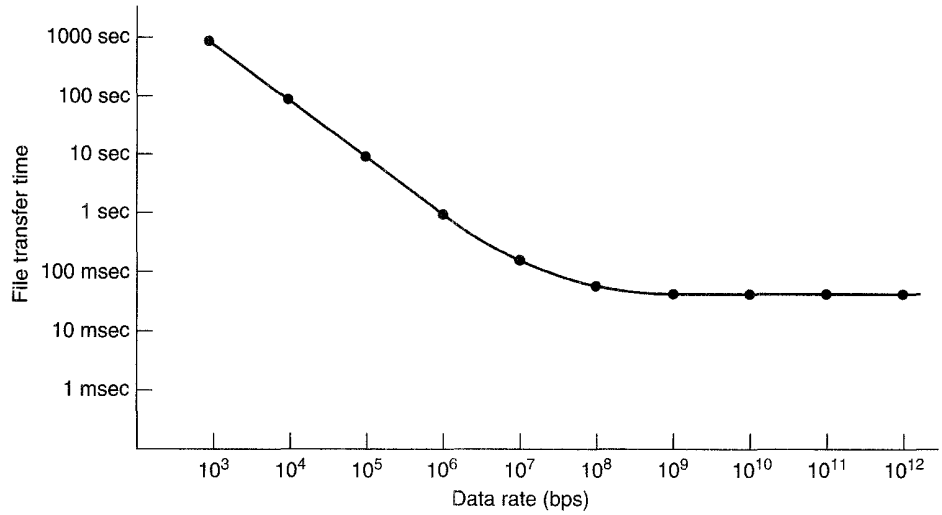


Fig. 6-52. Time to transfer and acknowledge a 1-megabit file over a 4000-km line.

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.

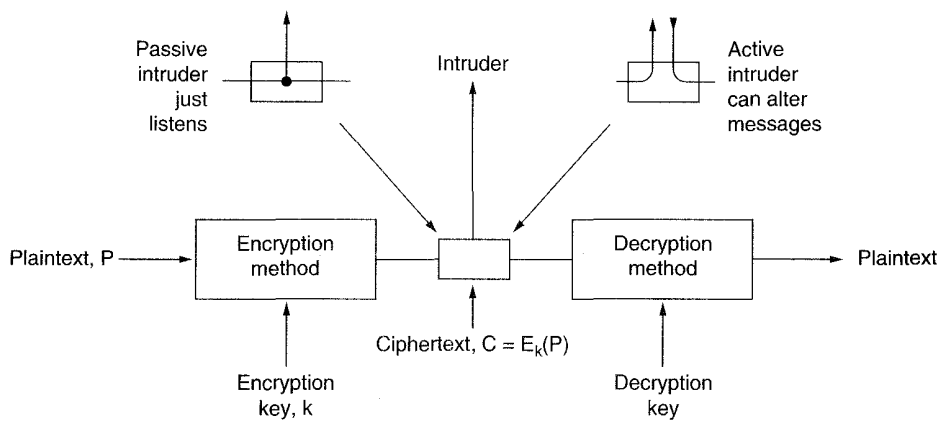


Fig. 7-2. The encryption model.

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.

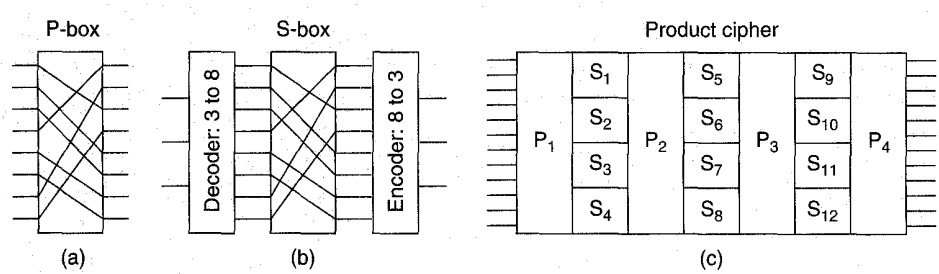


Fig. 7-4. Basic elements of product ciphers. (a) P-box. (b) S-box. (c) Product.

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.

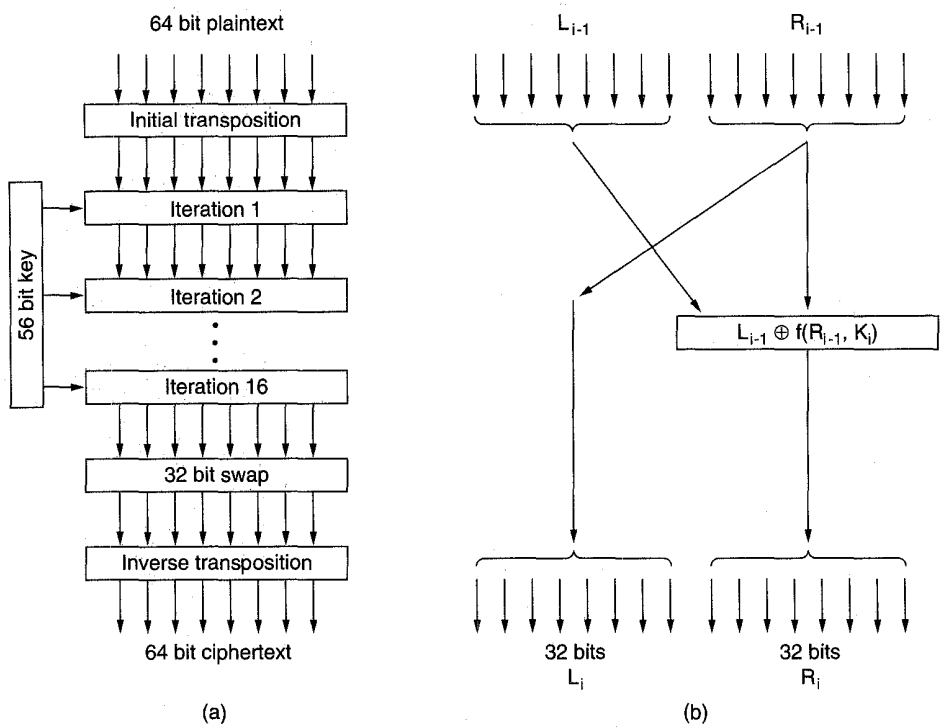


Fig. 7-5. The data encryption standard. (a) General outline. (b) Detail of one iteration.

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.

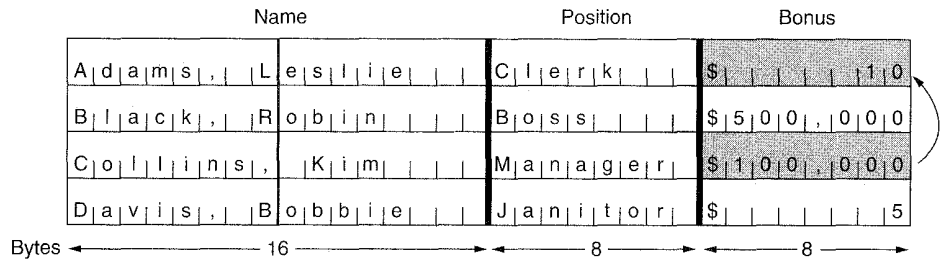


Fig. 7-6. The plaintext of a file encrypted as 16 DES blocks.

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

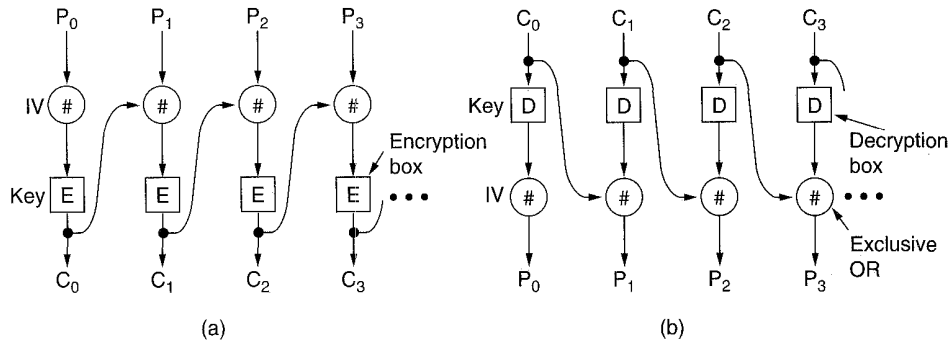


Fig. 7-7. Cipher block chaining

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.

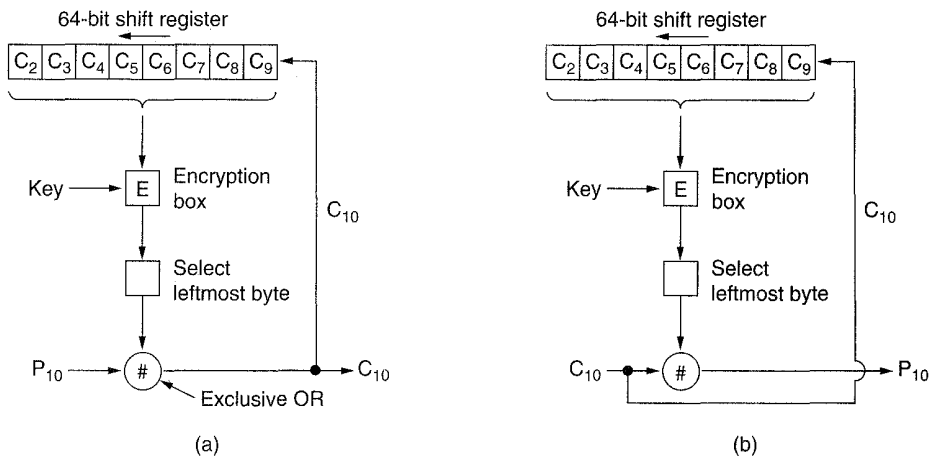


Fig. 7-8. Cipher feedback mode.

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

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

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.

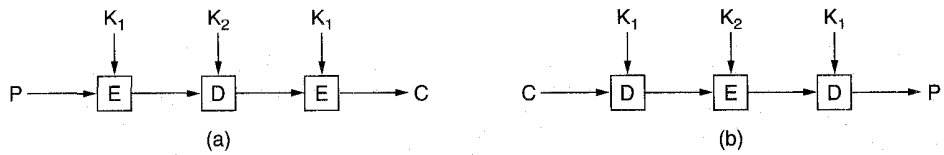


Fig. 7-9. Triple encryption using DES.

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.

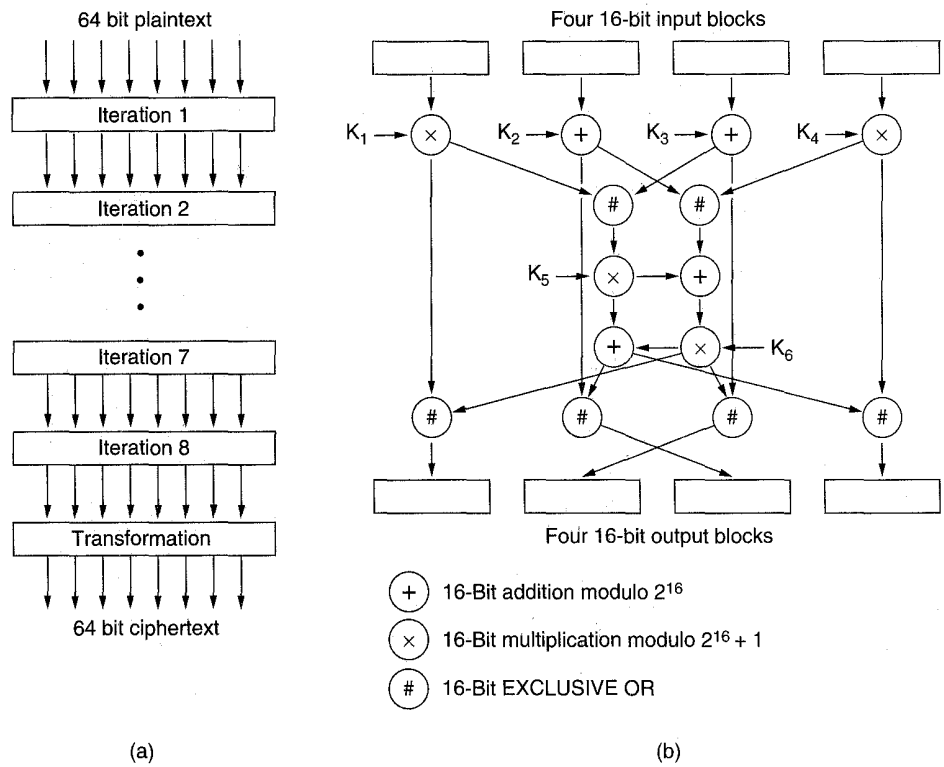


Fig. 7-10. (a) IDEA. (b) Detail of one iteration.

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 $C = P^e \pmod{n}$. To decrypt C, compute $P = C^d \pmod{n}$. 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 $7e = 1 \pmod{20}$, which yields $e = 3$. The ciphertext, C, for a plaintext message, P, is given by $C = P^3 \pmod{33}$. The ciphertext is decrypted by the receiver according to the rule $P = C^7 \pmod{33}$. 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.

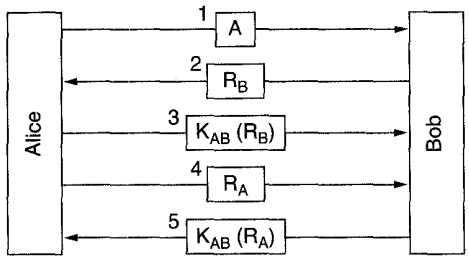


Fig. 7-12. Two-way authentication using a challenge-response protocol.

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.

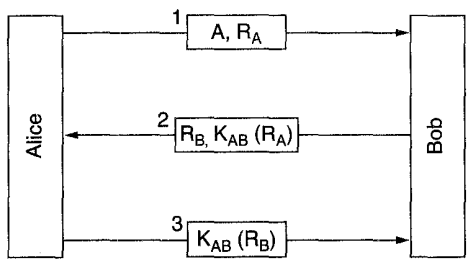


Fig. 7-13. A shortened two-way authentication protocol.

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 $K_{AB}(R_B)$.

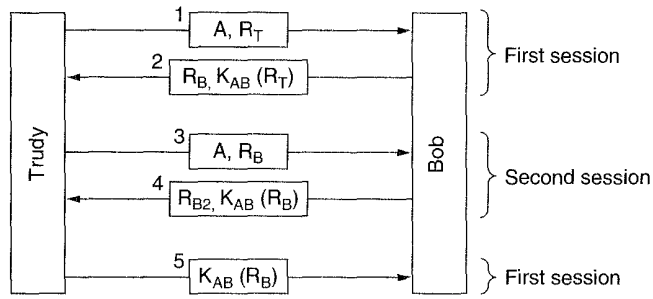


Fig. 7-14. The reflection attack.

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, $g^{xy} \bmod n$.

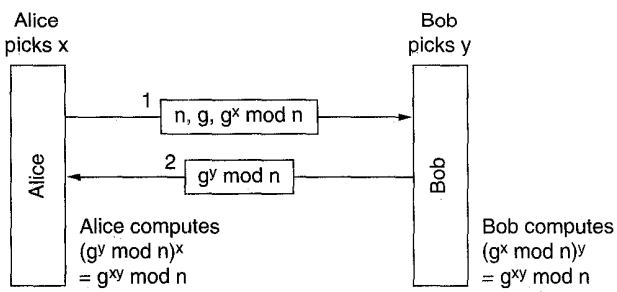


Fig. 7-15. The Diffie-Hellman key exchange.

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 mod 47 is 28. Bob's message to Alice is (17). Alice computes 17^8 mod 47, which is 4. Bob computes 28^{10} mod 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 mod 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.

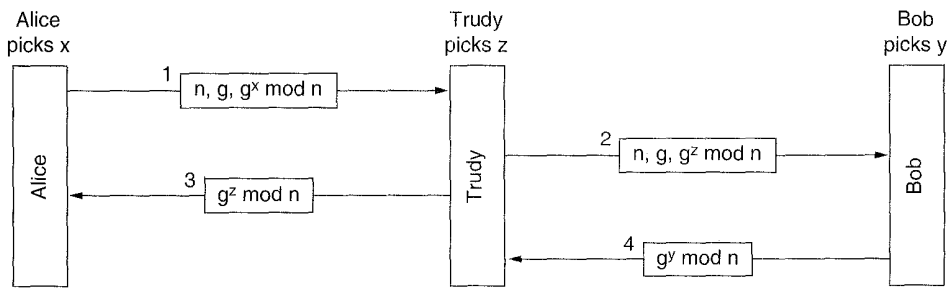


Fig. 7-16. The bucket brigade attack.

Now everybody does the modular arithmetic. Alice computes the secret key as g^{xz} mod n, and so does Trudy (for messages to Alice). Bob computes g^{yz} mod 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.

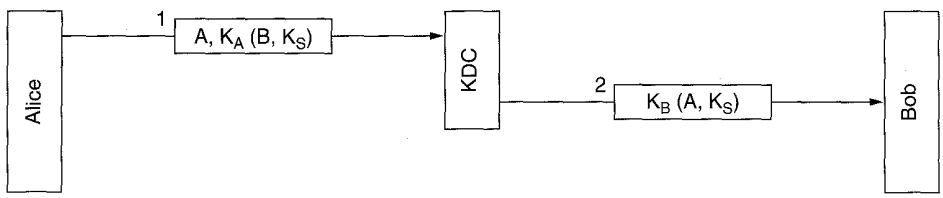


Fig. 7-17. A first attempt at an authentication protocol using a KDC.

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.

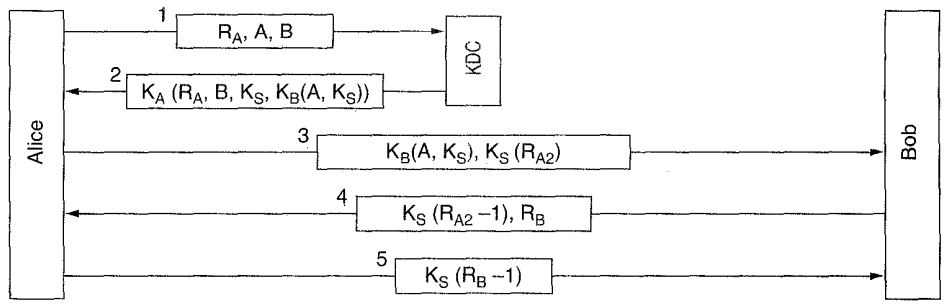


Fig. 7-18. The Needham-Schroeder authentication protocol.

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.

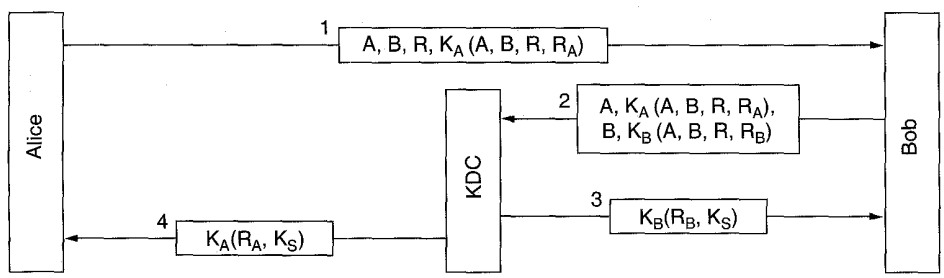


Fig. 7-19. The Otway-Rees authentication protocol (slightly simplified).

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 Rs 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

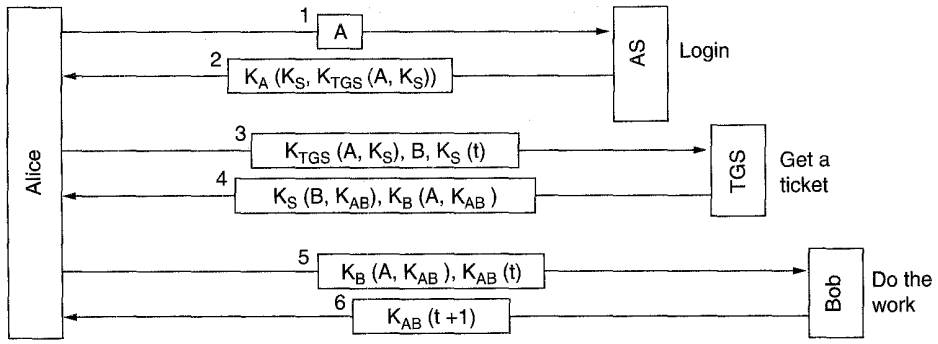


Fig. 7-20. The operation of Kerberos V4.

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.

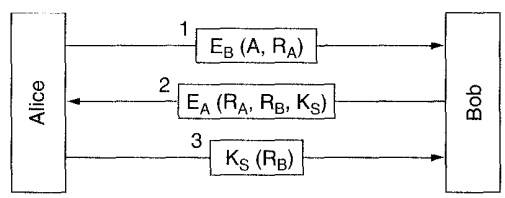


Fig. 7-21. Mutual authentication using public-key cryptography.

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.

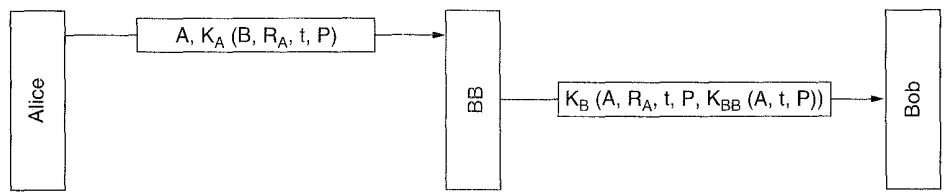


Fig. 7-22. Digital signatures with Big Brother.

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.

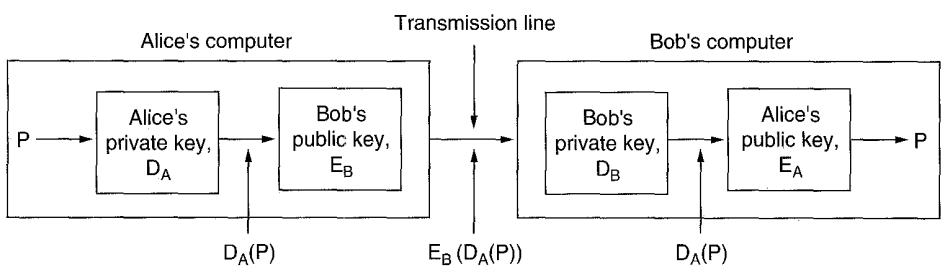


Fig. 7-23. Digital signatures using public-key cryptography.

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.

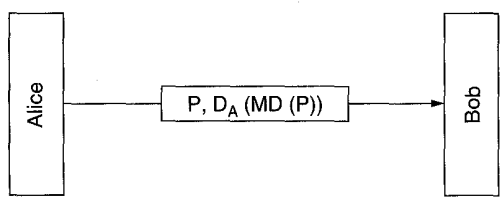


Fig. 7-24. Digital signatures using message digests.

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.

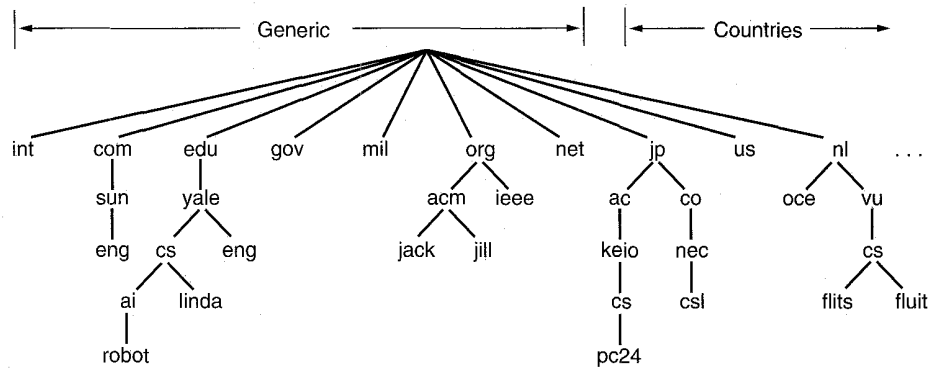


Fig. 7-25. A portion of the Internet domain name space.

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