The design of IPv6 presented a major opportunity to improve all of the features in IPv4 that fall short of what is now wanted. To develop a protocol that met all these requirements, IETF issued a call for proposals and discussion in RFC 1550. Twenty-one responses were initially received. 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 it with a completely different protocol.

One proposal was to run TCP over CLNP, the network layer protocol designed for OSI. With its 160-bit addresses, CLNP would have provided enough address space forever as it could give every molecule of water in the oceans enough addresses (roughly 2^5) to set up a small network. This choice would also 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.**

IPv6 meets IETF's 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 the other auxiliary Internet protocols, including TCP, UDP, ICMP, IGMP, OSPF, BGP, and DNS, with small modifications being required to deal with longer addresses. The main features of IPv6 are discussed below. More information about it can be found in RFCs 2460 through 2466.

First and foremost, IPv6 has longer addresses than IPv4. They are 128 bits long, which solves the problem that IPv6 set out to solve: providing 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 seven fields (versus 13 in IPv4). This change allows routers to process packets faster and thus improves throughput and delay. We will discuss the header shortly, too.

The third major improvement is better support for options. This change was essential with the new header because fields that previously were required are now optional (because they are not used so often). 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. These were later retrofitted to IPv4, however, so in the area of security the differences are not so great any more.

Finally, more attention has been paid to quality of service. Various half-hearted efforts to improve QoS have been made in the past, but now, with the growth of multimedia on the Internet, the sense of urgency is greater.

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 has already taken more than 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, given that the data link header usually indicates the network protocol for demultiplexing, so some routers may skip the check. For example, the Ethernet *Type* field has different values to indicate an IPv4 or an IPv6 payload. The discussions between the "Do it right" and "Make it fast" camps will no doubt be lengthy and vigorous.

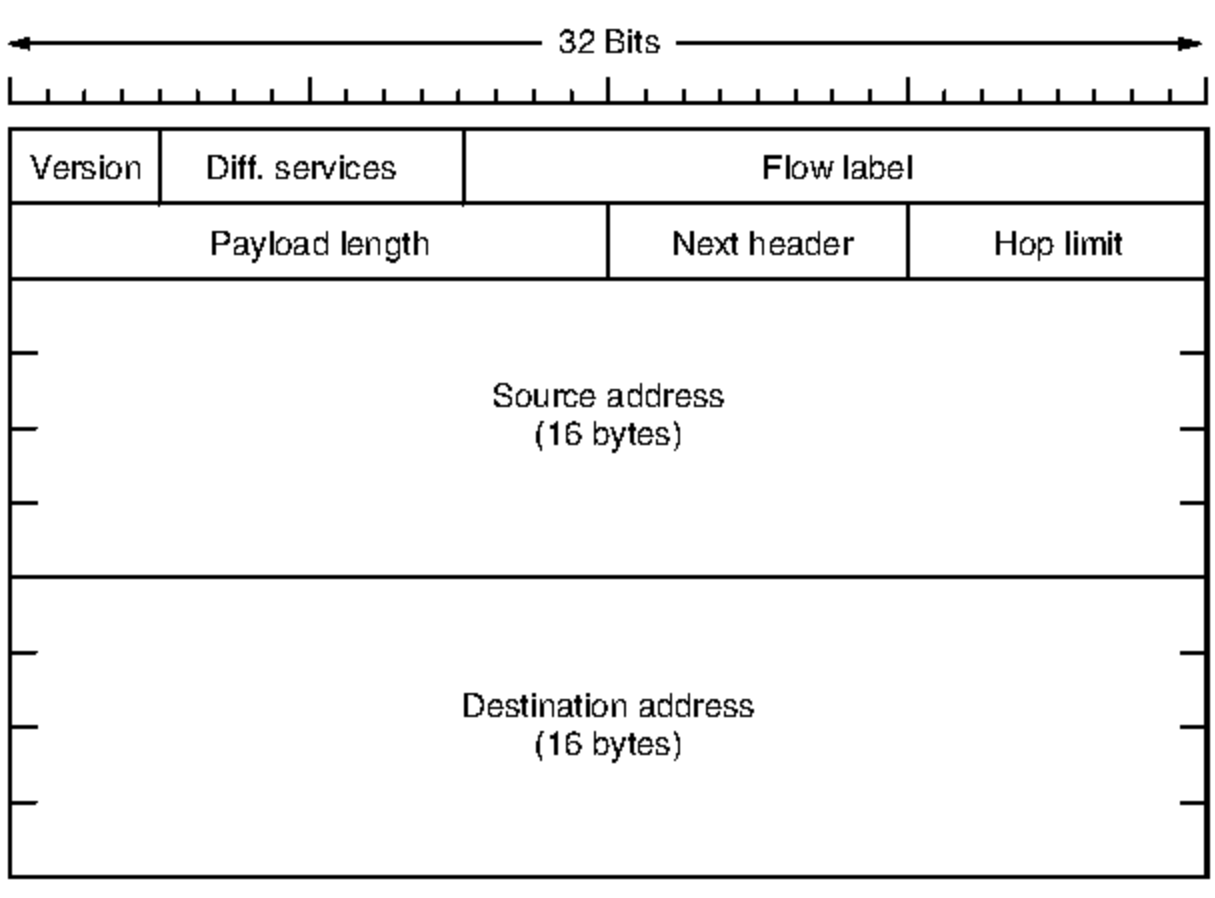


Figure 5-56. The IPv6 fixed header (required).

The *Differentiated services* field (originally called *Traffic class*) is used to distinguish the class of service for packets with different real-time delivery

requirements. It is used with the differentiated service architecture for quality of service in the same manner as the field of the same name in the IPv4 packet. Also, the low-order 2 bits are used to signal explicit congestion indications, again in the same way as with IPv4.

The *Flow label* field provides a way for a source and destination to mark groups of packets that have the same requirements and should be treated in the same way by the network, forming a pseudoconnection. For example, a stream of packets from one process on a certain source host to a process on a specific 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 network and the guarantees of a virtual-circuit network.

Each flow for quality of service purposes is designated by the source address, destination address, and flow number. This design means that up to 2^{20} flows may be active at the same time between a given pair of IP addresses. It also means that even if two flows coming from different hosts but with the same flow label 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 labels will be chosen randomly, rather than assigned sequentially starting at 1, so routers are expected 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). This change means the payload can now be 65,535 bytes instead of a mere 65,515 bytes.

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, follow 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. Still another faction wanted variable-sized addresses. After much debate and more than a few words

unprintable in an academic textbook, it was decided that fixed-length 16-byte addresses were the best compromise.

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 zero bits 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 3194, Durand and 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 entire 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 instructive to compare the IPv4 header (Fig. 5-46) 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 are expected to dynamically determine the packet size to use. They do this using the path MTU discovery procedure we described in Sec. 5.5.5. In brief, when a host sends an IPv6 packet that is too large, instead of fragmenting it, the router that is unable to forward it drops the packet and sends an error message back to the

sending host. 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. Also, the minimum-size packet that routers must be able to forward has been raised from 576 to 1280 bytes to allow 1024 bytes of data and many headers.

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 deemed 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—is met by this design.

Extension Headers

Some of the missing IPv4 fields are occasionally still needed, so IPv6 introduces the concept of (optional) **extension headers**. 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. 5-57. 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
Destination options	Additional information for the destination
Routing	Loose list of routers to visit
Fragmentation	Management of datagram fragments
Authentication	Verification of the sender's identity
Encrypted security payload	Information about the encrypted contents

Figure 5-57. IPv6 extension headers.

Some of the headers have a fixed format; others contain a variable number of variable-length options. 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 discard the packet but 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 64 KB. The format of this header is shown in Fig. 5-58. When it is used, the *Payload length* field in the fixed header is set to 0.

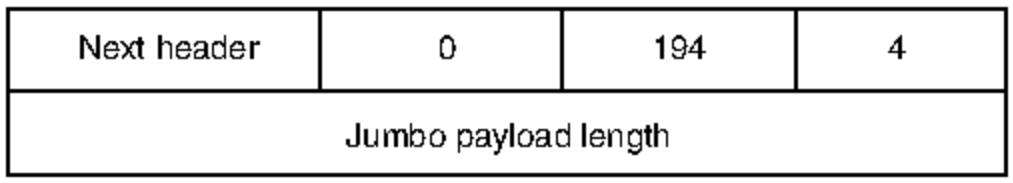


Figure 5-58. The hop-by-hop extension header for large datagrams (jumbograms).

As with all extension headers, this one starts 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. All extensions begin this way.

The next 2 bytes indicate that this option defines the datagram size (code 194) and that the size is a 4-byte number. The last 4 bytes give the size of the datagram. Sizes less than 65,536 bytes 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 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.

The routing header lists one or more routers that must be visited on the way to the destination. It is very similar to the IPv4 loose source routing in that all addresses listed must be visited in order, but other routers not listed may be visited in between. The format of the routing header is shown in Fig. 5-59.

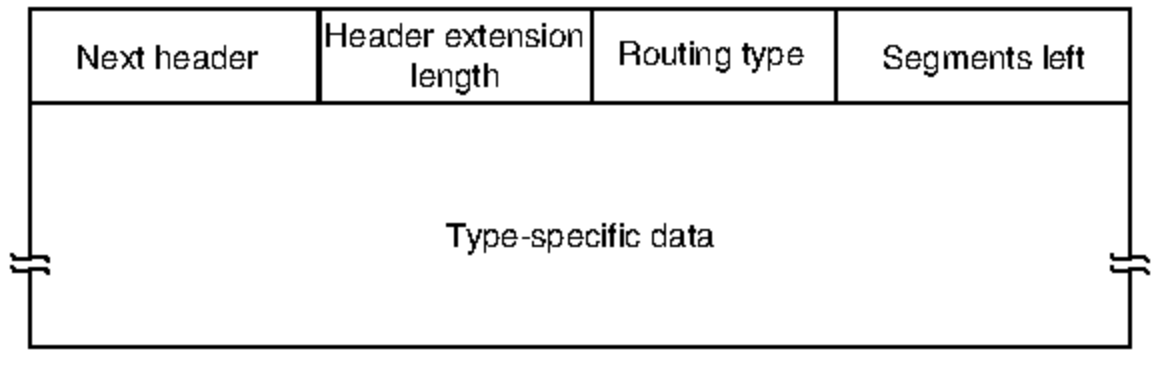


Figure 5-59. The extension header for routing.

The first 4 bytes of the routing extension header contain four 1-byte integers. The *Next header* and *Header extension length* fields were described above. The *Routing type* field gives the format of the rest of the header. Type 0 says that a reserved 32-bit word follows the first word, followed by some number of IPv6 addresses. Other types may be invented in the future, as needed. Finally, the *Segments left* field keeps track of how many of the addresses in the list have not yet been visited. It is decremented every time one is visited. When it hits 0, the packet is on its own with no more guidance about what route to follow. Usually, at this point it is so close to the destination that the best route is obvious.

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. This change is a major philosophical break with the original IP, but in keeping with current practice for IPv4. Plus, 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 error 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. 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 the cryptographic techniques that we will describe in Chap. 8 to accomplish their missions.

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, to say the least. We will summarize a few of these briefly below. For all the gory details, see the RFCs.

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 too long a time and 65,535 hops is far, 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 the 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 there with the same IPv6 address, or does it have to use a scheme with home agents? 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 essential. The war was about where to put it 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 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 very stringent export laws concerning cryptography. Some, notably France and Iraq, also restrict its use domestically, so that people cannot have secrets from the government. 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 point on which there was no controversy is that no one expects the IPv4 Internet to be turned off on a Sunday evening and come back up as an IPv6 Internet Monday morning. Instead, isolated "islands" of IPv6 will be converted, initially communicating via tunnels, as we showed in Sec. 5.5.3. As the IPv6 islands grow, they will merge into bigger islands. Eventually, all the islands will merge, and the Internet will be fully converted.

At least, that was the plan. Deployment has proved the Achilles heel of IPv6. It remains little used, even though all major operating systems fully support it. Most deployments are new situations in which a network operator—for example, a mobile phone operator— needs a large number of IP addresses. Many strategies have been defined to help ease the transition. Among them are ways to automatically configure the tunnels that carry IPv6 over the IPv4 Internet, and ways for hosts to automatically find the tunnel endpoints. Dual-stack hosts have an IPv4 and an IPv6 implementation so that they can select which protocol to use depending on the destination of the packet. These strategies will streamline the substantial deployment that seems inevitable when IPv4 addresses are exhausted. For more information about IPv6, see Davies (2008).

5.6.4 Internet Control Protocols

In addition to IP, which is used for data transfer, the Internet has several companion control protocols that are used in the network layer. They include ICMP, ARP, and DHCP. In this section, we will look at each of these in turn, describing the versions that correspond to IPv4 because they are the protocols that are in common use. ICMP and DHCP have similar versions for IPv6; the equivalent of ARP is called NDP (Neighbor Discovery Protocol) for IPv6.

IMCP—The Internet Control Message Protocol

The operation of the Internet is monitored closely by the routers. When something unexpected occurs during packet processing at a router, the event is reported to the sender by the **ICMP** (**Internet Control Message Protocol**). ICMP is also used to test the Internet. About a dozen types of ICMP messages are defined. Each ICMP message type is carried encapsulated in an IP packet. The most important ones are listed in Fig. 5-60.

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

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 and echo reply	Check if a machine is alive
Timestamp request/reply	Same as Echo, but with timestamp
Router advertisement/solicitation	Find a nearby router

Figure 5-60. The principal ICMP message types.

The TIME EXCEEDED message is sent when a packet is dropped because its *TtL (Time to live)* counter has reached zero. This event is a symptom that packets are looping, or that the counter values are being set too low.

One clever use of this error message is the **traceroute** utility that was developed by Van Jacobson in 1987. Traceroute finds the routers along the path from the host to a destination IP address. It finds this information without any kind of privileged network support. The method is simply to send a sequence of packets to the destination, first with a TtL of 1, then a TtL of 2, 3, and so on. The counters on these packets will reach zero at successive routers along the path. These routers will each obediently send a TIME EXCEEDED message back to the host. From those messages, the host can determine the IP addresses of the routers along the path, as well as keep statistics and timings on parts of the path. It is not what the TIME EXCEEDED message was intended for, but it is perhaps the most useful network debugging tool of all time.

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 long ago 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 anymore because when congestion occurs, these packets tend to add more fuel to the fire and it is unclear how to respond to them. Congestion control in the Internet is now done largely by taking action in the transport layer, using packet losses as a congestion signal; we will study it in detail in Chap. 6.

The REDIRECT message is used when a router notices that a packet seems to be routed incorrectly. It is used by the router to tell the sending host to update to a better route.

The ECHO and ECHO REPLY messages are sent by hosts to see if a given destination is reachable and currently alive. Upon receiving the ECHO message,

the destination is expected to send back an ECHO REPLY message. These messages are used in the **ping** utility that checks if a host is up and on the Internet.

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 can be used to measure network performance.

The ROUTER ADVERTISEMENT and ROUTER SOLICITATION messages are used to let hosts find nearby routers. A host needs to learn the IP address of at least one router to be able to send packets off the local network.

In addition to these messages, others have been defined. The online list is now kept at *www.iana.org/assignments/icmp-parameters*.

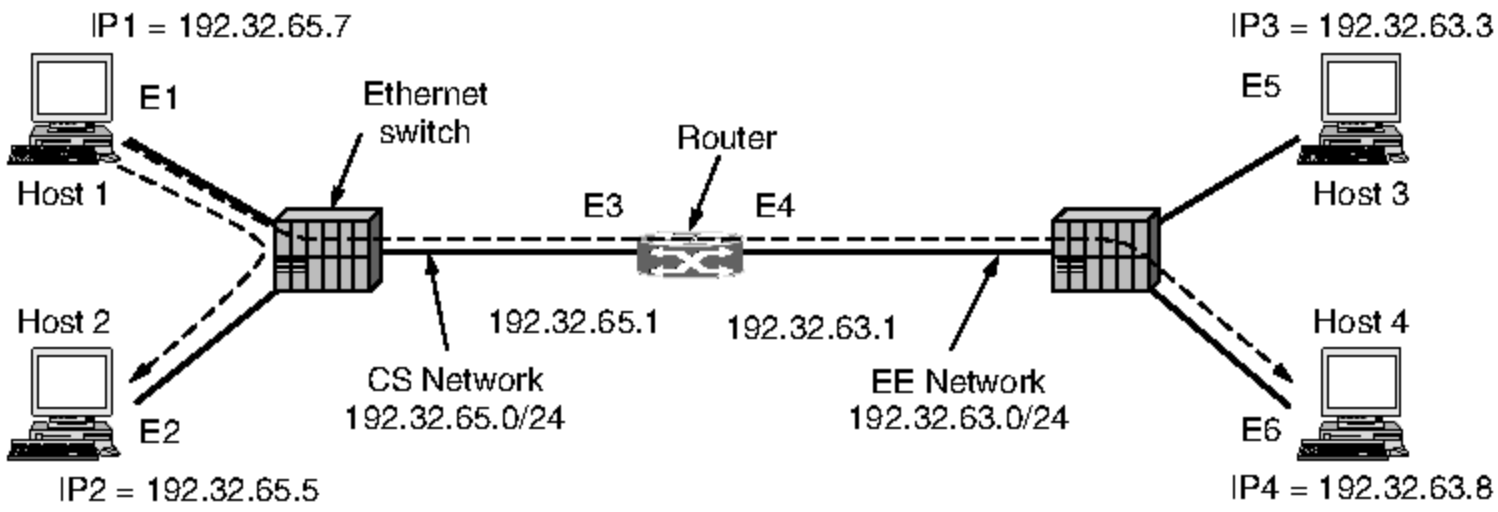
ARP—The Address Resolution Protocol

Although every machine on the Internet has one or more IP addresses, these addresses are not sufficient for sending packets. Data link layer NICs (Network Interface Cards) such as Ethernet cards do not understand Internet addresses. In the case of Ethernet, every NIC ever manufactured comes equipped with a unique 48-bit Ethernet address. Manufacturers of Ethernet NICs request a block of Ethernet addresses from IEEE to ensure that no two NICs have the same address (to avoid conflicts should the two NICs ever appear on the same LAN). The NICs 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. 5-61, in which a small university with two /24 networks is illustrated. One network (CS) is a switched Ethernet in the Computer Science Dept. It has the prefix 192.32.65.0/24. The other LAN (EE), also switched Ethernet, is in Electrical Engineering and has the prefix 192.32.63.0/24. The two LANs are connected by an IP router. Each machine on an Ethernet and each interface on the router has a unique Ethernet address, labeled *E1* through *E6*, and a unique IP address on the CS or EE network.

Let us start out by seeing how a user on host 1 sends a packet to a user on host 2 on the CS network. Let us assume the sender knows the name of the intended receiver, possibly something like *eagle.cs.uni.edu*. The first step is to find the IP address for host 2. This lookup is performed by DNS, which we will study in Chap. 7. For the moment, we will just assume that DNS returns the IP address for host 2 (192.32.65.5).

The upper layer software on host 1 now builds a packet with 192.32.65.5 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 the CS network, (i.e., its own network). However, it still needs some way to find the destination's Ethernet address to send the frame. One solution is to have a configuration file somewhere in the system that maps IP addresses onto Ethernet addresses. While



Frame	Source IP	Source Eth.	Destination IP	Destination Eth.
Host 1 to 2, on CS net	IP1	E1	IP2	E2
Host 1 to 4, on CS net	IP1	E1	IP4	E3
Host 1 to 4, on EE net	IP1	E4	IP4	E6

Figure 5-61. Two switched Ethernet LANs joined by a router.

this solution is certainly possible, for organizations with thousands of machines keeping all 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 192.32.65.5. The broadcast will arrive at every machine on the CS Ethernet, 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 192.32.65.5 is on the host with Ethernet address *E2*. The protocol used for asking this question and getting the reply is called **ARP** (**Address Resolution Protocol**). Almost every machine on the Internet runs it. ARP 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 192.32.65.5) in the payload field, and dumps it onto the Ethernet. The IP and Ethernet addresses of this packet are given in Fig. 5-61. The Ethernet NIC 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 work more efficiently. 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 the ARP broadcast arrives at host 2, the pair (192.32.65.7, *E1*) is entered into host 2's ARP cache. In fact, all machines on the Ethernet can enter this mapping into their ARP caches.

To allow mappings to change, for example, when a host is configured to use a new IP address (but keeps its old Ethernet address), entries in the ARP cache should time out after a few minutes. A clever way to help keep the cached information current and to optimize performance is to have every machine broadcast its mapping when it is configured. 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 or update an entry in everyone's ARP cache. This is known as a **gratuitous ARP**. If a response does (unexpectedly) arrive, two machines have been assigned the same IP address. The error must be resolved by the network manager before both machines can use the network.

Now let us look at Fig. 5-61 again, only this time assume that host 1 wants to send a packet to host 4 (192.32.63.8) on the EE network. Host 1 will see that the destination IP address is not on the CS network. It knows to send all such off-network traffic to the router, which is also known as the **default gateway**. By convention, the default gateway is the lowest address on the network (198.31.65.1). To send a frame to the router, host 1 must still know the Ethernet address of the router interface on the CS network. It discovers this by sending an ARP broadcast for 198.31.65.1, from which it learns *E3*. It then sends the frame. The same lookup mechanisms are used to send a packet from one router to the next over a sequence of routers in an Internet path.

When the Ethernet NIC of the router gets this frame, it gives the packet to the IP software. It knows from the network masks that the packet should be sent onto the EE network where it will reach host 4. If the router does not know the Ethernet address for host 4, then it will use ARP again. The table in Fig. 5-61 lists the source and destination Ethernet and IP addresses that are present in the frames as observed on the CS and EE networks. Observe that the Ethernet addresses change with the frame on each network while the IP addresses remain constant (because they indicate the endpoints across all of the interconnected networks).

It is also possible to send a packet from host 1 to host 4 without host 1 knowing that host 4 is on a different network. The solution is to have the router answer ARPs on the CS network for host 4 and give its Ethernet address, *E3*, as the response. It is not possible to have host 4 reply directly because it will not see the ARP request (as routers do not forward Ethernet-level broadcasts). The router will then receive frames sent to 192.32.63.8 and forward them onto the EE network. This solution is called **proxy ARP**. It is used in special cases in which a host wants to appear on a network even though it actually resides on another network. A common situation, for example, is a mobile computer that wants some other node to pick up packets for it when it is not on its home network.

DHCP—The Dynamic Host Configuration Protocol

ARP (as well as other Internet protocols) makes the assumption that hosts are configured with some basic information, such as their own IP addresses. How do hosts get this information? It is possible to manually configure each computer, but that is tedious and error-prone. There is a better way, and it is called **DHCP** (**Dynamic Host Configuration Protocol**).

With DHCP, every network must have a DHCP server that is responsible for configuration. When a computer is started, it has a built-in Ethernet or other link layer address embedded in the NIC, but no IP address. Much like ARP, the computer broadcasts a request for an IP address on its network. It does this by using a DHCP DISCOVER packet. This packet must reach the DHCP server. If that server is not directly attached to the network, the router will be configured to receive DHCP broadcasts and relay them to the DHCP server, wherever it is located.

When the server receives the request, it allocates a free IP address and sends it to the host in a DHCP OFFER packet (which again may be relayed via the router). To be able to do this work even when hosts do not have IP addresses, the server identifies a host using its Ethernet address (which is carried in the DHCP DISCOVER packet)

An issue that arises with automatic assignment of IP addresses from a pool is for how long an IP address should be allocated. If a host leaves the network and does not return its IP address to the DHCP server, that address will be permanently lost. After a period of time, many addresses may be lost. To prevent that from happening, IP address assignment may be for a fixed period of time, a technique called **leasing**. Just before the lease expires, the host must ask for a DHCP renewal. If it fails to make a request or the request is denied, the host may no longer use the IP address it was given earlier.

DHCP is described in RFCs 2131 and 2132. It is widely used in the Internet to configure all sorts of parameters in addition to providing hosts with IP addresses. As well as in business and home networks, DHCP is used by ISPs to set the parameters of devices over the Internet access link, so that customers do not need to phone their ISPs to get this information. Common examples of the information that is configured include the network mask, the IP address of the default gateway, and the IP addresses of DNS and time servers. DHCP has largely replaced earlier protocols (called RARP and BOOTP) with more limited functionality.

5.6.5 Label Switching and MPLS

So far, on our tour of the network layer of the Internet, we have focused exclusively on packets as datagrams that are forwarded by IP routers. There is also another kind of technology that is starting to be widely used, especially by ISPs, in order to move Internet traffic across their networks. This technology is

called **MPLS** (**MultiProtocol Label Switching**) and it is perilously close to circuit switching. Despite the fact that many people in the Internet community have an intense dislike for connection-oriented networking, the idea seems to keep coming back. As Yogi Berra once put it, it is like deja vu all over again. However, there are essential differences between the way the Internet handles route construction and the way connection-oriented networks do it, so the technique is certainly not traditional circuit switching.

MPLS adds a label in front of each packet, and forwarding is based on the label rather than on the destination address. Making the label an index into an internal table makes finding the correct output line just a matter of table lookup. Using this technique, forwarding can be done very quickly. This advantage was the original motivation behind MPLS, which began as proprietary technology known by various names including **tag switching**. Eventually, IETF began to standardize the idea. It is described in RFC 3031 and many other RFCs. The main benefits over time have come to be routing that is flexible and forwarding that is suited to quality of service as well as fast.

The first question to ask is where does the label go? Since IP packets were not designed for virtual circuits, there is no field available for virtual-circuit numbers within the IP header. For this reason, a new MPLS header had to be added in front of the IP header. On a router-to-router line using PPP as the framing protocol, the frame format, including the PPP, MPLS, IP, and TCP headers, is as shown in Fig. 5-62.

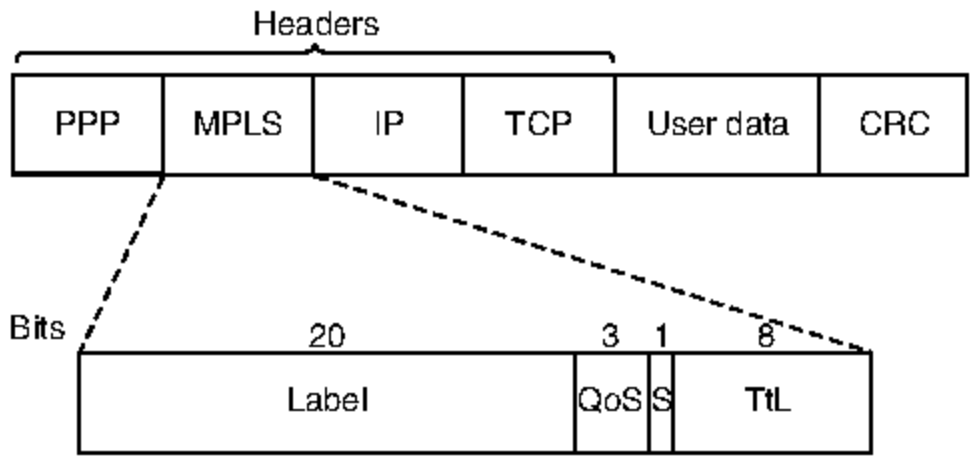


Figure 5-62. Transmitting a TCP segment using IP, MPLS, and PPP.

The generic MPLS header is 4 bytes long and has four fields. Most important is the *Label* field, which holds the index. The *QoS* field indicates the class of service. The *S* field relates to stacking multiple labels (which is discussed below). The *TtL* field indicates how many more times the packet may be forwarded. It is decremented at each router, and if it hits 0, the packet is discarded. This feature prevents infinite looping in the case of routing instability.

MPLS falls between the IP network layer protocol and the PPP link layer protocol. It is not really a layer 3 protocol because it depends on IP or other network

layer addresses to set up label paths. It is not really a layer 2 protocol either because it forwards packets across multiple hops, not a single link. For this reason, MPLS is sometimes described as a layer 2.5 protocol. It is an illustration that real protocols do not always fit neatly into our ideal layered protocol model.

On the brighter side, because the MPLS headers are not part of the network layer packet or the data link layer frame, MPLS is to a large extent independent of both layers. Among other things, this property means it is possible to build MPLS switches that can forward both IP packets and non-IP packets, depending on what shows up. This feature is where the "multiprotocol" in the name MPLS came from. MPLS can also carry IP packets over non-IP networks.

When an MPLS-enhanced packet arrives at a **LSR** (**Label Switched Router**), the label is used as an index into a table to determine the outgoing line to use and also the new label to use. This label swapping is used in all virtual-circuit networks. Labels have only local significance and two different routers can feed unrelated packets with the same label into another router for transmission on the same outgoing line. To be distinguishable at the other end, labels have to be remapped at every hop. We saw this mechanism in action in Fig. 5-3. MPLS uses the same technique.

As an aside, some people distinguish between *forwarding* and *switching*. Forwarding is the process of finding the best match for a destination address in a table to decide where to send packets. An example is the longest matching prefix algorithm used for IP forwarding. In contrast, switching uses a label taken from the packet as an index into a forwarding table. It is simpler and faster. These definitions are far from universal, however.

Since most hosts and routers do not understand MPLS, we should also ask when and how the labels are attached to packets. This happens when an IP packet reaches the edge of an MPLS network. The **LER** (**Label Edge Router**) inspects the destination IP address and other fields to see which MPLS path the packet should follow, and puts the right label on the front of the packet. Within the MPLS network, this label is used to forward the packet. At the other edge of the MPLS network, the label has served its purpose and is removed, revealing the IP packet again for the next network. This process is shown in Fig. 5-63. One difference from traditional virtual circuits is the level of aggregation. It is certainly possible for each flow to have its own set of labels through the MPLS network. However, it is more common for routers to group multiple flows that end at a particular router or LAN and use a single label for them. The flows that are grouped together under a single label are said to belong to the same **FEC** (**Forwarding Equivalence Class**). This class covers not only where the packets are going, but also their service class (in the differentiated services sense) because all the packets are treated the same way for forwarding purposes.

With traditional virtual-circuit routing, it is not possible to group several distinct paths with different endpoints onto the same virtual-circuit identifier because there would be no way to distinguish them at the final destination. With MPLS,

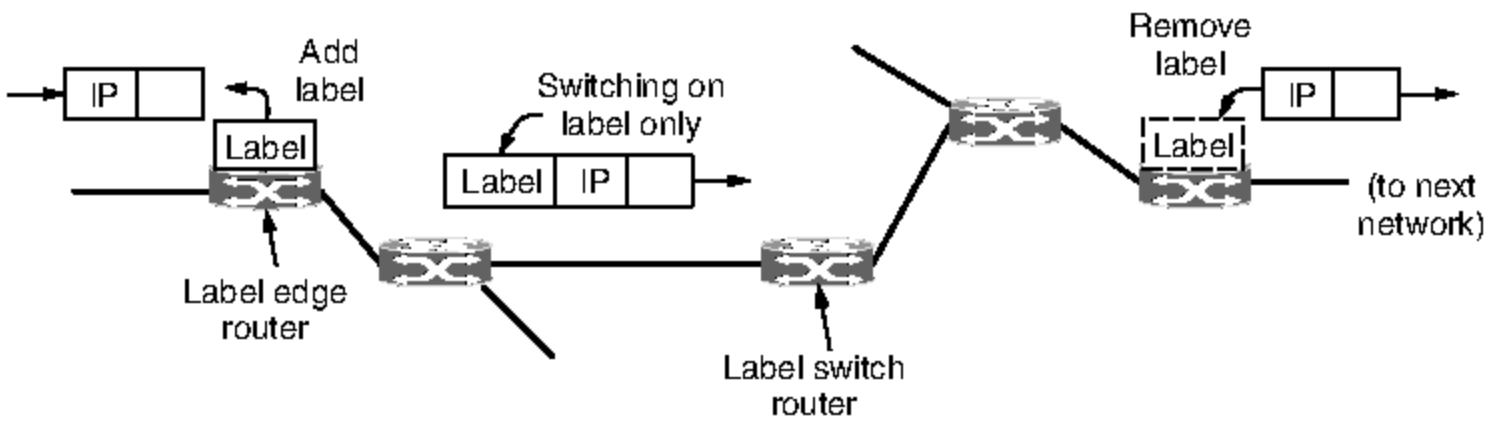


Figure 5-63. Forwarding an IP packet through an MPLS network.

the packets still contain their final destination address, in addition to the label. At the end of the labeled route, the label header can be removed and forwarding can continue the usual way, using the network layer destination address.

Actually, MPLS goes even further. It can operate at multiple levels at once by adding more than one label to the front of a packet. For example, suppose that there are many packets that already have different labels (because we want to treat the packets differently somewhere in the network) that should follow a common path to some destination. Instead of setting up many label switching paths, one for each of the different labels, we can set up a single path. When the already-labeled packets reach the start of this path, another label is added to the front. This is called a stack of labels. The outermost label guides the packets along the path. It is removed at the end of the path, and the labels revealed, if any, are used to forward the packet further. The *S* bit in Fig. 5-62 allows a router removing a label to know if there are any additional labels left. It is set to 1 for the bottom label and 0 for all the other labels.

The final question we will ask is how the label forwarding tables are set up so that packets follow them. This is one area of major difference between MPLS and conventional virtual-circuit designs. In traditional virtual-circuit networks, when a user wants to establish a connection, a setup packet is launched into the network to create the path and make the forwarding table entries. MPLS does not involve users in the setup phase. Requiring users to do anything other than send a datagram would break too much existing Internet software.

Instead, the forwarding information is set up by protocols that are a combination of routing protocols and connection setup protocols. These control protocols are cleanly separated from label forwarding, which allows multiple, different control protocols to be used. One of the variants works like this. When a router is booted, it checks to see which routes it is the final destination for (e.g., which prefixes belong to its interfaces). It then creates one or more FECs for them, allocates a label for each one, and passes the labels to its neighbors. They, in turn, enter the labels in their forwarding tables and send new labels to their neighbors, until all the routers have acquired the path. Resources can also be reserved as the

path is constructed to guarantee an appropriate quality of service. Other variants can set up different paths, such as traffic engineering paths that take unused capacity into account, and create paths on-demand to support service offerings such as quality of service.

Although the basic ideas behind MPLS are straightforward, the details are complicated, with many variations and use cases that are being actively developed. For more information, see Davie and Farrel (2008) and Davie and Rekhter (2000).

5.6.6 OSPF—An Interior Gateway Routing Protocol

We have now finished our study of how packets are forwarded in the Internet. It is time to move on to the next topic: routing in the Internet. As we mentioned earlier, the Internet is made up of a large number of independent networks or **ASes** (**Autonomous Systems**) that are operated by different organizations, usually a company, university, or ISP. Inside of its own network, an organization can use its own algorithm for internal routing, or **intradomain routing**, as it is more commonly known. Nevertheless, there are only a handful of standard protocols that are popular. In this section, we will study the problem of intradomain routing and look at the OSPF protocol that is widely used in practice. An intradomain routing protocol is also called an **interior gateway protocol**. In the next section, we will study the problem of routing between independently operated networks, or **interdomain routing**. For that case, all networks must use the same interdomain routing protocol or **exterior gateway protocol**. The protocol that is used in the Internet is BGP (Border Gateway Protocol).

Early intradomain routing protocols used a distance vector design, based on the distributed Bellman-Ford algorithm inherited from the ARPANET. RIP (Routing Information Protocol) is the main example that is used to this day. It works well in small systems, but less well as networks get larger. It also suffers from the count-to-infinity problem and generally slow convergence. The ARPANET switched over to a link state protocol in May 1979 because of these problems, and in 1988 IETF began work on a link state protocol for intradomain routing. That protocol, called **OSPF** (**Open Shortest Path First**), became a standard in 1990. It drew on a protocol called **IS-IS** (**Intermediate-System to Intermediate-System**), which became an ISO standard. Because of their shared heritage, the two protocols are much more alike than different. For the complete story, see RFC 2328. They are the dominant intradomain routing protocols, and most router vendors now support both of them. OSPF is more widely used in company networks, and IS-IS is more widely used in ISP networks. Of the two, we will give a sketch of how OSPF works.

Given the long experience with other routing protocols, the group designing OSPF 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. At the time, IP had a *Type of service* field, but no existing routing protocol used it. This field was included in OSPF but still nobody used it, and it was eventually removed. Perhaps this requirement was ahead of its time, as it preceded IETF's work on differentiated services, which has rejuvenated classes of service.

Fifth, and related to the above, OSPF had to do load balancing, splitting the load over multiple lines. Most previous protocols sent all packets over a single best route, even if there were two routes that were equally good. The other route was not used at all. In many cases, splitting the load over multiple routes gives better performance.

Sixth, support for hierarchical systems was needed. By 1988, some networks had grown so large that no router could be expected to know the entire topology. OSPF 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 both point-to-point links (e.g., SONET) and broadcast networks (e.g., most LANs). Actually, it is able to support networks with multiple routers, each of which can communicate directly with the others (called **multiaccess networks**) even if they do not have broadcast capability. Earlier protocols did not handle this case well.

An example of an autonomous system network is given in Fig. 5-64(a). Hosts are omitted because they do not generally play a role in OSPF, while routers and networks (which may contain hosts) do. Most of the routers in Fig. 5-64(a) are connected to other routers by point-to-point links, and to networks to reach the hosts on those networks. However, routers *R3*, *R4*, and *R5* are connected by a broadcast LAN such as switched Ethernet.

OSPF operates by abstracting the collection of actual networks, routers, and links into a directed graph in which each arc is assigned a weight (distance, delay, etc.). A point-to-point connection between two routers is represented by a pair of arcs, one in each direction. Their weights may be different. A broadcast network is represented by a node for the network itself, plus a node for each router. The arcs from that network node to the routers have weight 0. They are important nonetheless, as without them there is no path through the network. Other networks, which have only hosts, have only an arc reaching them and not one returning. This structure gives routes to hosts, but not through them.

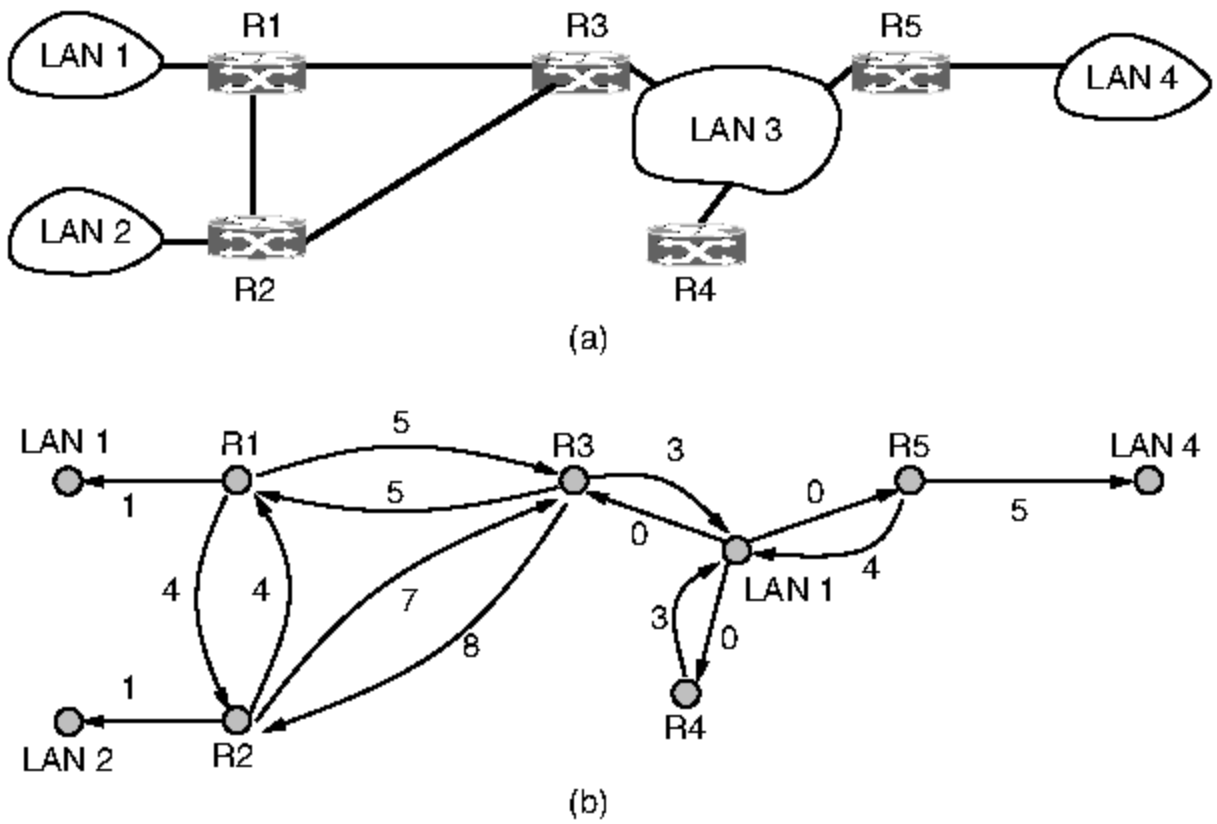


Figure 5-64. (a) An autonomous system. (b) A graph representation of (a).

Figure 5-64(b) shows the graph representation of the network of Fig. 5-64(a). What OSPF fundamentally does is represent the actual network as a graph like this and then use the link state method to have every router compute the shortest path from itself to all other nodes. Multiple paths may be found that are equally short. In this case, OSPF remembers the set of shortest paths and during packet forwarding, traffic is split across them. This helps to balance load. It is called **ECMP (Equal Cost MultiPath)**.

Many of the ASes in the Internet are themselves large and nontrivial to manage. To work at this scale, OSPF allows an AS to be divided 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. Routers that lie wholly within an area are called **internal routers**. An area is a generalization of an individual network. Outside an area, its destinations are visible but not its topology. This characteristic helps routing to scale.

Every AS has a **backbone area**, called area 0. The routers in this area are called **backbone routers**. 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 just another arc with a cost. As with other areas, the topology of the backbone is not visible outside the backbone.

Each router that is connected to two or more areas is called an **area border router**. It must also be part of the backbone. The job of an area border router is to summarize the destinations in one area and to inject this summary into the other

areas to which it is connected. This summary includes cost information but not all the details of the topology within an area. Passing cost information allows hosts in other areas to find the best area border router to use to enter an area. Not passing topology information reduces traffic and simplifies the shortest-path computations of routers in other areas. However, if there is only one border router out of an area, even the summary does not need to be passed. Routes to destinations out of the area always start with the instruction "Go to the border router." This kind of area is called a **stub area**.

The last kind of router is the **AS boundary router**. It injects routes to external destinations on other ASes into the area. The external routes then appear as destinations that can be reached via the AS boundary router with some cost. An external route can be injected at one or more AS boundary routers. The relationship between ASes, areas, and the various kinds of routers is shown in Fig. 5-65. One router may play multiple roles, for example, a border router is also a backbone router.

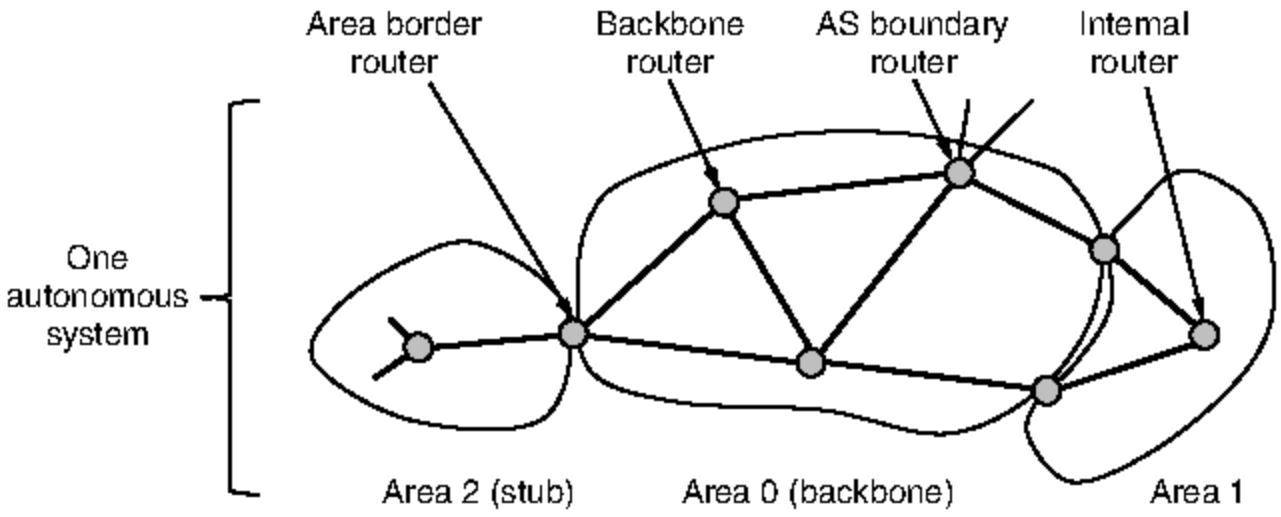


Figure 5-65. The relation between ASes, backbones, and areas in OSPF.

During normal operation, each router within an area 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 every other router and network in the entire AS. An area border router needs the databases for all the areas to which it is connected and must run the shortest path algorithm for each area separately.

For a source and destination in the same area, the best intra-area route (that lies wholly within the area) is chosen. For a source and destination in different areas, the inter-area route must go from the source to the backbone, across the backbone to the destination area, and then to the destination. This algorithm forces a star configuration on OSPF, with the backbone being the hub and the other areas being spokes. Because the route with the lowest cost is chosen, routers in different parts of the network may use different area border routers to enter the backbone and destination area. 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). Also, routes to external destinations may include the external cost from the AS boundary router over the external path, if desired, or just the cost internal to the AS.

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. From the responses, each router learns who its neighbors are. Routers on the same LAN are all neighbors.

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 on its LAN, and exchanges information with them. In effect, it is acting as the single node that represents the LAN. 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 and need to be replaced immediately.

During normal operation, each router periodically floods LINK STATE UPDATE messages to each of its adjacent routers. These messages gives its state and provide 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 link 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 link is brought up.

Either partner can request link state information from the other one by using LINK STATE REQUEST messages. The 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 directly in IP packets. The five kinds of messages are summarized in Fig. 5-66.

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

Figure 5-66. 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 links to other routers and networks and the cost of these links. This information allows each router to construct the graph for its area(s) and compute the shortest paths. The backbone area does this work, 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, internal routers can select the best route to a destination outside their area, including the best exit router to the backbone.

5.6.7 BGP—The Exterior Gateway Routing Protocol

Within a single AS, OSPF and IS-IS are the protocols that are commonly used. Between ASes, a different protocol, called **BGP** (**Border Gateway Protocol**), is used. A different protocol is needed because the goals of an intradomain protocol and an interdomain protocol are not the same. All an intradomain 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.

In contrast, interdomain routing protocols have to worry about politics a great deal (Metz, 2001). 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 is 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 carriers 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 possible routing constraints are:

1. Do not carry commercial traffic on the educational network.
2. Never send traffic from the Pentagon on a route through Iraq.
3. Use TeliaSonera instead of Verizon because it is cheaper.
4. Don't use AT&T in Australia because performance is poor.
5. Traffic starting or ending at Apple should not transit Google.

As you might imagine from this list, routing policies can be highly individual. They are often proprietary because they contain sensitive business information.

However, we can describe some patterns that capture the reasoning of the company above and that are often used as a starting point.

A routing policy is implemented by deciding what traffic can flow over which of the links between ASes. One common policy is that a customer ISP pays another provider ISP to deliver packets to any other destination on the Internet and receive packets sent from any other destination. The customer ISP is said to buy **transit service** from the provider ISP. This is just like a customer at home buying Internet access service from an ISP. To make it work, the provider should advertise routes to all destinations on the Internet to the customer over the link that connects them. In this way, the customer will have a route to use to send packets anywhere. Conversely, the customer should advertise routes only to the destinations on its network to the provider. This will let the provider send traffic to the customer only for those addresses; the customer does not want to handle traffic intended for other destinations.

We can see an example of transit service in Fig. 5-67. There are four ASes that are connected. The connection is often made with a link at **IXPs** (**Internet eXchange Points**), facilities to which many ISPs have a link for the purpose of connecting with other ISPs. *AS2*, *AS3*, and *AS4* are customers of *AS1*. They buy transit service from it. Thus, when source *A* sends to destination *C*, the packets travel from *AS2* to *AS1* and finally to *AS4*. The routing advertisements travel in the opposite direction to the packets. *AS4* advertises *C* as a destination to its transit provider, *AS1*, to let sources reach *C* via *AS1*. Later, *AS1* advertises a route to *C* to its other customers, including *AS2*, to let the customers know that they can send traffic to *C* via *AS1*.

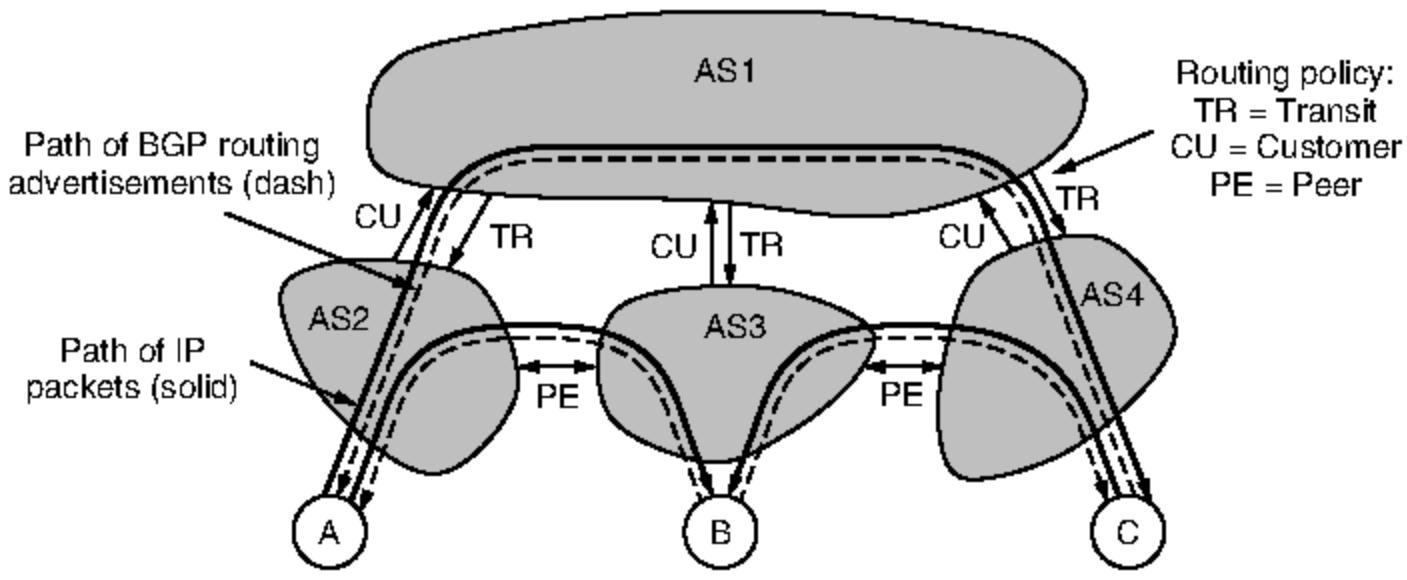


Figure 5-67. Routing policies between four autonomous systems.

In Fig. 5-67, all of the other ASes buy transit service from *AS1*. This provides them with connectivity so they can interact with any host on the Internet. However, they have to pay for this privilege. Suppose that *AS2* and *AS3* exchange a lot of traffic. Given that their networks are connected already, if they want to, they

can use a different policy—they can send traffic directly to each other for free. This will reduce the amount of traffic they must have *AS1* deliver on their behalf, and hopefully it will reduce their bills. This policy is called **peering**.

To implement peering, two ASes send routing advertisements to each other for the addresses that reside in their networks. Doing so makes it possible for *AS2* to send *AS3* packets from *A* destined to *B* and vice versa. However, note that peering is not transitive. In Fig. 5-67, *AS3* and *AS4* also peer with each other. This peering allows traffic from *C* destined for *B* to be sent directly to *AS4*. What happens if *C* sends a packet to *A*? *AS3* is only advertising a route to *B* to *AS4*. It is not advertising a route to *A*. The consequence is that traffic will not pass from *AS4* to *AS3* to *AS2*, even though a physical path exists. This restriction is exactly what *AS3* wants. It peers with *AS4* to exchange traffic, but does not want to carry traffic from *AS4* to other parts of the Internet since it is not being paid to so do. Instead, *AS4* gets transit service from *AS1*. Thus, it is *AS1* who will carry the packet from *C* to *A*.

Now that we know about transit and peering, we can also see that *A*, *B*, and *C* have transit arrangements. For example, *A* must buy Internet access from *AS2*. *A* might be a single home computer or a company network with many LANs. However, it does not need to run BGP because it is a **stub network** that is connected to the rest of the Internet by only one link. So the only place for it to send packets destined outside of the network is over the link to *AS2*. There is nowhere else to go. This path can be arranged simply by setting up a default route. For this reason, we have not shown *A*, *B*, and *C* as ASes that participate in interdomain routing.

On the other hand, some company networks are connected to multiple ISPs. This technique is used to improve reliability, since if the path through one ISP fails, the company can use the path via the other ISP. This technique is called **multihoming**. In this case, the company network is likely to run an interdomain routing protocol (e.g., BGP) to tell other ASes which addresses should be reached via which ISP links.

Many variations on these transit and peering policies are possible, but they already illustrate how business relationships and control over where route advertisements go can implement different kinds of policies. Now we will consider in more detail how routers running BGP advertise routes to each other and select paths over which to forward packets.

BGP is a form of distance vector protocol, but it is quite unlike intradomain distance vector protocols such as RIP. We have already seen that policy, instead of minimum distance, is used to pick which routes to use. Another large difference is that instead of maintaining just the cost of the route to each destination, each BGP router keeps track of the path used. This approach is called a **path vector protocol**. The path consists of the next hop router (which may be on the other side of the ISP, not adjacent) and the sequence of ASes, or **AS path**, that the route has followed (given in reverse order). Finally, pairs of BGP routers communicate

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

An example of how BGP routes are advertised is shown in Fig. 5-68. There are three ASes and the middle one is providing transit to the left and right ISPs. A route advertisement to prefix *C* starts in *AS3*. When it is propagated across the link to *R2c* at the top of the figure, it has the AS path of simply *AS3* and the next hop router of *R3a*. At the bottom, it has the same AS path but a different next hop because it came across a different link. This advertisement continues to propagate and crosses the boundary into *AS1*. At router *R1a*, at the top of the figure, the AS path is *AS2*, *AS3* and the next hop is *R2a*.

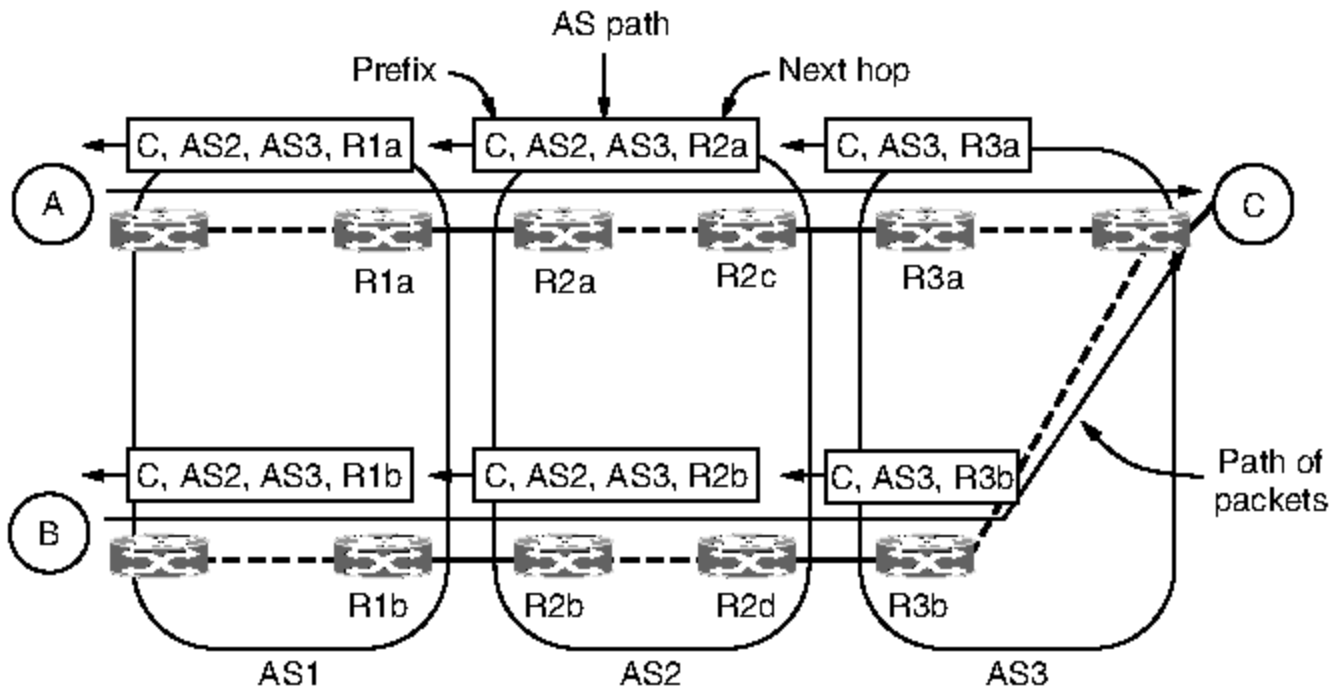


Figure 5-68. Propagation of BGP route advertisements.

Carrying the complete path with the route makes it easy for the receiving router to detect and break routing loops. The rule is that each router that sends a route outside of the AS prepends its own AS number to the route. (This is why the list is in reverse order.) When a router receives a route, it checks to see if its own AS number is already in the AS path. If it is, a loop has been detected and the advertisement is discarded. However, and somewhat ironically, it was realized in the late 1990s that despite this precaution BGP suffers from a version of the count-to-infinity problem (Labovitz et al., 2001). There are no long-lived loops, but routes can sometimes be slow to converge and have transient loops.

Giving a list of ASes is a very coarse way to specify a path. An AS might be a small company, or an international backbone network. There is no way of telling from the route. BGP does not even try because different ASes may use different intradomain protocols whose costs cannot be compared. Even if they could be compared, an AS may not want to reveal its internal metrics. This is one of the ways that interdomain routing protocols differ from intradomain protocols.

So far we have seen how a route advertisement is sent across the link between two ISPs. We still need some way to propagate BGP routes from one side of the ISP to the other, so they can be sent on to the next ISP. This task could be handled by the intradomain protocol, but because BGP is very good at scaling to large networks, a variant of BGP is often used. It is called **iBGP** (**internal BGP**) to distinguish it from the regular use of BGP as **eBGP** (**external BGP**).

The rule for propagating routes inside an ISP is that every router at the boundary of the ISP learns of all the routes seen by all the other boundary routers, for consistency. If one boundary router on the ISP learns of a prefix to IP 128.208.0.0/16, all the other routers will learn of this prefix. The prefix will then be reachable from all parts of the ISP, no matter how packets enter the ISP from other ASes.

We have not shown this propagation in Fig. 5-68 to avoid clutter, but, for example, router *R2b* will know that it can reach *C* via either router *R2c* at top or router *R2d* at bottom. The next hop is updated as the route crosses within the ISP so that routers on the far side of the ISP know which router to use to exit the ISP on the other side. This can be seen in the leftmost routes in which the next hop points to a router in the same ISP and not a router in the next ISP.

We can now describe the key missing piece, which is how BGP routers choose which route to use for each destination. Each BGP router may learn a route for a given destination from the router it is connected to in the next ISP and from all of the other boundary routers (which have heard different routes from the routers they are connected to in other ISPs). Each router must decide which route in this set of routes is the best one to use. Ultimately the answer is that it is up to the ISP to write some policy to pick the preferred route. However, this explanation is very general and not at all satisfying, so we can at least describe some common strategies.

The first strategy is that routes via peered networks are chosen in preference to routes via transit providers. The former are free; the latter cost money. A similar strategy is that customer routes are given the highest preference. It is only good business to send traffic directly to the paying customers.

A different kind of strategy is the default rule that shorter AS paths are better. This is debatable given that an AS could be a network of any size, so a path through three small ASes could actually be shorter than a path through one big AS. However, shorter tends to be better on average, and this rule is a common tiebreaker.

The final strategy is to prefer the route that has the lowest cost within the ISP. This is the strategy implemented in Fig. 5-68. Packets sent from *A* to *C* exit *AS1* at the top router, *R1a*. Packets sent from *B* exit via the bottom router, *R1b*. The reason is that both *A* and *B* are taking the lowest-cost path or quickest route out of *AS1*. Because they are located in different parts of the ISP, the quickest exit for each one is different. The same thing happens as the packets pass through *AS2*. On the last leg, *AS3* has to carry the packet from *B* through its own network.

This strategy is known as **early exit** or **hot-potato routing**. It has the curious side effect of tending to make routes asymmetric. For example, consider the path taken when *C* sends a packet back to *B*. The packet will exit *AS3* quickly, at the top router, to avoid wasting its resources. Similarly, it will stay at the top when *AS2* passes it to *AS1* as quickly as possible. Then the packet will have a longer journey in *AS1*. This is a mirror image of the path taken from *B* to *C*.

The above discussion should make clear that each BGP router chooses its own best route from the known possibilities. It is not the case, as might naively be expected, that BGP chooses a path to follow at the AS level and OSPF chooses paths within each of the ASes. BGP and the interior gateway protocol are integrated much more deeply. This means that, for example, BGP can find the best exit point from one ISP to the next and this point will vary across the ISP, as in the case of the hot-potato policy. It also means that BGP routers in different parts of one AS may choose different AS paths to reach the same destination. Care must be exercised by the ISP to configure all of the BGP routers to make compatible choices given all of this freedom, but this can be done in practice.

Amazingly, we have only scratched the surface of BGP. For more information, see the BGP version 4 specification in RFC 4271 and related RFCs. However, realize that much of its complexity lies with policies, which are not described in the specification of the BGP protocol.

5.6.8 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 streaming a live sports event to many viewers, delivering program updates to a pool of replicated servers, and handling digital conference (i.e., multiparty) telephone calls.

IP supports one-to-many communication, or multicasting, using class D IP 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-effort 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.

The range of IP addresses 224.0.0.0/24 is reserved for multicast on the local network. In this case, no routing protocol is needed. The packets are multicast by simply broadcasting them on the LAN with a multicast address. All hosts on the LAN receive the broadcasts, and hosts that are members of the group process the packet. Routers do not forward the packet off the LAN. Some examples of local multicast 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
224.0.0.251 All DNS servers on a LAN

Other multicast addresses may have members on different networks. In this case, a routing protocol is needed. But first the multicast routers need to know which hosts are members of a group. A process asks its host to join in a specific group. It can also ask its host to leave the group. Each host keeps track of which groups its processes currently belong to. When the last process on a host leaves a group, the host is no longer a member of that group. About once a minute, each multicast router sends a query packet to all the hosts on its LAN (using the local multicast address of 224.0.0.1, of course) asking them to report back on the groups to which they currently belong. The multicast routers may or may not be colocated with the standard routers. 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**). It is described in RFC 3376.

Any of several multicast routing protocols may be used to build multicast spanning trees that give paths from senders to all of the members of the group. The algorithms that are used are the ones we described in Sec. 5.2.8. Within an AS, the main protocol used is **PIM** (**Protocol Independent Multicast**). PIM comes in several flavors. In Dense Mode PIM, a pruned reverse path forwarding tree is created. This is suited to situations in which members are everywhere in the network, such as distributing files to many servers within a data center network. In Sparse Mode PIM, spanning trees that are built are similar to core-based trees. This is suited to situations such as a content provider multicasting TV to subscribers on its IP network. A variant of this design, called Source-Specific Multicast PIM, is optimized for the case that there is only one sender to the group. Finally, multicast extensions to BGP or tunnels need to be used to create multicast routes when the group members are in more than one AS.

5.6.9 Mobile IP

Many users of the Internet have mobile computers and want to stay connected when they are away from home and even on the road in between. Unfortunately, the IP addressing system makes working far from home easier said than done, as we will describe shortly. When people began demanding the ability anyway, 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. 5.2.10. In brief, every site that wants to allow its users to roam has to create a helper at the site called a **home agent**. When a mobile host shows up at a foreign site, it obtains a new IP address (called a care-of address) at the foreign site. The mobile then tells the home agent where it is now by giving it the care-of address. When a packet for the mobile arrives at the home site and the mobile is elsewhere, the home agent grabs the packet and tunnels it to the mobile at the current care-of address. The mobile can send reply packets directly to whoever it is communicating with, but still using its home address as the source address. This solution meets all the requirements stated above except that packets for mobile hosts do make detours.

Now that we have covered the network layer of the Internet, we can go into the solution in more detail. The need for mobility support in the first place comes from the IP addressing scheme itself. Every IP address contains a network number and a host number. For example, consider the machine with IP address 160.80.40.20/16. The 160.80 gives the network number; the 40.20 is the host number. Routers all over the world have routing tables telling which link 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).

At this stage, there are two options—both unattractive. The first is that we could create a route to a more specific prefix. That is, if the distant site advertises a route to 160.80.40.20/32, packets sent to the destination will start arriving in the right place again. This option depends on the longest matching prefix algorithm that is used at routers. However, we have added a route to an IP prefix with a single IP address in it. All ISPs in the world will learn about this prefix. If everyone changes global IP routes in this way when they move their computer, each router would have millions of table entries, at astronomical cost to the Internet. This option is not workable.

The second option is to change the IP address of the mobile. True, packets sent to the home IP address will no longer be delivered until all the relevant people, programs, and databases are informed of the change. But the mobile can still use the Internet at the new location to browse the Web and run other applications. This option handles mobility at a higher layer. It is what typically happens when a user takes a laptop to a coffee store and uses the Internet via the local wireless network. The disadvantage is that it breaks some applications, and it does not keep connectivity as the mobile moves around.

As an aside, mobility can also be handled at a lower layer, the link layer. This is what happens when using a laptop on a single 802.11 wireless network. The IP address of the mobile does not change and the network path remains the same. It is the wireless link that is providing mobility. However, the degree of mobility is limited. If the laptop moves too far, it will have to connect to the Internet via another network with a different IP address.

The mobile IP solution for IPv4 is given in RFC 3344. It works with the existing Internet routing and allows hosts to stay connected with their own IP addresses as they move about. For it to work, the mobile must be able to discover when it has moved. This is accomplished with ICMP router advertisement and solicitation messages. Mobiles listen for periodic router advertisements or send a solicitation to discover the nearest router. If this router is not the usual address of the router when the mobile is at home, it must be on a foreign network. If this router has changed since last time, the mobile has moved to another foreign network. This same mechanism lets mobile hosts find their home agents.

To get a care-of IP address on the foreign network, a mobile can simply use DHCP. Alternatively, if IPv4 addresses are in short supply, the mobile can send and receive packets via a foreign agent that already has an IP address on the network. The mobile host finds a foreign agent using the same ICMP mechanism used to find the home agent. After the mobile obtains an IP address or finds a foreign agent, it is able to use the network to send a message to its home agent, informing the home agent of its current location.

The home agent needs a way to intercept packets sent to the mobile only when the mobile is not at home. ARP provides a convenient mechanism. To send a packet over an Ethernet to an IP host, the router needs to know the Ethernet address of the host. The usual mechanism is for the router to send an ARP query to ask, for example, what is the Ethernet address of 160.80.40.20. When the mobile is at home, it answers ARP queries for its IP address with its own Ethernet address. When the mobile is away, the home agent responds to this query by giving its Ethernet address. The router then sends packets for 160.80.40.20 to the home agent. Recall that this is called a proxy ARP.

To quickly update ARP mappings back and forth when the mobile leaves home or arrives back home, another ARP technique called a **gratuitous ARP** can be used. Basically, the mobile or home agent send themselves an ARP query for the mobile IP address that supplies the right answer so that the router notices and updates its mapping.

Tunneling to send a packet between the home agent and the mobile host at the care-of address is done by encapsulating the packet with another IP header destined for the care-of address. When the encapsulated packet arrives at the care-of address, the outer IP header is removed to reveal the packet.

As with many Internet protocols, the devil is in the details, and most often the details of compatibility with other protocols that are deployed. There are two complications. First, NAT boxes depend on peeking past the IP header to look at the TCP or UDP header. The original form of tunneling for mobile IP did not use these headers, so it did not work with NAT boxes. The solution was to change the encapsulation to include a UDP header.

The second complication is that some ISPs check the source IP addresses of packets to see that they match where the routing protocol believes the source should be located. This technique is called **ingress filtering**, and it is a security

measure intended to discard traffic with seemingly incorrect addresses that may be malicious. However, packets sent from the mobile to other Internet hosts when it is on a foreign network will have a source IP address that is out of place, so they will be discarded. To get around this problem, the mobile can use the care-of address as a source to tunnel the packets back to the home agent. From here, they are sent into the Internet from what appears to be the right location. The cost is that the route is more roundabout.

Another issue we have not discussed is security. When a home agent gets a message asking it to please forward all of Roberta's packets to some IP address, it had better not comply unless it is convinced that Roberta 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. 8.

Mobility protocols for IPv6 build on the IPv4 foundation. The scheme above suffers from the triangle routing problem in which packets sent to the mobile take a dogleg through a distant home agent. In IPv6, route optimization is used to follow a direct path between the mobile and other IP addresses after the initial packets have followed the long route. Mobile IPv6 is defined in RFC 3775.

There is another kind of mobility that is also being defined for the Internet. Some airplanes have built-in wireless networking that passengers can use to connect their laptops to the Internet. The plane has a router that connects to the rest of the Internet via a wireless link. (Did you expect a wired link?) So now we have a flying router, which means that the whole network is mobile. Network mobility designs support this situation without the laptops realizing that the plane is mobile. As far as they are concerned, it is just another network. Of course, some of the laptops may be using mobile IP to keep their home addresses while they are on the plane, so we have two levels of mobility. Network mobility is defined for IPv6 in RFC 3963.

5.7 SUMMARY

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

Many routing algorithms are used in computer networks. Flooding is a simple algorithm to send a packet along all paths. Most algorithms find the shortest path and adapt to changes in the network topology. The main algorithms are distance vector routing and link state routing. Most actual networks use one of these. Other important routing topics are the use of hierarchy in large networks, routing for mobile hosts, and broadcast, multicast, and anycast routing.

Networks can easily become congested, leading to increased delay and lost packets. Network designers attempt to avoid congestion by designing the network to have enough capacity, choosing uncongested routes, refusing to accept more traffic, signaling sources to slow down, and shedding load.

The next step beyond just dealing with congestion is to actually try to achieve a promised quality of service. Some applications care more about throughput whereas others care more about delay and jitter. The methods that can be used to provide different qualities of service include a combination of traffic shaping, reserving resources at routers, and admission control. Approaches that have been designed for good quality of service include IETF integrated services (including RSVP) and differentiated services.

Networks differ in various ways, so when multiple networks are interconnected, problems can occur. When different networks have different maximum packet sizes, fragmentation may be needed. Different networks may run different routing protocols internally but need to run a common protocol externally. 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.

The Internet has a rich variety of protocols related to the network layer. These include the datagram protocol, IP, and associated control protocols such as ICMP, ARP, and DHCP. A connection-oriented protocol called MPLS carries IP packets across some networks. One of the main routing protocols used within networks is OSPF, and the routing protocol used across networks is BGP. The Internet is rapidly running out of IP addresses, so a new version of IP, IPv6, has been developed and is ever-so-slowly being deployed.

PROBLEMS

1. Give two example computer applications for which connection-oriented service is appropriate. Now give two examples for which connectionless service is best.
2. Datagram networks route each packet as a separate unit, independent of all others. Virtual-circuit networks do not have to do this, since each data packet follows a predetermined route. Does this observation mean that virtual-circuit networks do not need the capability to route isolated packets from an arbitrary source to an arbitrary destination? Explain your answer.
3. Give three examples of protocol parameters that might be negotiated when a connection is set up.
4. 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?

5. 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.

6. Consider the network of Fig. 5-12(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 cost of the links from *C* 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 cost.

7. If costs are recorded as 8-bit numbers in a 50-router network, and distance 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.

8. In Fig. 5-13 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 networks under all circumstances?

9. 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? A good starting place is the hypothesis that a solution with k clusters of k regions of k routers is close to optimal, which means that k is about the cube root of 4800 (around 16). Use trial and error to check out combinations where all three parameters are in the general vicinity of 16.

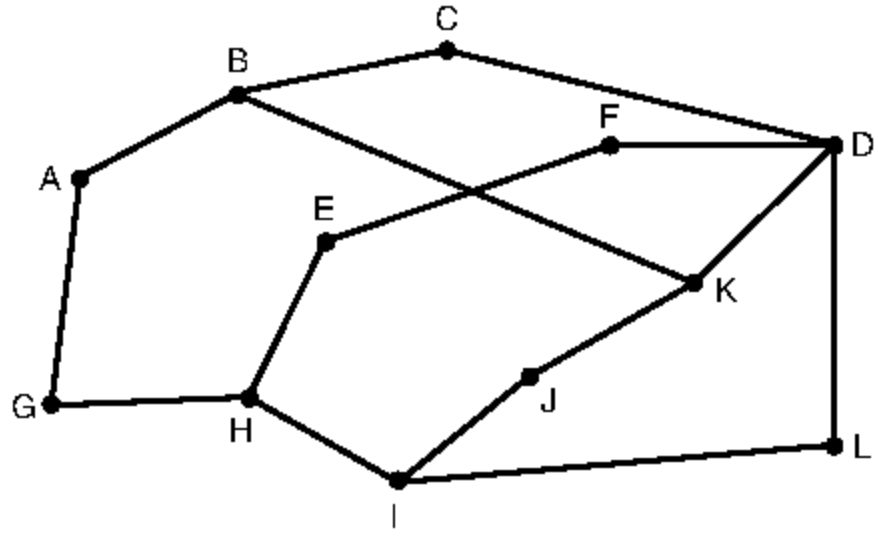
10. In the text it was stated that when a mobile host is not at home, packets sent to its home LAN are intercepted by its home agent on that LAN. For an IP network on an 802.3 LAN, how does the home agent accomplish this interception?

11. Looking at the network of Fig. 5-6, how many packets are generated by a broadcast from *B*, using

(a) reverse path forwarding?
(b) the sink tree?

12. Consider the network of Fig. 5-15(a). Imagine that one new line is added, between *F* and *G*, but the sink tree of Fig. 5-15(b) remains unchanged. What changes occur to Fig. 5-15(c)?

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



14. Suppose that node *B* in Fig. 5-20 has just rebooted and has no routing information in its tables. It suddenly needs a route to *H*. It sends out broadcasts with *TtL* set to 1, 2, 3, and so on. How many rounds does it take to find a route?

15. As a possible congestion control mechanism in a network 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 network 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. Describe two major differences between the ECN method and the RED method of congestion avoidance.

18. A token bucket scheme is used for traffic shaping. A new token is put into the bucket every 5 μsec. Each token is good for one short packet, which contains 48 bytes of data. What is the maximum sustainable data rate?

19. 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?

20. The network of Fig. 5-34 uses RSVP with multicast trees for hosts 1 and 2 as shown. Suppose that host 3 requests a channel of bandwidth 2 MB/sec for a flow from host 1 and another channel of bandwidth 1 MB/sec for a flow from host 2. At the same time, host 4 requests a channel of bandwidth 2 MB/sec for a flow from host 1 and host 5 requests a channel of bandwidth 1 MB/sec for a flow from host 2. How much total bandwidth will be reserved for these requests at routers *A*, *B*, *C*, *E*, *H*, *J*, *K*, and *L*?

21. A router can process 2 million packets/sec. The load offered to it is 1.5 million packets/sec on average. If a route from source to destination contains 10 routers, how much time is spent being queued and serviced by the router?

22. Consider the user of differentiated services with expedited forwarding. Is there a guarantee that expedited packets experience a shorter delay than regular packets? Why or why not?

23. Suppose that host *A* is connected to a router *R*1, *R*1 is connected to another router, *R*2, and *R*2 is connected to host *B*. Suppose that a TCP message that contains 900 bytes of data and 20 bytes of TCP header is passed to the IP code at host *A* for delivery to *B*. Show the *Total length*, *Identification*, *DF*, *MF*, and *Fragment offset* fields of the IP header in each packet transmitted over the three links. Assume that link *A-R1* can support a maximum frame size of 1024 bytes including a 14-byte frame header, link *R1-R2* can support a maximum frame size of 512 bytes, including an 8-byte frame header, and link *R2-B* can support a maximum frame size of 512 bytes including a 12-byte frame header.

24. A router is blasting out IP packets whose total length (data plus header) is 1024 bytes. Assuming that packets live for 10 sec, what is the maximum line speed the router can operate at without danger of cycling through the IP datagram ID number space?

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 originally, 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 network on the Internet has a subnet mask of 255.255.240.0. What is the maximum number of hosts it can handle?

29. While IP addresses are tried to specific networks, Ethernet addresses are not. Can you think of a good reason why they are not?

30. A large number of consecutive IP addresses are available starting at 198.16.0.0. Suppose that four organizations, *A*, *B*, *C*, and *D*, request 4000, 2000, 4000, and 8000 addresses, respectively, and in that order. For each of these, give the first IP address assigned, the last IP address assigned, and the mask in the *w.x.y.z/s* notation.

31. A router has just received the following new IP addresses: 57.6.96.0/21, 57.6.104.0/21, 57.6.112.0/21, and 57.6.120.0/21. If all of them use the same outgoing line, can they be aggregated? If so, to what? If not, why not?

32. The set of IP addresses from 29.18.0.0 to 19.18.128.255 has been aggregated to 29.18.0.0/17. However, there is a gap of 1024 unassigned addresses from 29.18.60.0 to 29.18.63.255 that are now suddenly assigned to a host using a different outgoing line. Is it now necessary to split up the aggregate address into its constituent blocks, add the new block to the table, and then see if any reaggregation is possible? If not, what can be done instead?

33. A router has the following (CIDR) entries in its routing table:

Address/mask	**Next hop**
135.46.56.0/22	Interface 0
135.46.60.0/22	Interface 1
192.53.40.0/23	Router 1
default	Router 2

For each of the following IP addresses, what does the router do if a packet with that address arrives?

(a) 135.46.63.10
(b) 135.46.57.14
(c) 135.46.52.2
(d) 192.53.40.7
(e) 192.53.56.7

34. Many companies have a policy of having two (or more) routers connecting the company to the Internet to provide some redundancy in case one of them goes down. Is this policy still possible with NAT? Explain your answer.

35. 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?

36. Describe a way to reassemble IP fragments at the destination.

37. Most IP datagram reassembly algorithms have a timer to avoid having a lost fragment tie up reassembly buffers forever. Suppose that a datagram is fragmented into four fragments. The first three fragments arrive, 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?

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

39. 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?

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

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

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

43. 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 both delay and the bandwidth used.

44. 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 time-outs 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 time-out interval, parameterized by error rate.

45. Write a function to do forwarding in an IP router. The procedure has one parameter, an IP address. It also has access to a global table consisting of an array of triples. Each triple contains three integers: an IP address, a subnet mask, and the outline line to use. The function looks up the IP address in the table using CIDR and returns the line to use as its value.

46. Use the *traceroute* (UNIX) or *tracert* (Windows) programs to trace the route from your computer to various universities on other continents. Make a list of transoceanic links you have discovered. Some sites to try are

www.berkeley.edu (California)
www.mit.edu (Massachusetts)
www.vu.nl (Amsterdam)
www.ucl.ac.uk (London)
www.usyd.edu.au (Sydney)
www.u-tokyo.ac.jp (Tokyo)
www.uct.ac.za (Cape Town)

6

THE TRANSPORT LAYER

Together with the network layer, the transport layer is the heart of the protocol hierarchy. The network layer provides end-to-end packet delivery using datagrams or virtual circuits. The transport layer builds on the network layer to provide data transport from a process on a source machine to a process on a destination machine with a desired level of reliability that is independent of the physical networks currently in use. It provides the abstractions that applications need to use the network. 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 and choice of API design to tackle issues of reliability, connections and congestion control, protocols such as TCP and UDP, 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. To make the issue of transport service more concrete, we will examine two sets of transport layer primitives. First comes a simple (but hypothetical) one to show the basic ideas. Then comes the interface commonly used in the Internet.

6.1.1 Services Provided to the Upper Layers

The ultimate goal of the transport layer is to provide efficient, reliable, and cost-effective data transmission service to its users, normally processes in the application layer. To achieve this, the transport layer makes use of the services provided by the network layer. The software and/or hardware within the transport layer that does the work is called the **transport entity**. The transport entity can be located in the operating system kernel, in a library package bound into network applications, in a separate user process, or even on the network interface card. The first two options are most common on the Internet. The (logical) relationship of the network, transport, and application layers is illustrated in Fig. 6-1.

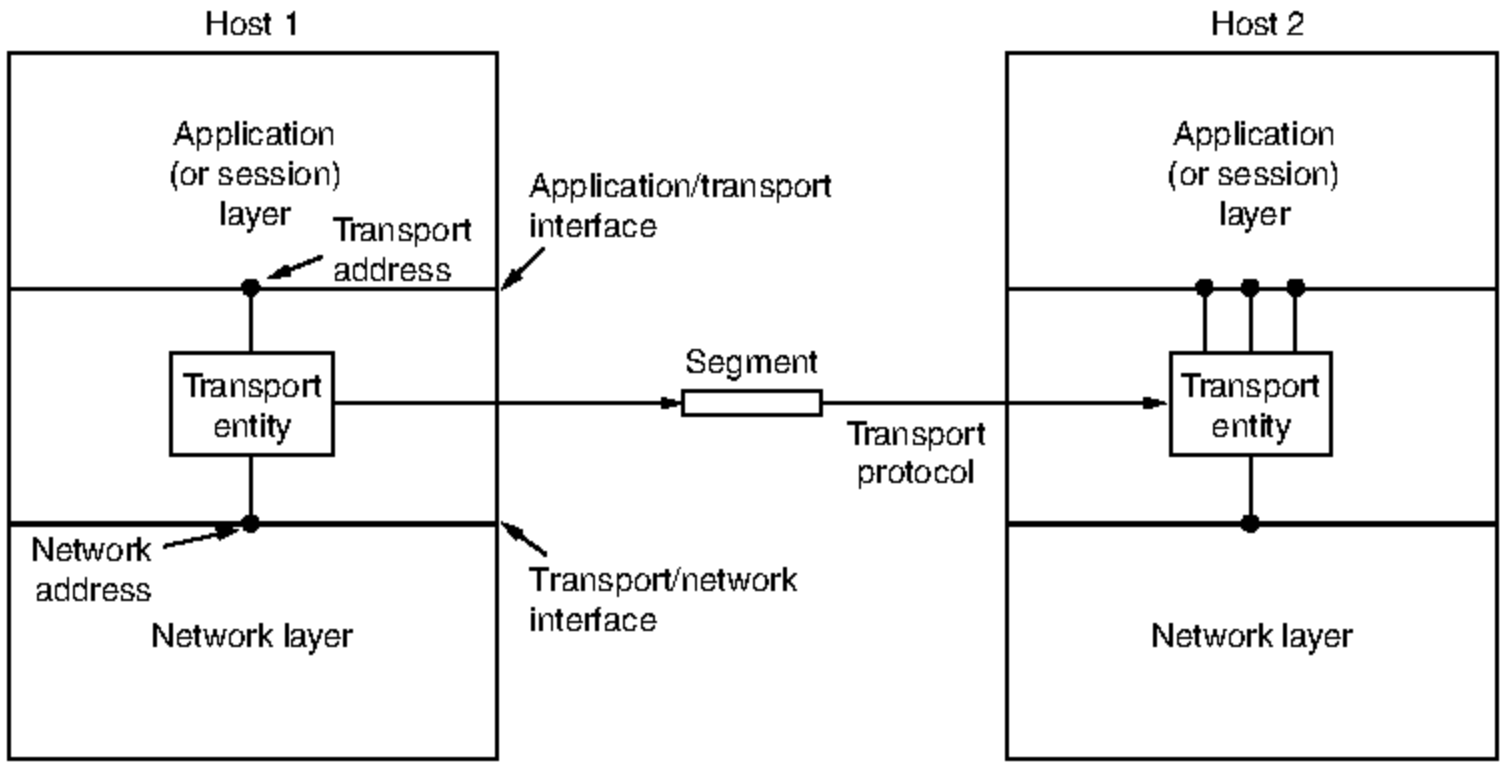


Figure 6-1. The network, transport, and application layers.

Just as there are two types of network service, connection-oriented and connectionless, there are also 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. However, note that it can be difficult to provide a connectionless transport service on top of a connection-oriented network service, since it is inefficient to set up a connection to send a single packet and then tear it down immediately afterwards.

The obvious question is this: 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. The transport code runs entirely on the users' machines, but the network layer mostly runs on the routers, which are operated by the carrier (at least for a wide area network). What happens if the network layer offers inadequate service? What if it frequently loses packets? What happens if routers crash from time to time?

Problems occur, that's what. The users have no real control over the network layer, so they cannot solve the problem of poor service by using better routers or putting more error handling in the data link layer because they don't own the routers. The only possibility is to put on top of the network layer another layer that improves the quality of the service. If, in a connectionless network, packets are lost or mangled, the transport entity can detect the problem and compensate for it by using retransmissions. If, in a connection-oriented network, 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 knowing where it was, 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. Furthermore, the transport primitives can be implemented as calls to library procedures to make them independent of the network primitives. The network service calls may vary considerably from one network to another (e.g., calls based on a connectionless Ethernet may be quite different from calls on a connection-oriented WiMAX network). Hiding the network service behind a set of transport service primitives ensures that changing the network merely requires replacing one set of library procedures with another one that does the same thing with a different underlying service.

Thanks to the transport layer, application programmers can write code according to a standard set of primitives and have these programs work on a wide variety of networks, without having to worry about dealing with different network interfaces and levels of reliability. If all real networks were flawless and all had the same service primitives and were guaranteed never, ever to change, the transport layer might 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 network.

For this reason, many people have made a qualitative 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. It is the level that applications see.

6.1.2 Transport Service Primitives

To allow users to access the transport service, the transport layer must provide some operations to application programs, that is, a transport service interface. Each transport service has its own interface. In this section, we will first examine a simple (hypothetical) transport service and its interface to see the bare essentials. In the following section, we will 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 on a single machine connected by a pipe in UNIX (or any other interprocess communication facility). They assume the connection between them is 100% perfect. They do not want to know about acknowledgements, lost packets, congestion, or anything at all like that. What they want is a 100% 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 even when they are on different machines.

As an aside, the transport layer can also provide unreliable (datagram) service. However, there is relatively little to say about that besides "it's datagrams," so we will mainly concentrate on the connection-oriented transport service in this chapter. Nevertheless, there are some applications, such as client-server computing and streaming multimedia, that build on a connectionless transport service, and we will say a little bit about that later on.

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. 6-2. 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 then release connections, which is sufficient for many applications.

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 that

Primitive	Packet 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 packet arrives
DISCONNECT	DISCONNECTION REQ.	Request a release of the connection

Figure 6-2. The primitives for a simple transport service.

blocks 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 use the term **segment** for messages sent from transport entity to transport entity. TCP, UDP and other Internet protocols use this term. Some older protocols used the ungainly name **TPDU** (**Transport Protocol Data Unit**). That term is not used much any more now but you may see it in older papers and books.

Thus, segments (exchanged by the transport layer) are contained in packets (exchanged by the network layer). In turn, these packets are contained in frames (exchanged by the data link layer). When a frame arrives, the data link layer processes the frame header and, if the destination address matches for local delivery, passes the contents of the frame payload field up to the network entity. The network entity similarly processes the packet header and then passes the contents of the packet payload up to the transport entity. This nesting is illustrated in Fig. 6-3.

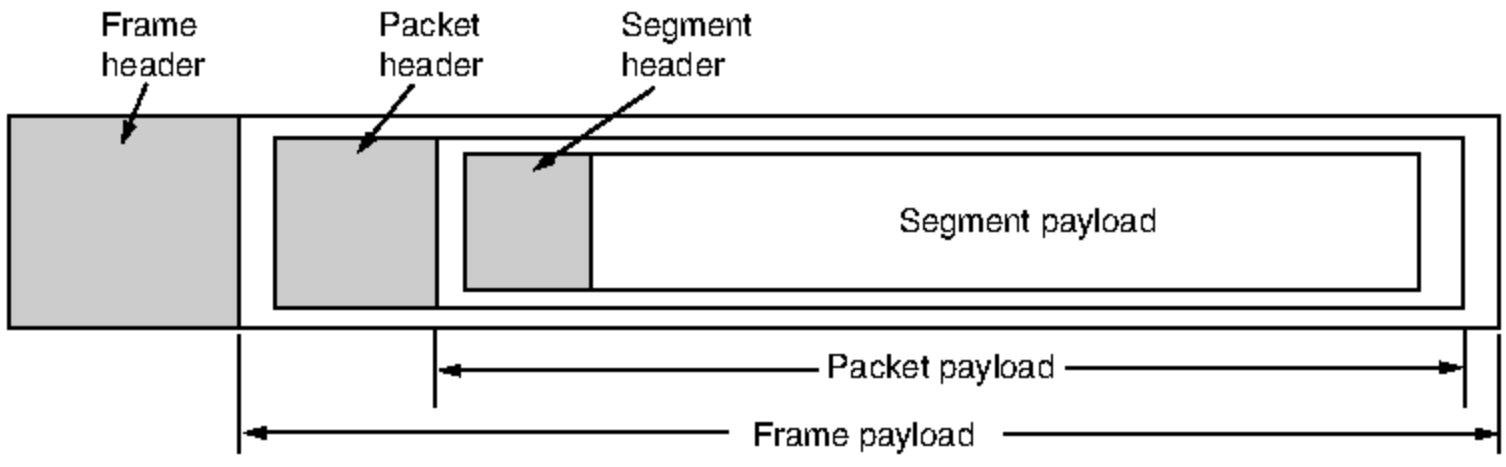


Figure 6-3. Nesting of segments, packets, and frames.

Getting back to our client-server example, the client's CONNECT call causes a CONNECTION REQUEST segment 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). If so, it then unblocks the server and sends a CONNECTION ACCEPTED segment back to the client. When this segment 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 segment arrives, the receiver is unblocked. It can then process the segment 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 in the transport layer, even a simple unidirectional data exchange is more complicated than at the network layer. Every data packet sent will also be acknowledged (eventually). The packets bearing control segments 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 need to worry about timers and retransmissions. None of this machinery is visible to the transport users. To the transport users, a connection is a reliable bit pipe: one user stuffs bits in and they magically appear in the same order 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 segment being sent to the remote transport entity. Upon its 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. 6-4. 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 segment 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 when we describe how TCP works.

6.1.3 Berkeley Sockets

Let us now briefly inspect another set of transport primitives, the socket primitives as they are used for TCP. Sockets were first released as part of the Berkeley UNIX 4.2BSD software distribution in 1983. They quickly became popular. The primitives are now widely used for Internet programming on many operating

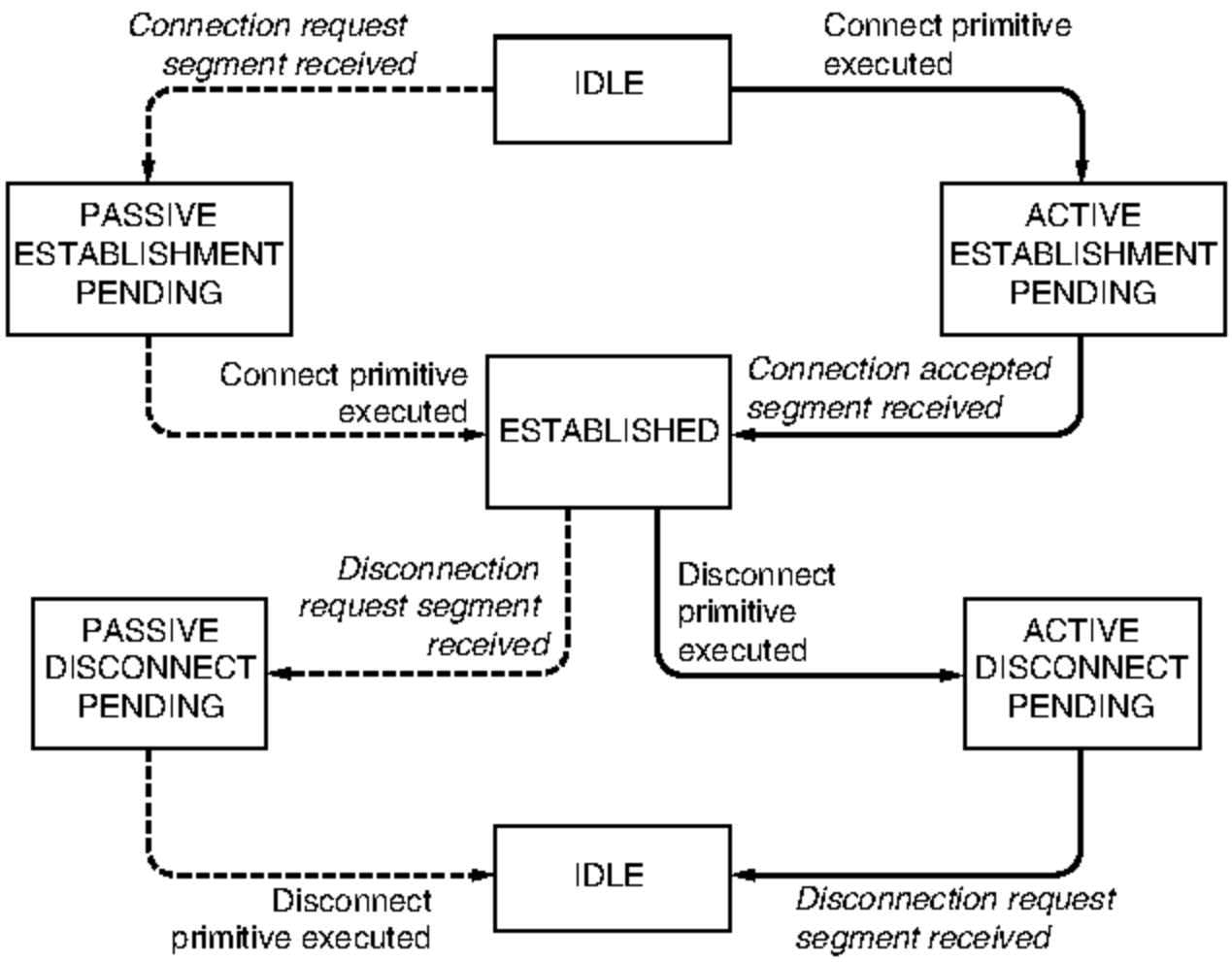


Figure 6-4. A state diagram for a simple connection management scheme. Transitions labeled in italics are caused by packet arrivals. The solid lines show the client's state sequence. The dashed lines show the server's state sequence.

systems, especially UNIX-based systems, and there is a socket-style API for Windows called "winsock."

The primitives are listed in Fig. 6-5. Roughly speaking, they follow the model of our first example but offer more features and flexibility. We will not look at the corresponding segments here. That discussion will come later.

Primitive	Meaning
SOCKET	Create a new communication endpoint
BIND	Associate a local address with a socket
LISTEN	Announce willingness to accept connections; give queue size
ACCEPT	Passively establish an incoming connection
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

Figure 6-5. The socket primitives for TCP.

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

Newly created sockets do not have network 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 addresses (e.g., they have been using the same address for years and everyone knows this address), whereas others do not.

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 segment 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. ACCEPT returns a file descriptor, which can be used for reading and writing in the standard way, the same as for files.

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 segment 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. The standard UNIX READ and WRITE system calls can also be used if none of the special options of SEND and RECEIVE are required.

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

Sockets have proved tremendously popular and are the de facto standard for abstracting transport services to applications. The socket API is often used with the TCP protocol to provide a connection-oriented service called a **reliable byte stream**, which is simply the reliable bit pipe that we described. However, other protocols could be used to implement this service using the same API. It should all be the same to the transport service users.

A strength of the socket API is that is can be used by an application for other transport services. For instance, sockets can be used with a connectionless transport service. In this case, CONNECT sets the address of the remote transport peer and SEND and RECEIVE send and receive datagrams to and from the remote peer.

(It is also common to use an expanded set of calls, for example, SENDTO and RECEIVEFROM, that emphasize messages and do not limit an application to a single transport peer.) Sockets can also be used with transport protocols that provide a message stream rather than a byte stream and that do or do not have congestion control. For example, **DCCP** (**Datagram Congestion Controlled Protocol**) is a version of UDP with congestion control (Kohler et al., 2006). It is up to the transport users to understand what service they are getting.

However, sockets are not likely to be the final word on transport interfaces. For example, applications often work with a group of related streams, such as a Web browser that requests several objects from the same server. With sockets, the most natural fit is for application programs to use one stream per object. This structure means that congestion control is applied separately for each stream, not across the group, which is suboptimal. It punts to the application the burden of managing the set. Newer protocols and interfaces have been devised that support groups of related streams more effectively and simply for the application. Two examples are **SCTP** (**Stream Control Transmission Protocol**) defined in RFC 4960 and **SST** (**Structured Stream Transport**) (Ford, 2007). These protocols must change the socket API slightly to get the benefits of groups of related streams, and they also support features such as a mix of connection-oriented and connectionless traffic and even multiple network paths. Time will tell if they are successful.

6.1.4 An Example of Socket Programming: An Internet File Server

As an example of the nitty-gritty of how real socket calls are made, consider the client and server code of Fig. 6-6. Here we have a very primitive Internet file server along with an example client that uses it. The code has many limitations (discussed below), but in principle the server code can be compiled and run on any UNIX system connected to the Internet. The client code can be compiled and run on any other UNIX machine on the Internet, anywhere in the world. The client code can be executed with appropriate parameters to fetch any file to which the server has access on its machine. The file is written to standard output, which, of course, can be redirected to a file or pipe.

Let us look at the server code first. It starts out by including some standard headers, the last three of which contain the main Internet-related definitions and data structures. Next comes a definition of *SERVER_PORT* as 12345. This number was chosen arbitrarily. Any number between 1024 and 65535 will work just as well, as long as it is not in use by some other process; ports below 1023 are reserved for privileged users.

The next two lines in the server define two constants needed. The first one determines the chunk size in bytes used for the file transfer. The second one determines how many pending connections can be held before additional ones are discarded upon arrival.

```
/* This page contains a client program that can request a file from the server program
 * on the next page. The server responds by sending the whole file.
 */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define SERVER_PORT 12345                       /* arbitrary, but client & server must agree */
#define BUF_SIZE 4096                           /* block transfer size */

int main(int argc, char **argv)
{
  int c, s, bytes;
  char buf[BUF_SIZE];                           /* buffer for incoming file */
  struct hostent *h;                            /* info about server */
  struct sockaddr_in channel;                   /* holds IP address */

  if (argc != 3) fatal("Usage: client server-name file-name");
  h = gethostbyname(argv[1]);                   /* look up host's IP address */
  if (!h) fatal("gethostbyname failed");

  s = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
  if (s <0) fatal("socket");
  memset(&channel, 0, sizeof(channel));
  channel.sin_family= AF_INET;
  memcpy(&channel.sin_addr.s_addr, h->h_addr, h->h_length);
  channel.sin_port= htons(SERVER_PORT);

  c = connect(s, (struct sockaddr *) &channel, sizeof(channel));
  if (c < 0) fatal("connect failed");

  /* Connection is now established. Send file name including 0 byte at end. */
  write(s, argv[2], strlen(argv[2])+1);

  /* Go get the file and write it to standard output. */
  while (1) {
      bytes = read(s, buf, BUF_SIZE);           /* read from socket */
      if (bytes <= 0) exit(0);                  /* check for end of file */
      write(1, buf, bytes);                     /* write to standard output */
  }
}

fatal(char *string)
{
  printf("%s\n", string);
  exit(1);
}
```

Figure 6-6. Client code using sockets. The server code is on the next page.

```
#include <sys/types.h>                          /* This is the server code */
#include <sys/fcntl.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define SERVER_PORT 12345                       /* arbitrary, but client & server must agree */
#define BUF_SIZE 4096                           /* block transfer size */
#define QUEUE_SIZE 10

int main(int argc, char *argv[])
{
  int s, b, l, fd, sa, bytes, on = 1;
  char buf[BUF_SIZE];                           /* buffer for outgoing file */
  struct sockaddr_in channel;                   /* holds IP address */

  /* Build address structure to bind to socket. */
  memset(&channel, 0, sizeof(channel));         /* zero channel */
  channel.sin_family = AF_INET;
  channel.sin_addr.s_addr = htonl(INADDR_ANY);
  channel.sin_port = htons(SERVER_PORT);

  /* Passive open. Wait for connection. */
  s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);    /* create socket */
  if (s < 0) fatal("socket failed");
  setsockopt(s, SOL_SOCKET, SO_REUSEADDR, (char *) &on, sizeof(on));
  b = bind(s, (struct sockaddr *) &channel, sizeof(channel));
  if (b < 0) fatal("bind failed");

  l = listen(s, QUEUE_SIZE);                    /* specify queue size */
  if (l < 0) fatal("listen failed");

  /* Socket is now set up and bound. Wait for connection and process it. */
  while (1) {
      sa = accept(s, 0, 0);                     /* block for connection request */
      if (sa < 0) fatal("accept failed");

      read(sa, buf, BUF_SIZE);                  /* read file name from socket */

      /* Get and return the file. */
      fd = open(buf, O_RDONLY);                 /* open the file to be sent back */
      if (fd < 0) fatal("open failed");

      while (1) {
          bytes = read(fd, buf, BUF_SIZE); /* read from file */
          if (bytes <= 0) break;                /* check for end of file */
          write(sa, buf, bytes);                /* write bytes to socket */
      }
      close(fd);                                /* close file */
      close(sa);                                /* close connection */
  }
}
```

After the declarations of local variables, the server code begins. It starts out by initializing a data structure that will hold the server's IP address. This data structure will soon be bound to the server's socket. The call to *memset* sets the data structure to all 0s. The three assignments following it fill in three of its fields. The last of these contains the server's port. The functions *htonl* and *htons* have to do with converting values to a standard format so the code runs correctly on both little-endian machines (e.g., Intel x86) and big-endian machines (e.g., the SPARC). Their exact semantics are not relevant here.

Next, the server creates a socket and checks for errors (indicated by $s < 0$). In a production version of the code, the error message could be a trifle more explanatory. The call to *setsockopt* is needed to allow the port to be reused so the server can run indefinitely, fielding request after request. Now the IP address is bound to the socket and a check is made to see if the call to *bind* succeeded. The final step in the initialization is the call to *listen* to announce the server's willingness to accept incoming calls and tell the system to hold up to *QUEUE_SIZE* of them in case new requests arrive while the server is still processing the current one. If the queue is full and additional requests arrive, they are quietly discarded.

At this point, the server enters its main loop, which it never leaves. The only way to stop it is to kill it from outside. The call to *accept* blocks the server until some client tries to establish a connection with it. If the *accept* call succeeds, it returns a socket descriptor that can be used for reading and writing, analogous to how file descriptors can be used to read from and write to pipes. However, unlike pipes, which are unidirectional, sockets are bidirectional, so *sa* (the accepted socket) can be used for reading from the connection and also for writing to it. A pipe file descriptor is for reading or writing but not both.

After the connection is established, the server reads the file name from it. If the name is not yet available, the server blocks waiting for it. After getting the file name, the server opens the file and enters a loop that alternately reads blocks from the file and writes them to the socket until the entire file has been copied. Then the server closes the file and the connection and waits for the next connection to show up. It repeats this loop forever.

Now let us look at the client code. To understand how it works, it is necessary to understand how it is invoked. Assuming it is called *client*, a typical call is

```
client flits.cs.vu.nl /usr/tom/filename >f
```

This call only works if the server is already running on *flits.cs.vu.nl* and the file */usr/tom/filename* exists and the server has read access to it. If the call is successful, the file is transferred over the Internet and written to *f*, after which the client program exits. Since the server continues after a transfer, the client can be started again and again to get other files.

The client code starts with some includes and declarations. Execution begins by checking to see if it has been called with the right number of arguments (*argc* = 3 means the program name plus two arguments). Note that *argv*[1] contains the

name of the server (e.g., *flits.cs.vu.nl*) and is converted to an IP address by *gethostbyname*. This function uses DNS to look up the name. We will study DNS in Chap. 7.

Next, a socket is created and initialized. After that, the client attempts to establish a TCP connection to the server, using *connect*. If the server is up and running on the named machine and attached to *SERVER_PORT* and is either idle or has room in its *listen* queue, the connection will (eventually) be established. Using the connection, the client sends the name of the file by writing on the socket. The number of bytes sent is one larger than the name proper, since the 0 byte terminating the name must also be sent to tell the server where the name ends.

Now the client enters a loop, reading the file block by block from the socket and copying it to standard output. When it is done, it just exits.

The procedure *fatal* prints an error message and exits. The server needs the same procedure, but it was omitted due to lack of space on the page. Since the client and server are compiled separately and normally run on different computers, they cannot share the code of *fatal*.

These two programs (as well as other material related to this book) can be fetched from the book's Web site

http://www.pearsonhighered.com/tanenbaum

Just for the record, this server is not the last word in serverdom. Its error checking is meager and its error reporting is mediocre. Since it handles all requests strictly sequentially (because it has only a single thread), its performance is poor. It has clearly never heard about security, and using bare UNIX system calls is not the way to gain platform independence. It also makes some assumptions that are technically illegal, such as assuming that the file name fits in the buffer and is transmitted atomically. These shortcomings notwithstanding, it is a working Internet file server. In the exercises, the reader is invited to improve it. For more information about programming with sockets, see Donahoo and Calvert (2008, 2009).

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, whether wired or wireless, whereas at the transport layer, this physical channel is replaced by the entire network. This difference has many important implications for the protocols.

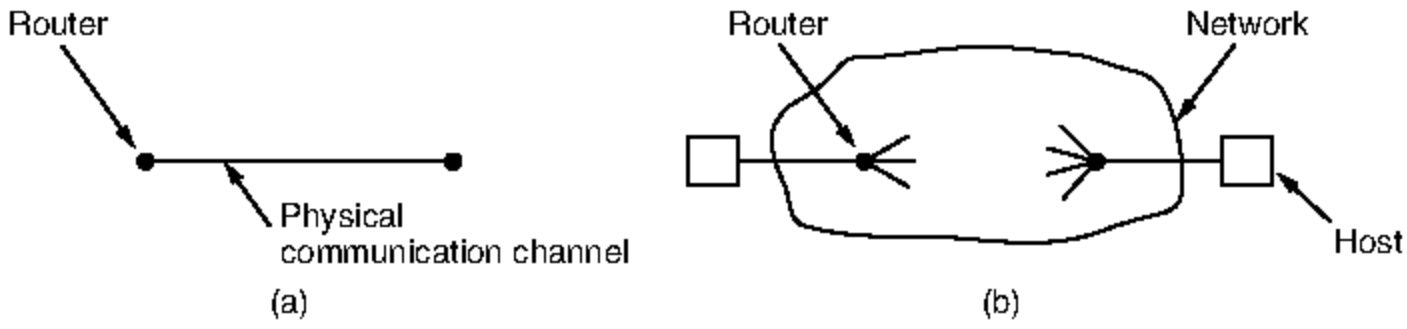


Figure 6-7. (a) Environment of the data link layer. (b) Environment of the transport layer.

For one thing, over point-to-point links such as wires or optical fiber, it is usually not necessary for a router to specify which router it wants to talk to—each outgoing line leads directly to 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. Even on wireless links, the process is not much different. Just sending a message is sufficient to have it reach all other destinations. If the message is not acknowledged due to an error, it can be resent. In the transport layer, initial connection establishment is 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 network. When a router sends a packet over a link, 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 suddenly emerge after other packets that were sent much later. If the network uses datagrams, which are independently routed inside, there is a nonnegligible probability that a packet may take the scenic route and arrive late and out of the expected order, or even that duplicates of the packet will arrive. The consequences of the network's ability to delay and duplicate packets can sometimes be disastrous and can require the use of special protocols to correctly transport information.

A final difference between the data link and transport layers is one of degree rather than of kind. Buffering and flow control are needed in both layers, but the presence in the transport layer of a large and varying number of connections with bandwidth that fluctuates as the connections compete with each other may require a different approach than we used in the data link layer. Some of the protocols discussed in Chap. 3 allocate a fixed number of buffers to each line, so that when a frame arrives a buffer is always available. In the transport layer, the larger number of connections that must be managed and variations in the bandwidth each

connection may receive 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 (e.g., a user) 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 endpoints are called **ports**. We will use the generic term **TSAP** (**Transport Service Access Point**) to mean a specific endpoint in the transport layer. The analogous endpoints in the network layer (i.e., network layer addresses) are not-surprisingly called **NSAPs** (**Network Service Access Points**). IP addresses are examples of NSAPs.

Figure 6-8 illustrates the relationship between the NSAPs, the TSAPs, and a transport connection. Application processes, both clients and servers, can attach themselves to a local TSAP to establish a connection to a remote TSAP. These connections run through NSAPs on each host, as shown. The purpose of having TSAPs is that in some networks, each computer has a single NSAP, so some way is needed to distinguish multiple transport endpoints that share that NSAP.

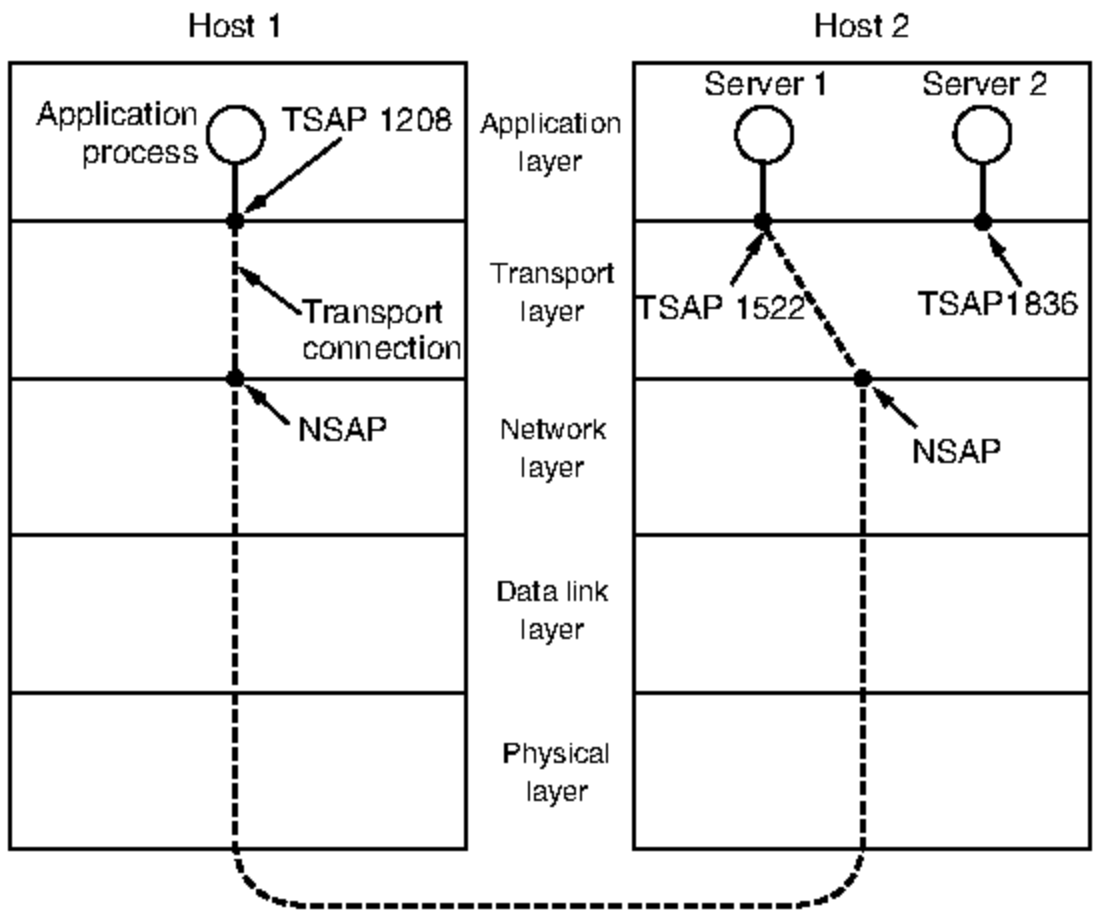


Figure 6-8. TSAPs, NSAPs, and transport connections.

A possible scenario for a transport connection is as follows:

1. A mail server process attaches itself to TSAP 1522 on host 2 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 send an email message, so it attaches itself to TSAP 1208 and issues a CONNECT request. The request specifies TSAP 1208 on host 1 as the source and TSAP 1522 on host 2 as the destination. This action ultimately results in a transport connection being established between the application process and the server.
3. The application process sends over the mail message.
4. The mail server responds to say that it will deliver the message.
5. The transport connection is released.

Note that there may well be other servers on host 2 that are attached to other TSAPs and are waiting for incoming connections that arrive over the same NSAP.

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 mail server is attached to TSAP 1522? One possibility is that the mail server has been attaching itself to TSAP 1522 for years and gradually all the network users have learned this. In this model, services have stable TSAP addresses that are listed in files in well-known places. For example, the */etc/services* file on UNIX systems lists which servers are permanently attached to which ports, including the fact that the mail server is found on TCP port 25.

While stable TSAP addresses work for a small number of key services that never change (e.g., the Web server), user processes, in general, often want to talk to other user processes that do not have TSAP addresses that are known in advance, or that may exist for only a short time.

To handle this situation, an alternative scheme can be used. In this scheme, there exists a special process called a **portmapper**. To find the TSAP address corresponding to a given service name, such as "BitTorrent," a user sets up a connection to the portmapper (which listens to a well-known TSAP). The user then sends a message specifying the service name, and the portmapper sends back the TSAP address. Then the user releases the connection with the portmapper and establishes a new one with the desired service.

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

The function of the portmapper is analogous to that of a 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 portmapper 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 sometime.

Many of the server processes that can exist on a machine will be used only rarely. It is wasteful to have each of them active and listening to a stable TSAP address all day long. An alternative scheme 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 services to remote users has a special **process server** that acts as a proxy for less heavily used servers. This server is called *inetd* on UNIX systems. It listens to a set of ports at the same time, waiting for a connection request. Potential users of a service begin by doing a CONNECT request, specifying the TSAP address 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).

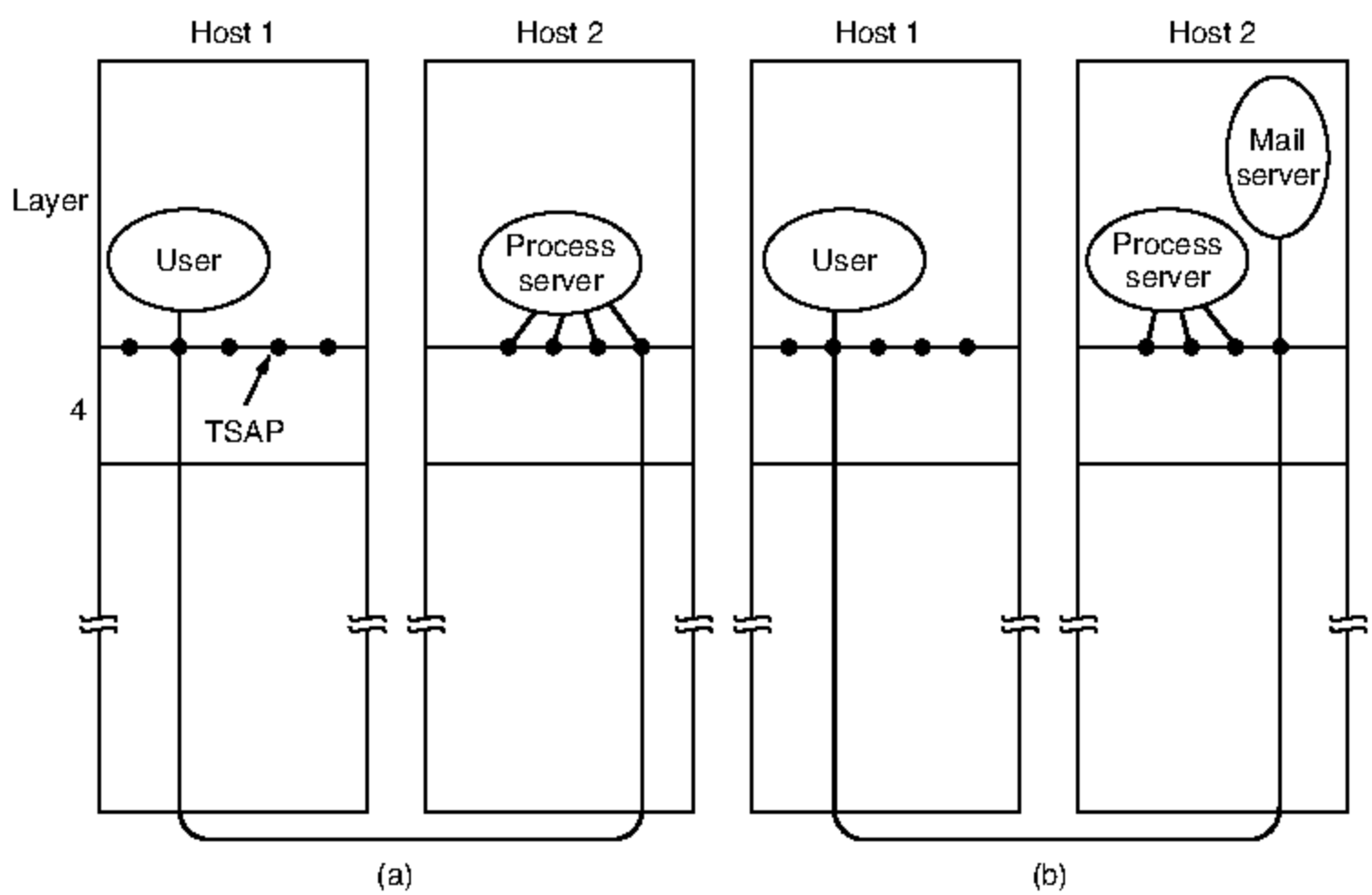


Figure 6-9. How a user process in host 1 establishes a connection with a mail server in host 2 via a process server.

After it gets the incoming request, the process server spawns the requested server, allowing it to inherit the existing connection with the user. The new server

does the requested work, while the process server goes back to listening for new requests, as shown in Fig. 6-9(b). This method is only applicable when servers can be created on demand.

6.2.2 Connection Establishment

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 segment to the destination and wait for a CONNECTION ACCEPTED reply. The problem occurs when the network can lose, delay, corrupt, and duplicate packets. This behavior causes serious complications.

Imagine a network 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 network uses datagrams inside and that every packet follows a different route. Some of the packets might get stuck in a traffic jam inside the network and take a long time to arrive. That is, they may be delayed in the network and pop out much later, when the sender thought that they had been lost.

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. Unfortunately, the packets decide to take the scenic route to the destination and go off exploring a remote corner of the network. The sender then times out and sends them all again. This time the packets take the shortest route and are delivered quickly so the sender releases the connection.

Unfortunately, eventually the initial batch of packets finally come out of hiding and arrive at the destination in order, asking the bank to establish a new connection and transfer money (again). 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.

This scenario may sound unlikely, or even implausible but the point is this: protocols must be designed to be correct in all cases. Only the common cases need be implemented efficiently to obtain good network performance, but the protocol must be able to cope with the uncommon cases without breaking. If it cannot, we have built a fair-weather network that can fail without warning when the conditions get tough.

For the remainder of this section, we will study the problem of delayed duplicates, with 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 that the delayed duplicates are thought to be new packets. We cannot prevent packets from being duplicated and delayed. But if and when this happens, the packets must be rejected as duplicates and not processed as fresh packets.

The problem 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 and never used again. Delayed duplicate packets then never find their way to a transport process and can do no damage. However, this approach makes it more difficult to connect with a process in the first place.

Another possibility is to give each connection a unique identifier (i.e., a sequence number incremented for each connection established) chosen by the initiating party and put in each segment, including the one requesting the connection. After each connection is released, each transport entity can update a table listing obsolete connections as (peer transport entity, connection identifier) pairs. Whenever a connection request comes in, it can be checked against the table to see if it belongs 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. This history must persist at both the source and destination machines. Otherwise, if a machine crashes and loses its memory, it will no longer know which connection identifiers have already been used by its peers.

Instead, we need to take a different tack to simplify the problem. Rather than allowing packets to live forever within the network, we devise a mechanism to kill off aged packets that are still hobbling about. With this restriction, the problem becomes somewhat more manageable.

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

1. Restricted network design.
2. Putting a hop counter in each packet.
3. Timestamping each packet.

The first technique includes any method that prevents packets from looping, combined with some way of bounding delay including congestion over the (now known) longest possible path. It is difficult, given that internets may range from a single city to international in scope. The second method consists of having the hop count initialized to some appropriate value and decremented each time the packet is forwarded. The network protocol simply discards any packet whose hop counter becomes zero. 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, and in practice a hop counter is a close enough approximation to age.

In practice, we will need to guarantee not only that a packet is dead, but also that all acknowledgements to it are dead, too, so we will now introduce a period T, which is some small multiple of the true maximum packet lifetime. The maximum packet lifetime is a conservative constant for a network; for the Internet, it is somewhat arbitrarily taken to be 120 seconds. The multiple is protocol dependent

and simply has the effect of making T longer. If we wait a time T secs 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 practical and foolproof way to reject delayed duplicate segments. The method described below is due to Tomlinson (1975), as refined by Sunshine and Dalal (1978). Variants of it are widely used in practice, including in TCP.

The heart of the method is for the source to label segments with sequence numbers that will not be reused within T secs. The period, T, and the rate of packets per second determine the size of the sequence numbers. In this way, only one packet with a given sequence number may be outstanding at any given time. Duplicates of this packet may still occur, and they must be discarded by the destination. However, it is no longer the case that a delayed duplicate of an old packet may beat a new packet with the same sequence number and be accepted by the destination in its stead.

To get around the problem of a machine losing all memory of where it was after a crash, one possibility is to require transport entities to be idle for T secs after a recovery. The idle period will let all old segments die off, so the sender can start again with any sequence number. However, in a complex internetwork, T may be large, so this strategy is unattractive.

Instead, 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.

When a connection is set up, the low-order k bits of the clock are used as the k-bit initial sequence number. Thus, unlike our protocols of Chap. 3, each connection starts numbering its segments with a different initial sequence number. The sequence space should be so large that by the time sequence numbers wrap around, old segments with the same sequence number are long gone. This linear relation between time and initial sequence numbers is shown in Fig. 6-10(a). The forbidden region shows the times for which segment sequence numbers are illegal leading up to their use. If any segment is sent with a sequence number in this region, it could be delayed and impersonate a different packet with the same sequence number that will be issued slightly later. For example, if the host crashes and restarts at time 70 seconds, it will use initial sequence numbers based on the clock to pick up after it left off; the host does not start with a lower sequence number in the forbidden region.

Once both transport entities have agreed on the initial sequence number, any sliding window protocol can be used for data flow control. This window protocol will correctly find and discard duplicates of packets after they have already been

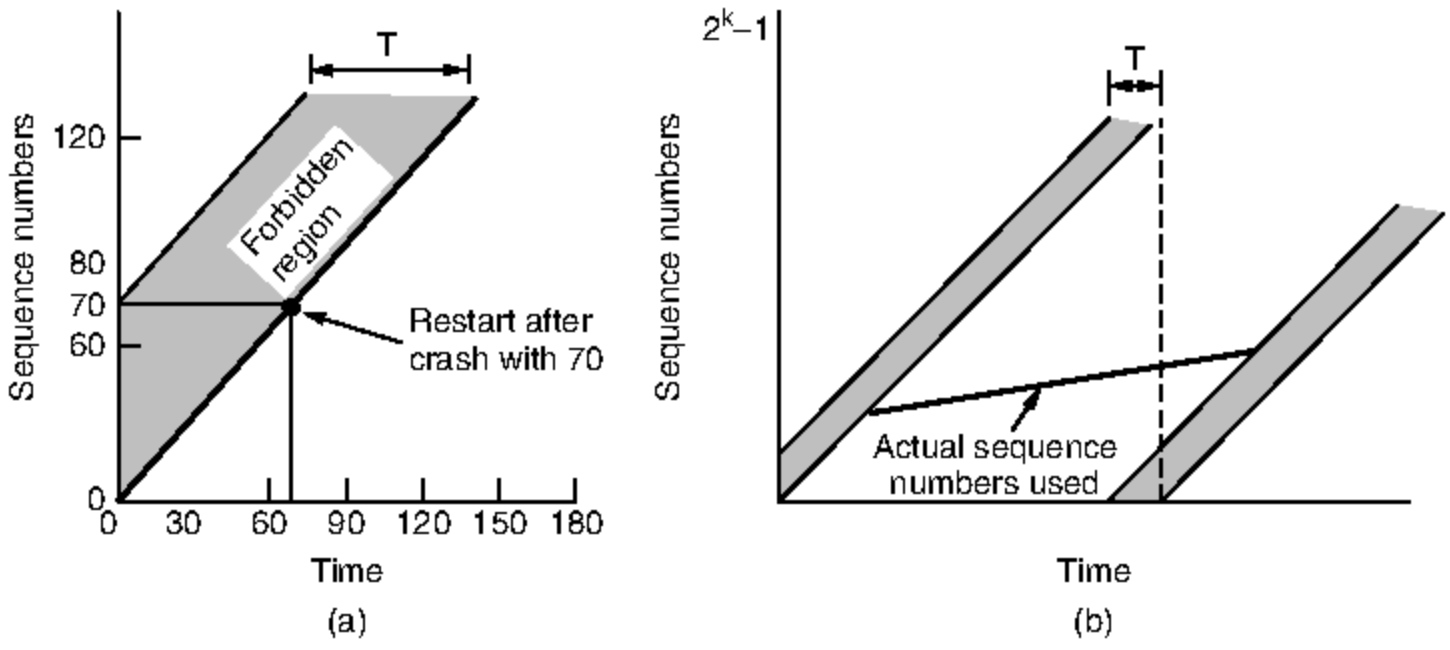


Figure 6-10. (a) Segments may not enter the forbidden region. (b) The resynchronization problem.

accepted. In reality, the initial sequence number curve (shown by the heavy line) is not linear, but a staircase, since the clock advances in discrete steps. For simplicity, we will ignore this detail.

To keep packet sequence numbers out of the forbidden region, we need to take care in two respects. We can get into trouble in two distinct 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, causing the sequence number to enter the forbidden region. To prevent this from happening, the maximum data rate on any connection is one segment per clock tick. This 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 in favor of a short clock tick (1 μsec or less). But the clock cannot tick too fast relative to the sequence number. For a clock rate of C and a sequence number space of size S, we must have $S/C > T$ so that the sequence numbers cannot wrap around too quickly.

Entering the forbidden region from underneath by sending too fast is not the only way to get into trouble. From Fig. 6-10(b), we see 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 as the sequence numbers wrap around. The greater the slope of the actual sequence numbers, the longer this event will be delayed. Avoiding this situation limits how slowly sequence numbers can advance on a connection (or how long the connections may last).

The clock-based method solves the problem of not being able to distinguish delayed duplicate segments from new segments. However, there is a practical snag for using it for establishing connections. Since we do not normally remember sequence numbers across connections at the destination, we still have no way of

knowing if a CONNECTION REQUEST segment containing an initial sequence number is a duplicate of a recent connection. This snag does not exist during a connection because the sliding window protocol does remember the current sequence number.

To solve this specific problem, Tomlinson (1975) introduced the **three-way handshake**. This establishment protocol involves one peer checking with the other that the connection request is indeed current. 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 segment containing it to host 2. Host 2 replies with an ACK segment 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 segment that it sends.

Now let us see how the three-way handshake works in the presence of delayed duplicate control segments. In Fig. 6-11(b), the first segment is a delayed duplicate CONNECTION REQUEST from an old connection. This segment arrives at host 2 without host 1's knowledge. Host 2 reacts to this segment by sending host 1 an ACK segment, 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 a connection, 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 ACK 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 segments containing sequence number y or acknowledgements to y are still in existence. When the second delayed segment 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 segments that can cause the protocol to fail and have a connection set up by accident when no one wants it.

TCP uses this three-way handshake to establish connections. Within a connection, a timestamp is used to extend the 32-bit sequence number so that it will not wrap within the maximum packet lifetime, even for gigabit-per-second connections. This mechanism is a fix to TCP that was needed as it was used on faster and faster links. It is described in RFC 1323 and called **PAWS** (**Protection Against Wrapped Sequence numbers**). Across connections, for the initial sequence numbers and before PAWS can come into play, TCP originally used the clock-based scheme just described. However, this turned out to have a security vulnerability. The clock made it easy for an attacker to predict the next initial sequence number and send packets that tricked the three-way handshake and established a forged connection. To close this hole, pseudorandom initial sequence numbers are used for connections in practice. However, it remains important that

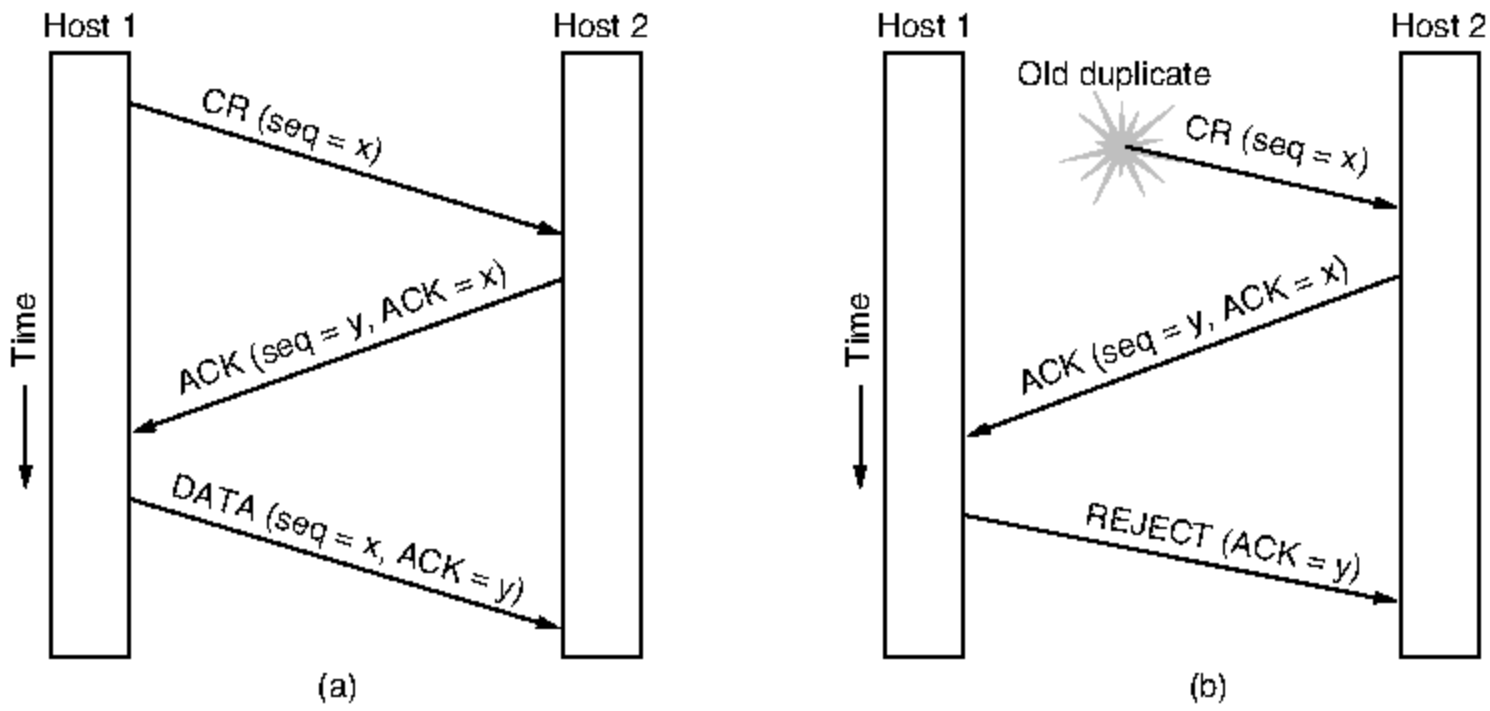


Figure 6-11. Three protocol scenarios for establishing a connection using a three-way handshake. CR denotes CONNECTION REQUEST. (a) Normal operation. (b) Old duplicate CONNECTION REQUEST appearing out of nowhere. (c) Duplicate CONNECTION REQUEST and duplicate ACK.

the initial sequence numbers not repeat for an interval even though they appear random to an observer. Otherwise, delayed duplicates can wreak havoc.

6.2.3 Connection Release

Releasing a connection is easier than establishing one. Nevertheless, there are more pitfalls than one might expect here. 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.

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 segment that arrives properly at host 2. Then host 1 sends another segment. Unfortunately, host 2 issues a DISCONNECT before the second segment arrives. The result is that the connection is released and data are lost.

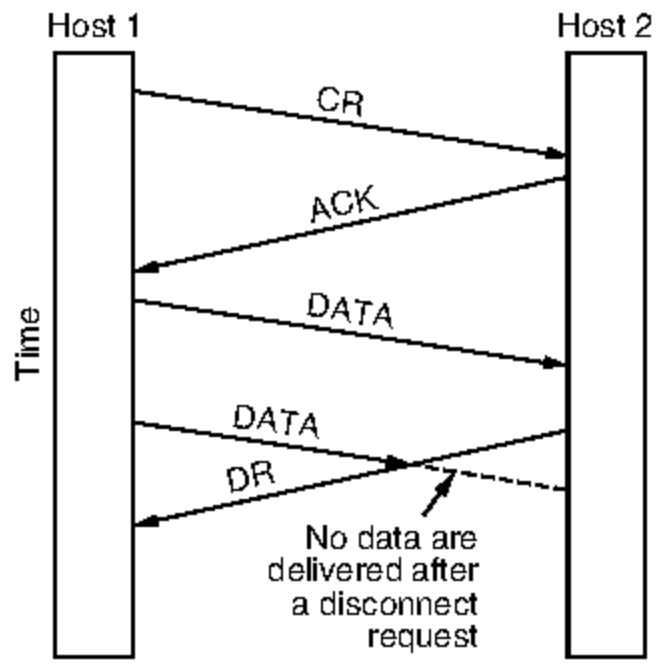


Figure 6-12. Abrupt disconnection with loss of data.

Clearly, a more sophisticated release protocol is needed 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 segment.

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 illustrates 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 the blue armies 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.

The blue armies want to synchronize their attacks. However, their only communication medium is to send messengers on foot down into the valley, where

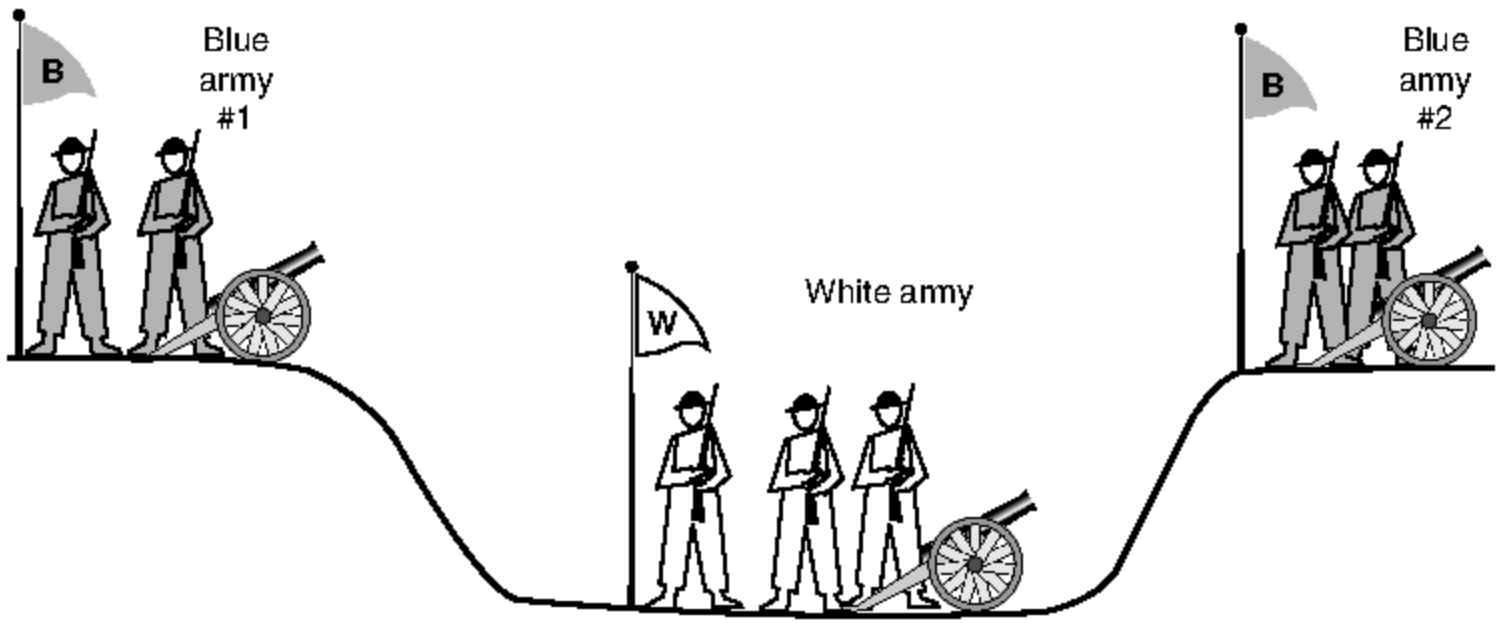


Figure 6-13. The two-army problem.

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, the commander of blue army #2 agrees, and 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, we can 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, rather than to military affairs, 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, we can avoid this quandary by foregoing the need for agreement and pushing the problem up to the transport user, letting each side independently decide when it is done. This is an easier problem to solve. 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) segment to initiate the connection release. When it arrives, the recipient sends back a DR segment and starts a timer, just in case its DR is lost. When this DR arrives, the original sender sends back an ACK segment and releases the connection. Finally, when the ACK segment 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 currently 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 segment 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 segments are lost and all segments are delivered correctly and on time.

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 segments. 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 and forcing it to go on forever until it gets a response. However, if the other side is allowed to time out, 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, the protocol hangs in Fig. 6-14(d).

One way to kill off half-open connections is to have a rule saying that if no segments 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. This rule also takes care of the case where the connection is broken (because the network can no longer deliver packets between the hosts) without either end disconnecting first. 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 segment is sent. If this timer expires, a

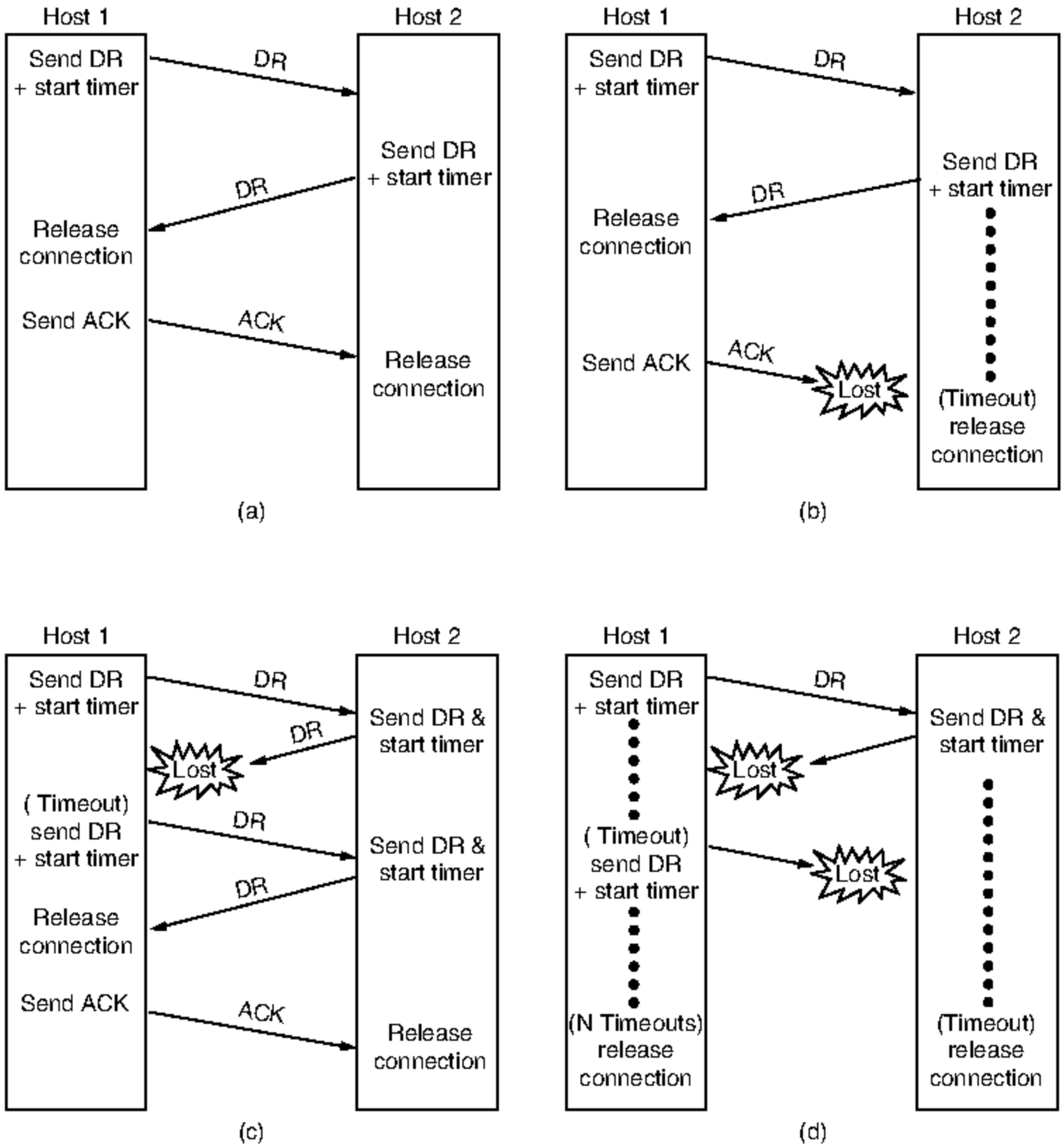


Figure 6-14. Four protocol scenarios for releasing a connection. (a) Normal case of three-way handshake. (b) Final ACK lost. (c) Response lost. (d) Response lost and subsequent DRs lost.

dummy segment 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 segments in a row are lost on an otherwise idle connection, first one side, then the other will automatically disconnect.

We will not belabor this point any more, but by now it should be clear that releasing a connection without data loss is not nearly as simple as it first appears. The lesson here is that the transport user must be involved in deciding when to

disconnect—the problem cannot be cleanly solved by the transport entities themselves. To see the importance of the application, consider that while TCP normally does a symmetric close (with each side independently closing its half of the connection with a FIN packet when it has sent its data), many Web servers send the client a RST packet that causes an abrupt close of the connection that is more like an asymmetric close. This works only because the Web server knows the pattern of data exchange. First it receives a request from the client, which is all the data the client will send, and then it sends a response to the client. When the Web server is finished with its response, all of the data has been sent in either direction. The server can send the client a warning and abruptly shut the connection. If the client gets this warning, it will release its connection state then and there. If the client does not get the warning, it will eventually realize that the server is no longer talking to it and release the connection state. The data has been successfully transferred in either case.

6.2.4 Error Control and Flow Control

Having examined connection establishment and release in some detail, let us now look at how connections are managed while they are in use. The key issues are error control and flow control. Error control is ensuring that the data is delivered with the desired level of reliability, usually that all of the data is delivered without any errors. Flow control is keeping a fast transmitter from overrunning a slow receiver.

Both of these issues have come up before, when we studied the data link layer. The solutions that are used at the transport layer are the same mechanisms that we studied in Chap. 3. As a very brief recap:

1. A frame carries an error-detecting code (e.g., a CRC or checksum) that is used to check if the information was correctly received.
2. A frame carries a sequence number to identify itself and is retransmitted by the sender until it receives an acknowledgement of successful receipt from the receiver. This is called **ARQ** (**Automatic Repeat reQuest**).
3. There is a maximum number of frames that the sender will allow to be outstanding at any time, pausing if the receiver is not acknowledging frames quickly enough. If this maximum is one packet the protocol is called **stop-and-wait**. Larger windows enable pipelining and improve performance on long, fast links.
4. The **sliding window** protocol combines these features and is also used to support bidirectional data transfer.

Given that these mechanisms are used on frames at the link layer, it is natural to wonder why they would be used on segments at the transport layer as well.

However, there is little duplication between the link and transport layers in practice. Even though the same mechanisms are used, there are differences in function and degree.

For a difference in function, consider error detection. The link layer checksum protects a frame while it crosses a single link. The transport layer checksum protects a segment while it crosses an entire network path. It is an end-to-end check, which is not the same as having a check on every link. Saltzer et al. (1984) describe a situation in which packets were corrupted inside a router. The link layer checksums protected the packets only while they traveled across a link, not while they were inside the router. Thus, packets were delivered incorrectly even though they were correct according to the checks on every link.

This and other examples led Saltzer et al. to articulate the **end-to-end argument**. According to this argument, the transport layer check that runs end-to-end is essential for correctness, and the link layer checks are not essential but nonetheless valuable for improving performance (since without them a corrupted packet can be sent along the entire path unnecessarily).

As a difference in degree, consider retransmissions and the sliding window protocol. Most wireless links, other than satellite links, can have only a single frame outstanding from the sender at a time. That is, the bandwidth-delay product for the link is small enough that not even a whole frame can be stored inside the link. In this case, a small window size is sufficient for good performance. For example, 802.11 uses a stop-and-wait protocol, transmitting or retransmitting each frame and waiting for it to be acknowledged before moving on to the next frame. Having a window size larger than one frame would add complexity without improving performance. For wired and optical fiber links, such as (switched) Ethernet or ISP backbones, the error-rate is low enough that link-layer retransmissions can be omitted because the end-to-end retransmissions will repair the residual frame loss.

On the other hand, many TCP connections have a bandwidth-delay product that is much larger than a single segment. Consider a connection sending data across the U.S. at 1 Mbps with a round-trip time of 100 msec. Even for this slow connection, 200 Kbit of data will be stored at the receiver in the time it takes to send a segment and receive an acknowledgement. For these situations, a large sliding window must be used. Stop-and-wait will cripple performance. In our example it would limit performance to one segment every 200 msec, or 5 segments/sec no matter how fast the network really is.

Given that transport protocols generally use larger sliding windows, we will look at the issue of buffering data more carefully. Since a host may have many connections, each of which is treated separately, it may need a substantial amount of buffering for the sliding windows. The buffers are needed at both the sender and the receiver. Certainly they are needed at the sender to hold all transmitted but as yet unacknowledged segments. They are needed there because these segments may be lost and need to be retransmitted.

However, since the sender is buffering, 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 segment comes in, an attempt is made to dynamically acquire a new buffer. If one is available, the segment is accepted; otherwise, it is discarded. Since the sender is prepared to retransmit segments lost by the network, no permanent harm is done by having the receiver drop segments, although some resources are wasted. The sender just keeps trying until it gets an acknowledgement.

The best 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 reasonable not to dedicate any buffers, but rather to acquire them dynamically at both ends, relying on buffering at the sender if segments must occasionally be discarded. 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. This is the strategy that TCP uses.

There still remains the question of how to organize the buffer pool. If most segments are nearly the same size, it is natural to organize the buffers as a pool of identically sized buffers, with one segment per buffer, as in Fig. 6-15(a). However, if there is wide variation in segment size, from short requests for Web pages to large packets in peer-to-peer file transfers, a pool of fixed-sized buffers presents problems. If the buffer size is chosen to be equal to the largest possible segment, space will be wasted whenever a short segment arrives. If the buffer size is chosen to be less than the maximum segment size, multiple buffers will be needed for long segments, with the attendant complexity.

Another approach to the buffer size problem is to use variable-sized 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 is simple and elegant and does not depend on segment sizes, but makes good use of memory only when the connections are heavily loaded.

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

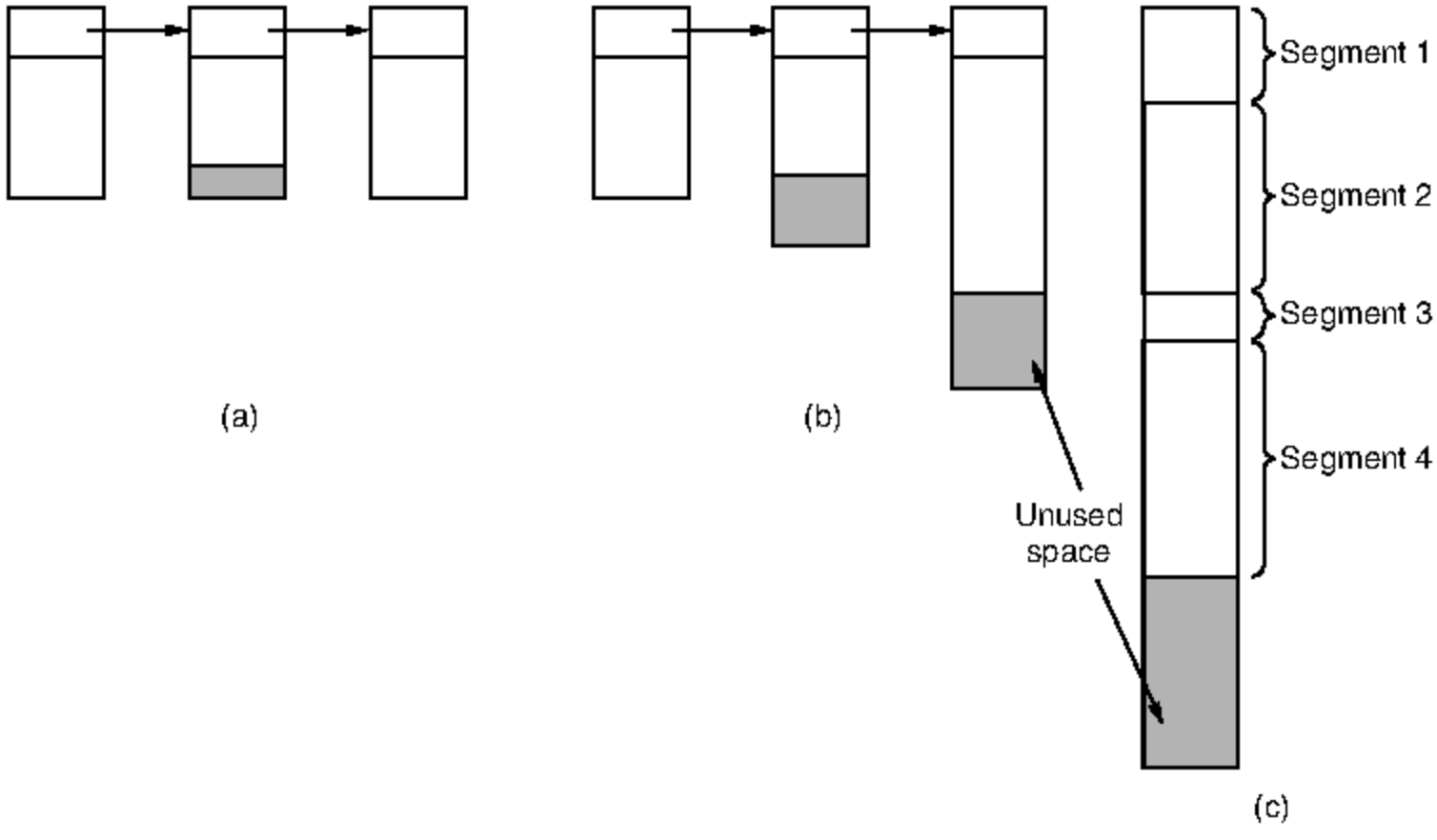


Figure 6-15. (a) Chained fixed-size buffers. (b) Chained variable-sized buffers. (c) One large circular buffer per connection.

window. Initially, the sender requests a certain number of buffers, based on its expected needs. The receiver then grants as many of these as it can afford. Every time the sender transmits a segment, it must decrement its allocation, stopping altogether when the allocation reaches zero. The receiver separately piggybacks both acknowledgements and buffer allocations onto the reverse traffic. TCP uses this scheme, carrying buffer allocations in a header field called *Window size*.

Figure 6-16 shows an example of how dynamic window management might work in a datagram network with 4-bit sequence numbers. In this example, data flows in segments from host *A* to host *B* and acknowledgements and buffer allocations flow in segments in the reverse direction. Initially, *A* wants eight buffers, but it is granted only four of these. It then sends three segments, of which the third is lost. Segment 6 acknowledges receipt of all segments 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 segments starting beyond 1 (i.e., segments 2, 3, and 4). *A* knows that it has already sent number 2, so it thinks that it may send segments 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 segments 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 segment from *B* to *A* allocates

another buffer and allows *A* to continue. This will happen when *B* has buffer space, likely because the transport user has accepted more segment data.

	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 1 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

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

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

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. This is often not the case. Memory was once expensive but prices have fallen dramatically. Hosts may be equipped with sufficient memory that the lack of buffers is rarely, if ever, a problem, even for wide area connections. Of course, this depends on the buffer size being set to be large enough, which has not always been the case for TCP (Zhang et al., 2002).

When buffer space no longer limits the maximum flow, another bottleneck will appear: the carrying capacity of the network. If adjacent routers can exchange at most x packets/sec and there are k disjoint paths between a pair of hosts, there is no way that those hosts can exchange more than kx segments/sec, no matter how much buffer space is available at each end. If the sender pushes too hard

(i.e., sends more than kx segments/sec), the network will become congested because it will be unable to deliver segments as fast as they are coming in.

What is needed is a mechanism that limits transmissions from the sender based on the network's carrying capacity rather than on the receiver's buffering capacity. 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. This means that a dynamic sliding window can implement both flow control and congestion control. If the network can handle c segments/sec and the round-trip time (including transmission, propagation, queueing, processing at the receiver, and return of the acknowledgement) is r, 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. Since the network capacity available to any given flow varies over time, the window size should be adjusted frequently, to track changes in the carrying capacity. As we will see later, TCP uses a similar scheme.

6.2.5 Multiplexing

Multiplexing, or sharing several conversations over 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, if only one network address is available on a host, all transport connections on that machine have to use it. When a segment comes in, some way is needed to tell which process to give it to. This situation, called **multiplexing**, is shown in Fig. 6-17(a). In this figure, four distinct transport connections all use the same network connection (e.g., IP address) to the remote host.

Multiplexing can also be useful in the transport layer for another reason. Suppose, for example, that a host has multiple network paths that it can use. If a user needs more bandwidth or more reliability than one of the network paths can provide, a way out is to have a connection that distributes the traffic among multiple network paths on a round-robin basis, as indicated in Fig. 6-17(b). This modus operandi is called **inverse multiplexing**. With k network connections open, the effective bandwidth might be increased by a factor of k. An example of inverse multiplexing is **SCTP** (**Stream Control Transmission Protocol**), which can run a connection using multiple network interfaces. In contrast, TCP uses a single network endpoint. Inverse multiplexing is also found at the link layer, when several low-rate links are used in parallel as one high-rate link.

6.2.6 Crash Recovery

If hosts and routers are subject to crashes or connections are long-lived (e.g., large software or media downloads), recovery from these crashes becomes an issue. If the transport entity is entirely within the hosts, recovery from network

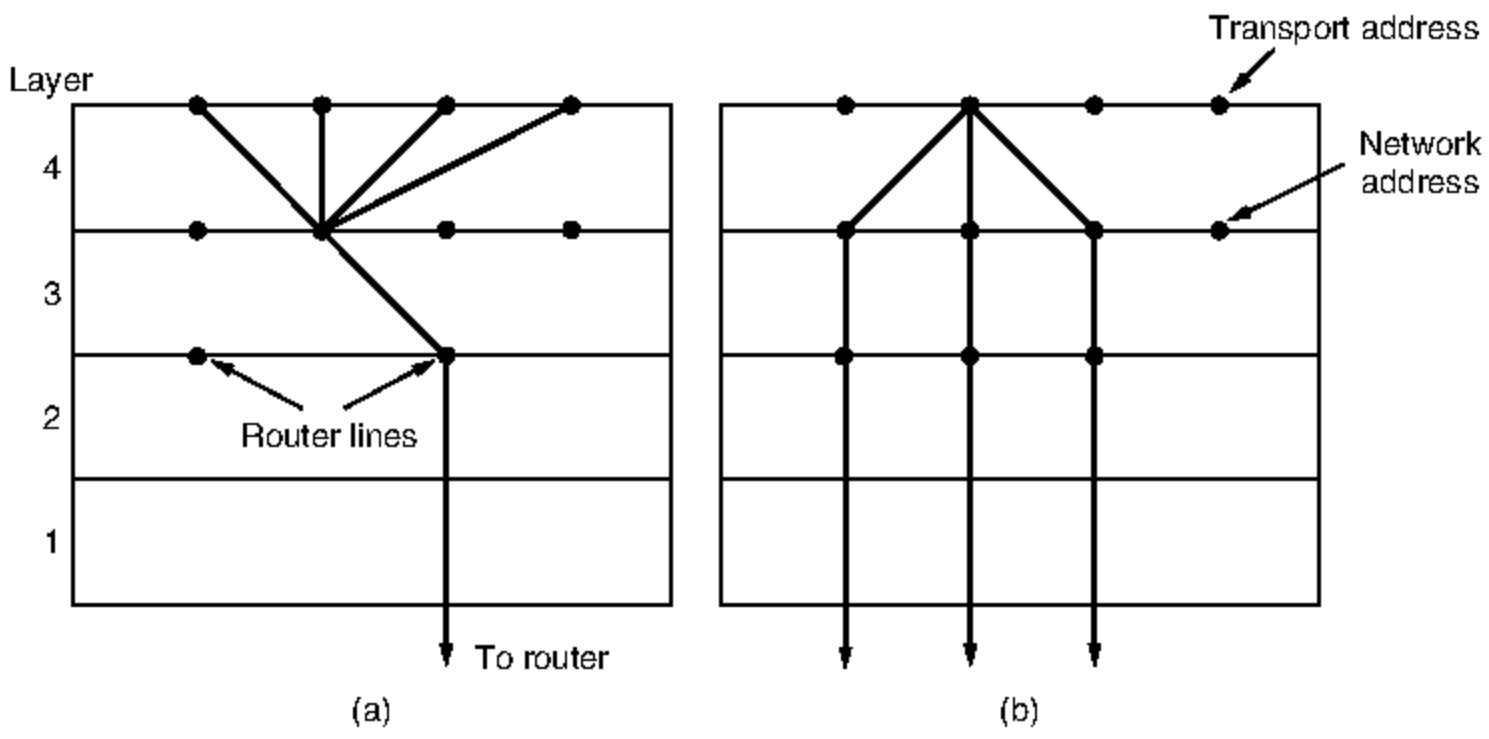


Figure 6-17. (a) Multiplexing. (b) Inverse multiplexing.

and router crashes is straightforward. The transport entities expect lost segments all the time and know how to cope with them by using retransmissions.

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 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 just passes the incoming segments to the transport user, one by one. Partway 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 segment to all other hosts, announcing that it has 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 segment outstanding, *S1*, or no segments outstanding, *S0*. Based on only this state information, the client must decide whether to retransmit the most recent segment.

At first glance, it would seem obvious: the client should retransmit if and only if it has an unacknowledged segment 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 in which the server's transport entity first sends an acknowledgement and then, when the acknowledgement has been sent, writes to the application process. Writing a segment onto the output stream and sending an acknowledgement are two distinct events that cannot be done simultaneously. If a crash occurs after the acknowledgement has been sent but before the write has been fully completed, 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 segment has arrived. This decision by the client leads to a missing segment.

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 segment in the output stream to the server application process.

No matter how the client and server 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 segment, never retransmit the last segment, 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 can follow C (i.e., once it has crashed, it has crashed). Figure 6-18 shows all eight combinations of client and server strategies 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

Figure 6-18. Different combinations of client and server strategies.

Making the protocol more elaborate does not help. Even if the client and server exchange several segments 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—that is, separate events happen one after another not at the same time—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 to reconstruct where it was before the problem occurred. This is consistent with the case mentioned above that 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 segments 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 CONGESTION CONTROL

If the transport entities on many machines send too many packets into the network too quickly, the network will become congested, with performance degraded as packets are delayed and lost. Controlling congestion to avoid this problem is the combined responsibility of the network and transport layers. Congestion occurs at routers, so it is detected at the network layer. However, congestion is ultimately caused by traffic sent into the network by the transport layer. The only effective way to control congestion is for the transport protocols to send packets into the network more slowly.

In Chap. 5, we studied congestion control mechanisms in the network layer. In this section, we will study the other half of the problem, congestion control mechanisms in the transport layer. After describing the goals of congestion control, we will describe how hosts can regulate the rate at which they send packets into the network. The Internet relies heavily on the transport layer for congestion control, and specific algorithms are built into TCP and other protocols.

6.3.1 Desirable Bandwidth Allocation

Before we describe how to regulate traffic, we must understand what we are trying to achieve by running a congestion control algorithm. That is, we must specify the state in which a good congestion control algorithm will operate the network. The goal is more than to simply avoid congestion. It is to find a good allocation of bandwidth to the transport entities that are using the network. A good allocation will deliver good performance because it uses all the available bandwidth but avoids congestion, it will be fair across competing transport entities, and it will quickly track changes in traffic demands. We will make each of these criteria more precise in turn.

Efficiency and Power

An efficient allocation of bandwidth across transport entities will use all of the network capacity that is available. However, it is not quite right to think that if there is a 100-Mbps link, five transport entities should get 20 Mbps each. They should usually get less than 20 Mbps for good performance. The reason is that the traffic is often bursty. Recall that in Sec. 5.3 we described the **goodput** (or rate of useful packets arriving at the receiver) as a function of the offered load. This curve and a matching curve for the delay as a function of the offered load are given in Fig. 6-19.

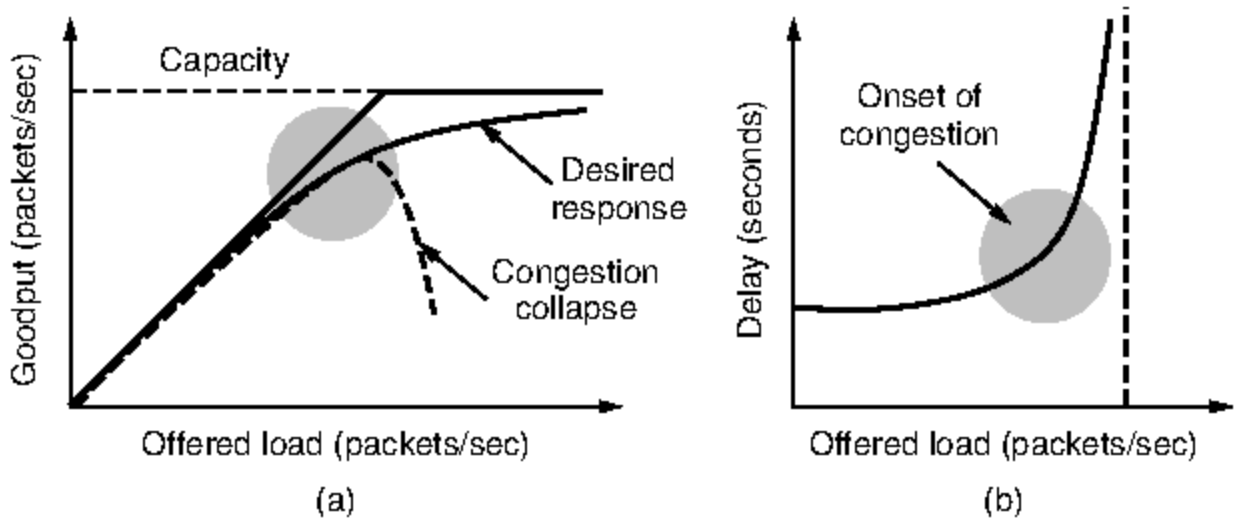


Figure 6-19. (a) Goodput and (b) delay as a function of offered load.

As the load increases in Fig. 6-19(a) goodput initially increases at the same rate, but as the load approaches the capacity, goodput rises more gradually. This falloff is because bursts of traffic can occasionally mount up and cause some losses at buffers inside the network. If the transport protocol is poorly designed and retransmits packets that have been delayed but not lost, the network can enter congestion collapse. In this state, senders are furiously sending packets, but increasingly little useful work is being accomplished.

The corresponding delay is given in Fig. 6-19(b) Initially the delay is fixed, representing the propagation delay across the network. As the load approaches the capacity, the delay rises, slowly at first and then much more rapidly. This is again because of bursts of traffic that tend to mound up at high load. The delay cannot really go to infinity, except in a model in which the routers have infinite buffers. Instead, packets will be lost after experiencing the maximum buffering delay.

For both goodput and delay, performance begins to degrade at the onset of congestion. Intuitively, we will obtain the best performance from the network if we allocate bandwidth up until the delay starts to climb rapidly. This point is below the capacity. To identify it, Kleinrock (1979) proposed the metric of **power**, where

$$power = \frac{load}{delay}$$

Power will initially rise with offered load, as delay remains small and roughly constant, but will reach a maximum and fall as delay grows rapidly. The load with the highest power represents an efficient load for the transport entity to place on the network.

Max-Min Fairness

In the preceding discussion, we did not talk about how to divide bandwidth between different transport senders. This sounds like a simple question to answer—give all the senders an equal fraction of the bandwidth—but it involves several considerations.

Perhaps the first consideration is to ask what this problem has to do with congestion control. After all, if the network gives a sender some amount of bandwidth to use, the sender should just use that much bandwidth. However, it is often the case that networks do not have a strict bandwidth reservation for each flow or connection. They may for some flows if quality of service is supported, but many connections will seek to use whatever bandwidth is available or be lumped together by the network under a common allocation. For example, IETF's differentiated services separates traffic into two classes and connections compete for bandwidth within each class. IP routers often have all connections competing for the same bandwidth. In this situation, it is the congestion control mechanism that is allocating bandwidth to the competing connections.

A second consideration is what a fair portion means for flows in a network. It is simple enough if N flows use a single link, in which case they can all have $1/N$ of the bandwidth (although efficiency will dictate that they use slightly less if the traffic is bursty). But what happens if the flows have different, but overlapping, network paths? For example, one flow may cross three links, and the other flows may cross one link. The three-link flow consumes more network resources. It might be fairer in some sense to give it less bandwidth than the one-link flows. It

should certainly be possible to support more one-link flows by reducing the bandwidth of the three-link flow. This point demonstrates an inherent tension between fairness and efficiency.

However, we will adopt a notion of fairness that does not depend on the length of the network path. Even with this simple model, giving connections an equal fraction of bandwidth is a bit complicated because different connections will take different paths through the network and these paths will themselves have different capacities. In this case, it is possible for a flow to be bottlenecked on a downstream link and take a smaller portion of an upstream link than other flows; reducing the bandwidth of the other flows would slow them down but would not help the bottlenecked flow at all.

The form of fairness that is often desired for network usage is **max-min fairness**. An allocation is max-min fair if the bandwidth given to one flow cannot be increased without decreasing the bandwidth given to another flow with an allocation that is no larger. That is, increasing the bandwidth of a flow will only make the situation worse for flows that are less well off.

Let us see an example. A max-min fair allocation is shown for a network with four flows, *A*, *B*, *C*, and *D*, in Fig. 6-20. Each of the links between routers has the same capacity, taken to be 1 unit, though in the general case the links will have different capacities. Three flows compete for the bottom-left link between routers *R4* and *R5*. Each of these flows therefore gets 1/3 of the link. The remaining flow, *A*, competes with *B* on the link from *R2* to *R3*. Since *B* has an allocation of 1/3, *A* gets the remaining 2/3 of the link. Notice that all of the other links have spare capacity. However, this capacity cannot be given to any of the flows without decreasing the capacity of another, lower flow. For example, if more of the bandwidth on the link between *R2* and *R3* is given to flow *B*, there will be less for flow *A*. This is reasonable as flow *A* already has more bandwidth. However, the capacity of flow *C* or *D* (or both) must be decreased to give more bandwidth to *B*, and these flows will have less bandwidth than *B*. Thus, the allocation is max-min fair.

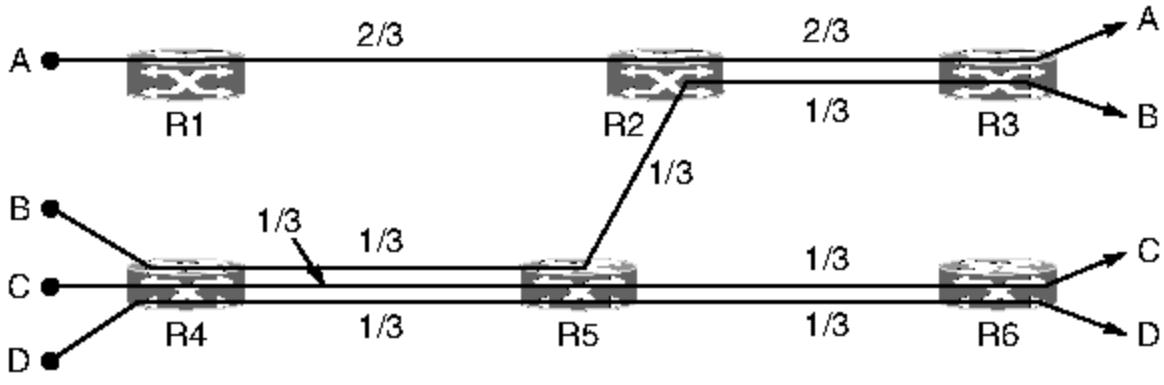


Figure 6-20. Max-min bandwidth allocation for four flows.

Max-min allocations can be computed given a global knowledge of the network. An intuitive way to think about them is to imagine that the rate for all of the

flows starts at zero and is slowly increased. When the rate reaches a bottleneck for any flow, then that flow stops increasing. The other flows all continue to increase, sharing equally in the available capacity, until they too reach their respective bottlenecks.

A third consideration is the level over which to consider fairness. A network could be fair at the level of connections, connections between a pair of hosts, or all connections per host. We examined this issue when we were discussing WFQ (Weighted Fair Queueing) in Sec. 5.4 and concluded that each of these definitions has its problems. For example, defining fairness per host means that a busy server will fare no better than a mobile phone, while defining fairness per connection encourages hosts to open more connections. Given that there is no clear answer, fairness is often considered per connection, but precise fairness is usually not a concern. It is more important in practice that no connection be starved of bandwidth than that all connections get precisely the same amount of bandwidth. In fact, with TCP it is possible to open multiple connections and compete for bandwidth more aggressively. This tactic is used by bandwidth-hungry applications such as BitTorrent for peer-to-peer file sharing.

Convergence

A final criterion is that the congestion control algorithm converge quickly to a fair and efficient allocation of bandwidth. The discussion of the desirable operating point above assumes a static network environment. However, connections are always coming and going in a network, and the bandwidth needed by a given connection will vary over time too, for example, as a user browses Web pages and occasionally downloads large videos.

Because of the variation in demand, the ideal operating point for the network varies over time. A good congestion control algorithm should rapidly converge to the ideal operating point, and it should track that point as it changes over time. If the convergence is too slow, the algorithm will never be close to the changing operating point. If the algorithm is not stable, it may fail to converge to the right point in some cases, or even oscillate around the right point.

An example of a bandwidth allocation that changes over time and converges quickly is shown in Fig. 6-21. Initially, flow 1 has all of the bandwidth. One second later, flow 2 starts. It needs bandwidth as well. The allocation quickly changes to give each of these flows half the bandwidth. At 4 seconds, a third flow joins. However, this flow uses only 20% of the bandwidth, which is less than its fair share (which is a third). Flows 1 and 2 quickly adjust, dividing the available bandwidth to each have 40% of the bandwidth. At 9 seconds, the second flow leaves, and the third flow remains unchanged. The first flow quickly captures 80% of the bandwidth. At all times, the total allocated bandwidth is approximately 100%, so that the network is fully used, and competing flows get equal treatment (but do not have to use more bandwidth than they need).

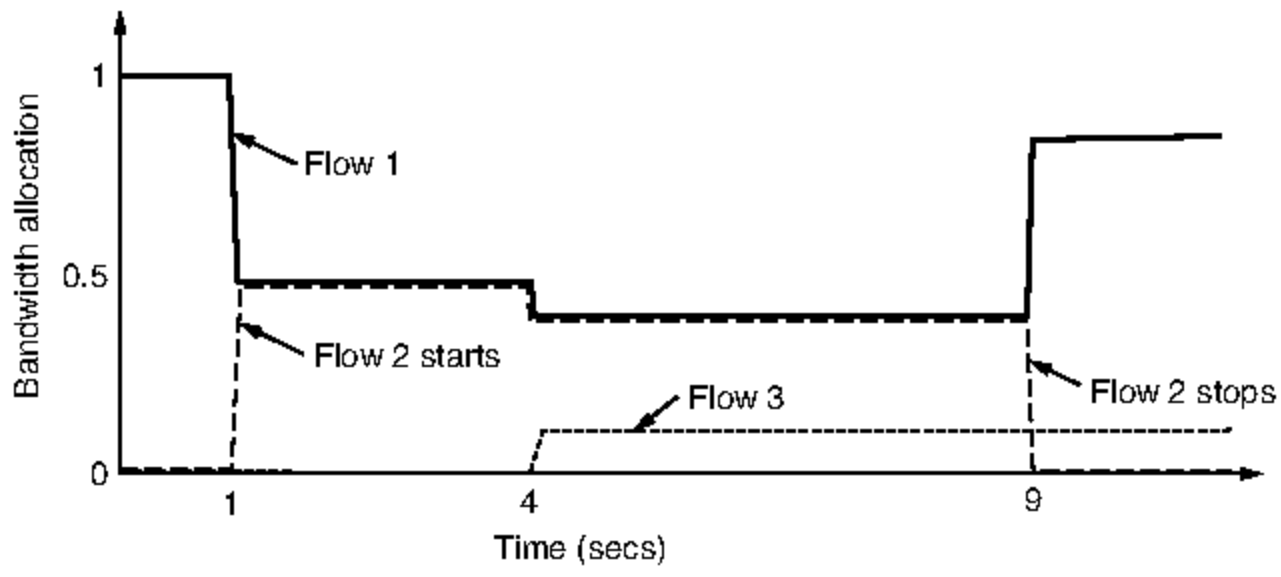


Figure 6-21. Changing bandwidth allocation over time.

6.3.2 Regulating the Sending Rate

Now it is time for the main course. How do we regulate the sending rates to obtain a desirable bandwidth allocation? The sending rate may be limited by two factors. The first is flow control, in the case that there is insufficient buffering at the receiver. The second is congestion, in the case that there is insufficient capacity in the network. In Fig. 6-22, we see this problem illustrated hydraulically. In Fig. 6-22(a), we see a thick pipe leading to a small-capacity receiver. This is a flow-control limited situation. As long as the sender does not send more water than the bucket can contain, no water will be lost. In Fig. 6-22(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).

These cases may appear similar to the sender, as transmitting too fast causes packets to be lost. However, they have different causes and call for different solutions. We have already talked about a flow-control solution with a variable-sized window. Now we will consider a congestion control solution. Since either of these problems can occur, the transport protocol will in general need to run both solutions and slow down if either problem occurs.

The way that a transport protocol should regulate the sending rate depends on the form of the feedback returned by the network. Different network layers may return different kinds of feedback. The feedback may be explicit or implicit, and it may be precise or imprecise.

An example of an explicit, precise design is when routers tell the sources the rate at which they may send. Designs in the literature such as XCP (eXplicit Congestion Protocol) operate in this manner (Katabi et al., 2002). An explicit, imprecise design is the use of ECN (Explicit Congestion Notification) with TCP. In this design, routers set bits on packets that experience congestion to warn the senders to slow down, but they do not tell them how much to slow down.

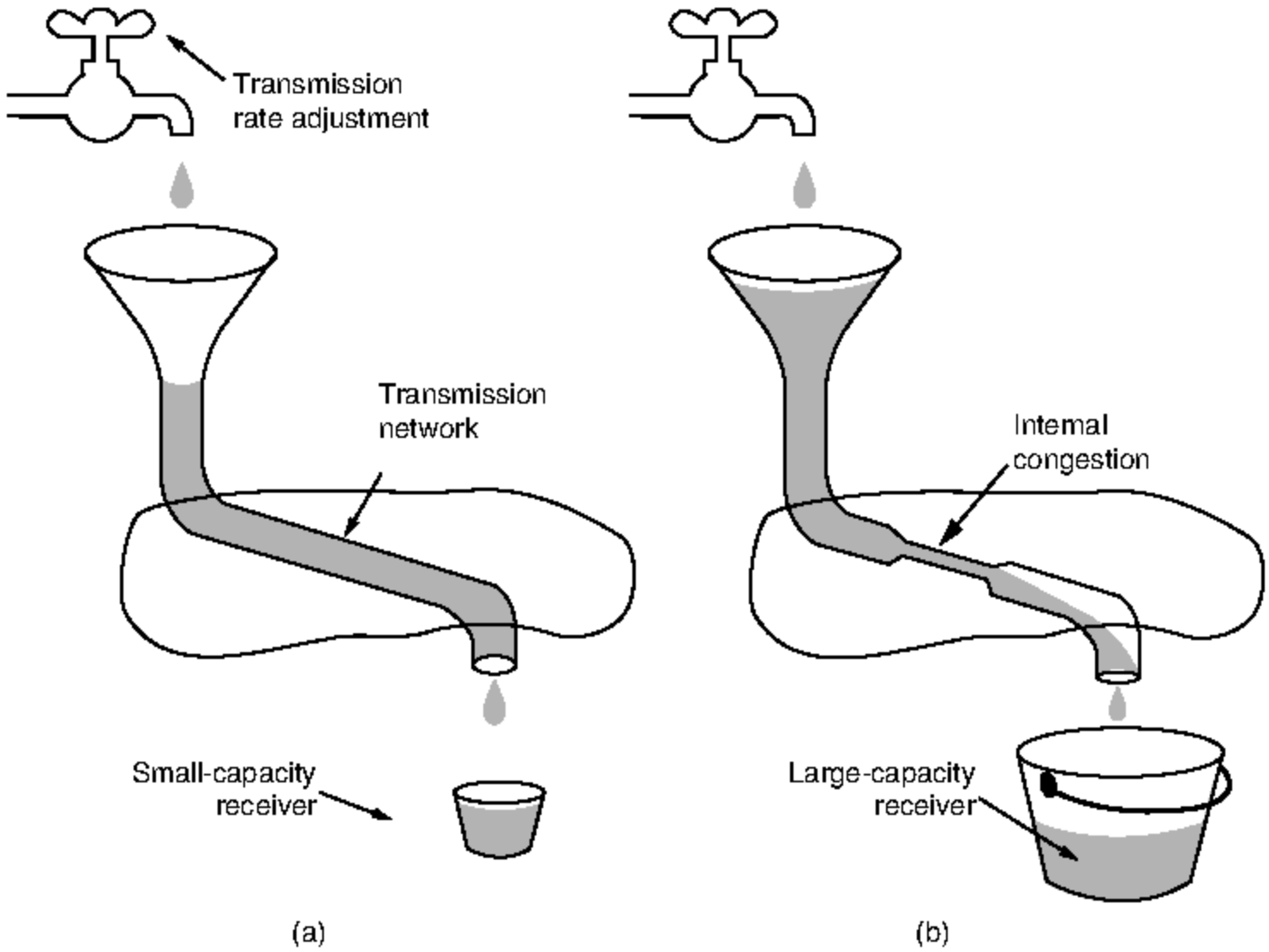


Figure 6-22. (a) A fast network feeding a low-capacity receiver. (b) A slow network feeding a high-capacity receiver.

In other designs, there is no explicit signal. FAST TCP measures the round-trip delay and uses that metric as a signal to avoid congestion (Wei et al., 2006). Finally, in the form of congestion control most prevalent in the Internet today, TCP with drop-tail or RED routers, packet loss is inferred and used to signal that the network has become congested. There are many variants of this form of TCP, including CUBIC TCP, which is used in Linux (Ha et al., 2008). Combinations are also possible. For example, Windows includes Compound TCP that uses both packet loss and delay as feedback signals (Tan et al., 2006). These designs are summarized in Fig. 6-23.

If an explicit and precise signal is given, the transport entity can use that signal to adjust its rate to the new operating point. For example, if XCP tells senders the rate to use, the senders may simply use that rate. In the other cases, however, some guesswork is involved. In the absence of a congestion signal, the senders should decrease their rates. When a congestion signal is given, the senders should decrease their rates. The way in which the rates are increased or decreased is given by a **control law**. These laws have a major effect on performance.

Protocol	Signal	Explicit?	Precise?
XCP	Rate to use	Yes	Yes
TCP with ECN	Congestion warning	Yes	No
FAST TCP	End-to-end delay	No	Yes
Compound TCP	Packet loss & end-to-end delay	No	Yes
CUBIC TCP	Packet loss	No	No
TCP	Packet loss	No	No

Figure 6-23. Signals of some congestion control protocols.

Chiu and Jain (1989) studied the case of binary congestion feedback and concluded that **AIMD** (**Additive Increase Multiplicative Decrease**) is the appropriate control law to arrive at the efficient and fair operating point. To argue this case, they constructed a graphical argument for the simple case of two connections competing for the bandwidth of a single link. The graph in Fig. 6-24 shows the bandwidth allocated to user 1 on the x-axis and to user 2 on the y-axis. When the allocation is fair, both users will receive the same amount of bandwidth. This is shown by the dotted fairness line. When the allocations sum to 100%, the capacity of the link, the allocation is efficient. This is shown by the dotted efficiency line. A congestion signal is given by the network to both users when the sum of their allocations crosses this line. The intersection of these lines is the desired operating point, when both users have the same bandwidth and all of the network bandwidth is used.

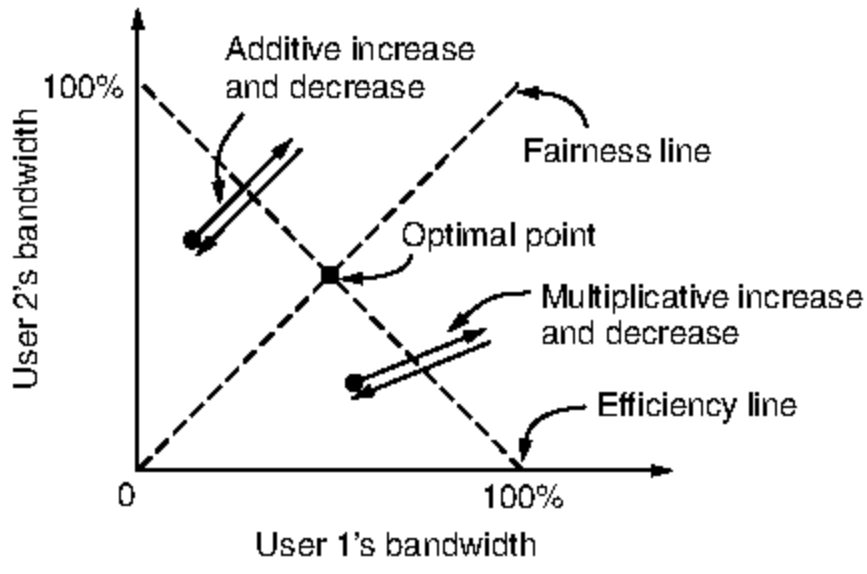


Figure 6-24. Additive and multiplicative bandwidth adjustments.

Consider what happens from some starting allocation if both user 1 and user 2 additively increase their respective bandwidths over time. For example, the users may each increase their sending rate by 1 Mbps every second. Eventually, the

operating point crosses the efficiency line and both users receive a congestion signal from the network. At this stage, they must reduce their allocations. However, an additive decrease would simply cause them to oscillate along an additive line. This situation is shown in Fig. 6-24. The behavior will keep the operating point close to efficient, but it will not necessarily be fair.

Similarly, consider the case when both users multiplicatively increase their bandwidth over time until they receive a congestion signal. For example, the users may increase their sending rate by 10% every second. If they then multiplicatively decrease their sending rates, the operating point of the users will simply oscillate along a multiplicative line. This behavior is also shown in Fig. 6-24. The multiplicative line has a different slope than the additive line. (It points to the origin, while the additive line has an angle of 45 degrees.) But it is otherwise no better. In neither case will the users converge to the optimal sending rates that are both fair and efficient.

Now consider the case that the users additively increase their bandwidth allocations and then multiplicatively decrease them when congestion is signaled. This behavior is the AIMD control law, and it is shown in Fig. 6-25. It can be seen that the path traced by this behavior does converge to the optimal point that is both fair and efficient. This convergence happens no matter what the starting point, making AIMD broadly useful. By the same argument, the only other combination, multiplicative increase and additive decrease, would diverge from the optimal point.

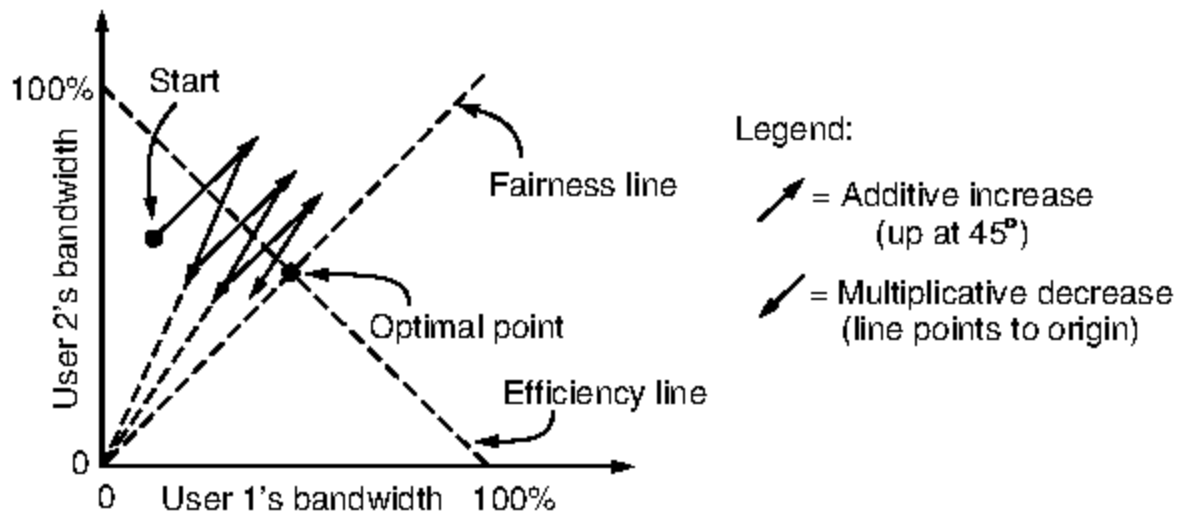


Figure 6-25. Additive Increase Multiplicative Decrease (AIMD) control law.

AIMD is the control law that is used by TCP, based on this argument and another stability argument (that it is easy to drive the network into congestion and difficult to recover, so the increase policy should be gentle and the decrease policy aggressive). It is not quite fair, since TCP connections adjust their window size by a given amount every round-trip time. Different connections will have different round-trip times. This leads to a bias in which connections to closer hosts receive more bandwidth than connections to distant hosts, all else being equal.

In Sec. 6.5, we will describe in detail how TCP implements an AIMD control law to adjust the sending rate and provide congestion control. This task is more difficult than it sounds because rates are measured over some interval and traffic is bursty. Instead of adjusting the rate directly, a strategy that is often used in practice is to adjust the size of a sliding window. TCP uses this strategy. If the window size is W and the round-trip time is RTT, the equivalent rate is W/RTT. This strategy is easy to combine with flow control, which already uses a window, and has the advantage that the sender paces packets using acknowledgements and hence slows down in one RTT if it stops receiving reports that packets are leaving the network.

As a final issue, there may be many different transport protocols that send traffic into the network. What will happen if the different protocols compete with different control laws to avoid congestion? Unequal bandwidth allocations, that is what. Since TCP is the dominant form of congestion control in the Internet, there is significant community pressure for new transport protocols to be designed so that they compete fairly with it. The early streaming media protocols caused problems by excessively reducing TCP throughput because they did not compete fairly. This led to the notion of **TCP-friendly** congestion control in which TCP and non-TCP transport protocols can be freely mixed with no ill effects (Floyd et al., 2000).

6.3.3 Wireless Issues

Transport protocols such as TCP that implement congestion control should be independent of the underlying network and link layer technologies. That is a good theory, but in practice there are issues with wireless networks. The main issue is that packet loss is often used as a congestion signal, including by TCP as we have just discussed. Wireless networks lose packets all the time due to transmission errors.

With the AIMD control law, high throughput requires very small levels of packet loss. Analyses by Padhye et al. (1998) show that the throughput goes up as the inverse square-root of the packet loss rate. What this means in practice is that the loss rate for fast TCP connections is very small; 1% is a moderate loss rate, and by the time the loss rate reaches 10% the connection has effectively stopped working. However, for wireless networks such as 802.11 LANs, frame loss rates of at least 10% are common. This difference means that, absent protective measures, congestion control schemes that use packet loss as a signal will unnecessarily throttle connections that run over wireless links to very low rates.

To function well, the only packet losses that the congestion control algorithm should observe are losses due to insufficient bandwidth, not losses due to transmission errors. One solution to this problem is to mask the wireless losses by using retransmissions over the wireless link. For example, 802.11 uses a stop-and-wait protocol to deliver each frame, retrying transmissions multiple times if

need be before reporting a packet loss to the higher layer. In the normal case, each packet is delivered despite transient transmission errors that are not visible to the higher layers.

Fig. 6-26 shows a path with a wired and wireless link for which the masking strategy is used. There are two aspects to note. First, the sender does not necessarily know that the path includes a wireless link, since all it sees is the wired link to which it is attached. Internet paths are heterogeneous and there is no general method for the sender to tell what kind of links comprise the path. This complicates the congestion control problem, as there is no easy way to use one protocol for wireless links and another protocol for wired links.

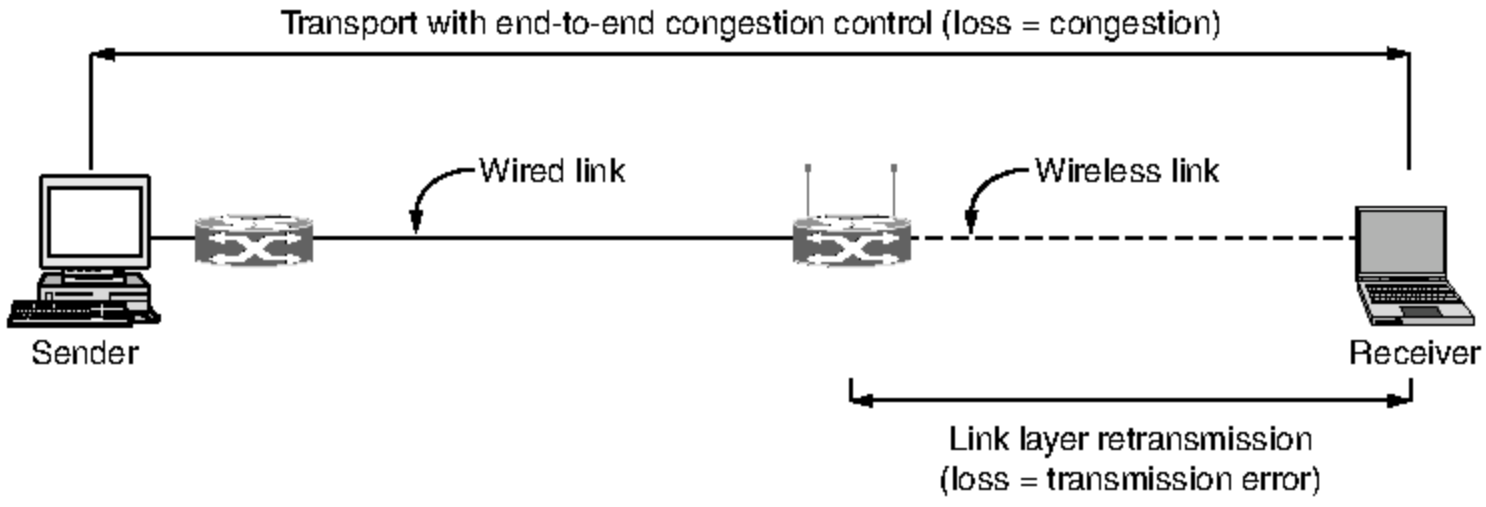


Figure 6-26. Congestion control over a path with a wireless link.

The second aspect is a puzzle. The figure shows two mechanisms that are driven by loss: link layer frame retransmissions, and transport layer congestion control. The puzzle is how these two mechanisms can co-exist without getting confused. After all, a loss should cause only one mechanism to take action because it is either a transmission error or a congestion signal. It cannot be both. If both mechanisms take action (by retransmitting the frame and slowing down the sending rate) then we are back to the original problem of transports that run far too slowly over wireless links. Consider this puzzle for a moment and see if you can solve it.

The solution is that the two mechanisms act at different timescales. Link layer retransmissions happen on the order of microseconds to milliseconds for wireless links such as 802.11. Loss timers in transport protocols fire on the order of milliseconds to seconds. The difference is three orders of magnitude. This allows wireless links to detect frame losses and retransmit frames to repair transmission errors long before packet loss is inferred by the transport entity.

The masking strategy is sufficient to let most transport protocols run well across most wireless links. However, it is not always a fitting solution. Some wireless links have long round-trip times, such as satellites. For these links other techniques must be used to mask loss, such as FEC (Forward Error Correction), or the transport protocol must use a non-loss signal for congestion control.

A second issue with congestion control over wireless links is variable capacity. That is, the capacity of a wireless link changes over time, sometimes abruptly, as nodes move and the signal-to-noise ratio varies with the changing channel conditions. This is unlike wired links whose capacity is fixed. The transport protocol must adapt to the changing capacity of wireless links, otherwise it will either congest the network or fail to use the available capacity.

One possible solution to this problem is simply not to worry about it. This strategy is feasible because congestion control algorithms must already handle the case of new users entering the network or existing users changing their sending rates. Even though the capacity of wired links is fixed, the changing behavior of other users presents itself as variability in the bandwidth that is available to a given user. Thus it is possible to simply run TCP over a path with an 802.11 wireless link and obtain reasonable performance.

However, when there is much wireless variability, transport protocols designed for wired links may have trouble keeping up and deliver poor performance. The solution in this case is a transport protocol that is designed for wireless links. A particularly challenging setting is a wireless mesh network in which multiple, interfering wireless links must be crossed, routes change due to mobility, and there is lots of loss. Research in this area is ongoing. See Li et al. (2009) for an example of wireless transport protocol design.

6.4 THE INTERNET TRANSPORT PROTOCOLS: UDP

The Internet has two main protocols in the transport layer, a connectionless protocol and a connection-oriented one. The protocols complement each other. The connectionless protocol is UDP. It does almost nothing beyond sending packets between applications, letting applications build their own protocols on top as needed. The connection-oriented protocol is TCP. It does almost everything. It makes connections and adds reliability with retransmissions, along with flow control and congestion control, all on behalf of the applications that use it.

In the following sections, we will study UDP and TCP. We will start with UDP because it is simplest. We will also look at two uses of UDP. Since UDP is a transport layer protocol that typically runs in the operating system and protocols that use UDP typically run in user space, these uses might be considered applications. However, the techniques they use are useful for many applications and are better considered to belong to a transport service, so we will cover them here.

6.4.1 Introduction to UDP

The Internet protocol suite supports a connectionless transport protocol called **UDP** (**User Datagram Protocol**). UDP provides a way for applications to send encapsulated IP datagrams without having to establish a connection. UDP is described in RFC 768.

UDP transmits **segments** consisting of an 8-byte header followed by the payload. The header is shown in Fig. 6-27. The two **ports** serve to identify the endpoints within the source and destination machines. When a UDP packet arrives, its payload is handed to the process attached to the destination port. This attachment occurs when the BIND primitive or something similar is used, as we saw in Fig. 6-6 for TCP (the binding process is the same for UDP). Think of ports as mailboxes that applications can rent to receive packets. We will have more to say about them when we describe TCP, which also uses ports. In fact, the main value of UDP over just using raw IP is the addition of the source and destination ports. Without the port fields, the transport layer would not know what to do with each incoming packet. With them, it delivers the embedded segment to the correct application.

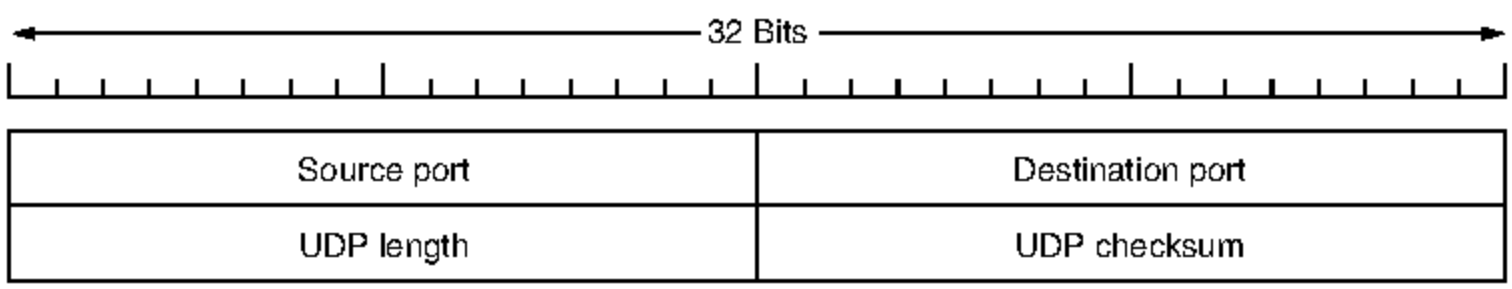


Figure 6-27. The UDP header.

The source port is primarily needed when a reply must be sent back to the source. By copying the *Source port* field from the incoming segment into the *Destination port* field of the outgoing segment, the process sending the reply can specify which process on the sending machine is to get it.

The *UDP length* field includes the 8-byte header and the data. The minimum length is 8 bytes, to cover the header. The maximum length is 65,515 bytes, which is lower than the largest number that will fit in 16 bits because of the size limit on IP packets.

An optional *Checksum* is also provided for extra reliability. It checksums the header, the data, and a conceptual IP pseudoheader. When performing this computation, the *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 one's complement and to take the one'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. If the checksum is not computed, it is stored as a 0, since by a happy coincidence of one's complement arithmetic a true computed 0 is stored as all 1s. However, turning it off is foolish unless the quality of the data does not matter (e.g., for digitized speech).

The pseudoheader for the case of IPv4 is shown in Fig. 6-28. It contains the 32-bit IPv4 addresses of the source and destination machines, the protocol number for UDP (17), and the byte count for the UDP segment (including the header). It

is different but analogous for IPv6. Including the pseudoheader in the UDP checksum computation helps detect misdelivered packets, but including it also violates the protocol hierarchy since the IP addresses in it belong to the IP layer, not to the UDP layer. TCP uses the same pseudoheader for its checksum.

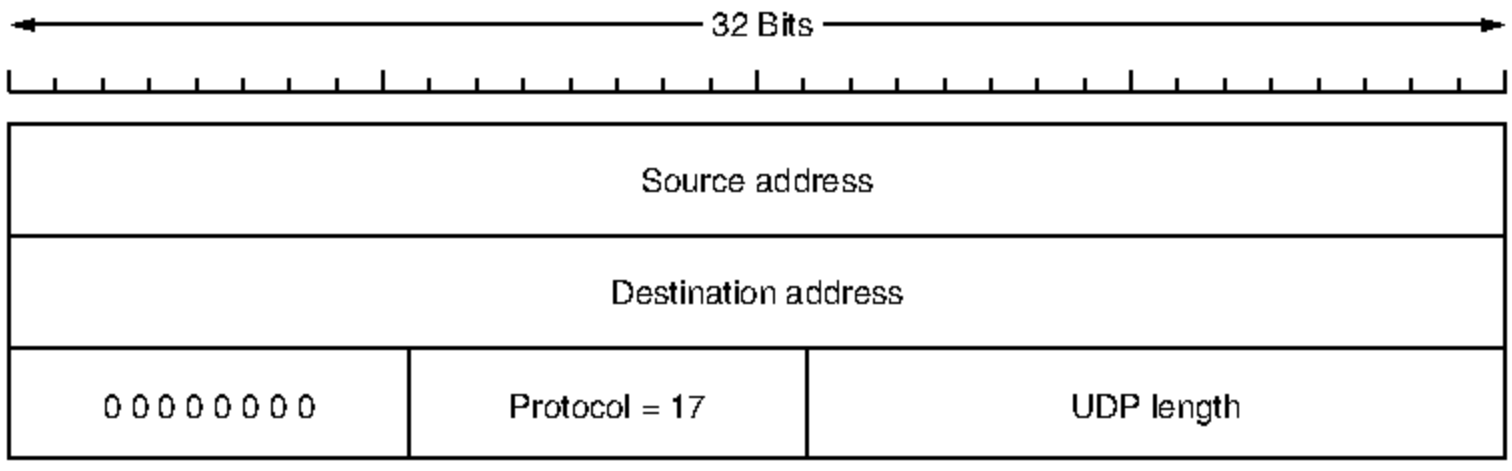


Figure 6-28. The IPv4 pseudoheader included in the UDP checksum.

It is probably worth mentioning explicitly some of the things that UDP does *not* do. It does not do flow control, congestion control, or retransmission upon receipt of a bad segment. All of that is up to the user processes. What it does do is provide an interface to the IP protocol with the added feature of demultiplexing multiple processes using the ports and optional end-to-end error detection. That is all it does.

For applications that need to have precise control over the packet flow, error control, or timing, UDP provides just what the doctor ordered. One area where it is especially useful is in client-server situations. Often, the client sends a short request to the server and expects a short reply back. If either the request or the reply is lost, the client can just time out and try again. Not only is the code simple, but fewer messages are required (one in each direction) than with a protocol requiring an initial setup like TCP.

An application that uses UDP this way is DNS (Domain Name System), which we will study in Chap. 7. In brief, a program that needs to look up the IP address of some host name, for example, *www.cs.berkeley.edu*, can send a UDP packet containing the host name to a DNS server. The server replies with a UDP packet containing the host's IP address. No setup is needed in advance and no release is needed afterward. Just two messages go over the network.

6.4.2 Remote Procedure Call

In a certain sense, sending a message to a remote host and getting a reply back is a lot like making a function call in a programming language. In both cases, you start with one or more parameters and you get back a result. This observation has led people to try to arrange request-reply interactions on networks to be cast in the

form of procedure calls. Such an arrangement makes network applications much easier to program and more familiar to deal with. For example, just imagine a procedure named *get_IP_address*(*host_name*) that works by sending a UDP packet to a DNS server and waiting for the reply, timing out and trying again if one is not forthcoming quickly enough. In this way, all the details of networking can be hidden from the programmer.

The key work in this area was done by Birrell and Nelson (1984). In a nutshell, what Birrell and Nelson suggested was allowing programs to call procedures located on remote hosts. When a process on machine 1 calls a procedure on machine 2, the calling process on 1 is suspended and execution of the called procedure takes place on 2. Information can be transported from the caller to the callee in the parameters and can come back in the procedure result. No message passing is visible to the application programmer. This technique is known as **RPC** (**Remote Procedure Call**) and has become the basis for many networking applications. Traditionally, the calling procedure is known as the client and the called procedure is known as the server, and we will use those names here too.

The idea behind RPC is to make a remote procedure call look as much as possible like a local one. In the simplest form, to call a remote procedure, the client program must be bound with a small library procedure, called the **client stub**, that represents the server procedure in the client's address space. Similarly, the server is bound with a procedure called the **server stub**. These procedures hide the fact that the procedure call from the client to the server is not local.

The actual steps in making an RPC are shown in Fig. 6-29. Step 1 is the client calling the client stub. This call is a local procedure call, with the parameters pushed onto the stack in the normal way. Step 2 is the client stub packing the parameters into a message and making a system call to send the message. Packing the parameters is called **marshaling**. Step 3 is the operating system sending the message from the client machine to the server machine. Step 4 is the operating system passing the incoming packet to the server stub. Finally, step 5 is the server stub calling the server procedure with the unmarshaled parameters. The reply traces the same path in the other direction.

The key item to note here is that the client procedure, written by the user, just makes a normal (i.e., local) procedure call to the client stub, which has the same name as the server procedure. Since the client procedure and client stub are in the same address space, the parameters are passed in the usual way. Similarly, the server procedure is called by a procedure in its address space with the parameters it expects. To the server procedure, nothing is unusual. In this way, instead of I/O being done on sockets, network communication is done by faking a normal procedure call.

Despite the conceptual elegance of RPC, there are a few snakes hiding under the grass. A big one is the use of pointer parameters. Normally, passing a pointer to a procedure is not a problem. The called procedure can use the pointer in the same way the caller can because both procedures live in the same virtual address

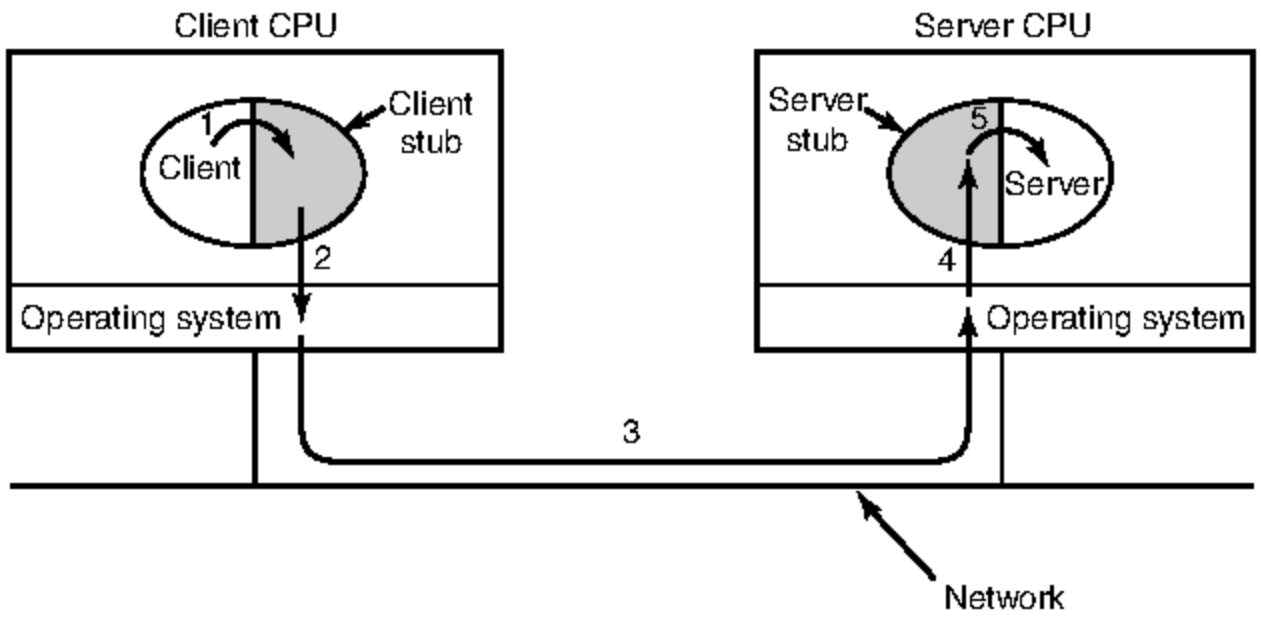


Figure 6-29. Steps in making a remote procedure call. The stubs are shaded.

space. With RPC, passing pointers is impossible because the client and server are in different address spaces.

In some cases, tricks can be used to make it possible to pass pointers. Suppose that the first parameter is a pointer to an integer, *k*. The client stub can marshal *k* and send it along to the server. The server stub then creates a pointer to *k* and passes it to the server procedure, just as it expects. When the server procedure returns control to the server stub, the latter sends *k* back to the client, where the new *k* is copied over the old one, just in case the server changed it. In effect, the standard calling sequence of call-by-reference has been replaced by call-by-copy-restore. Unfortunately, this trick does not always work, for example, if the pointer points to a graph or other complex data structure. For this reason, some restrictions must be placed on parameters to procedures called remotely, as we shall see.

A second problem is that in weakly typed languages, like C, it is perfectly legal to write a procedure that computes the inner product of two vectors (arrays), without specifying how large either one is. Each could be terminated by a special value known only to the calling and called procedures. Under these circumstances, it is essentially impossible for the client stub to marshal the parameters: it has no way of determining how large they are.

A third problem is that it is not always possible to deduce the types of the parameters, not even from a formal specification or the code itself. An example is *printf*, which may have any number of parameters (at least one), and the parameters can be an arbitrary mixture of integers, shorts, longs, characters, strings, floating-point numbers of various lengths, and other types. Trying to call *printf* as a remote procedure would be practically impossible because C is so permissive. However, a rule saying that RPC can be used provided that you do not program in C (or C++) would not be popular with a lot of programmers.

A fourth problem relates to the use of global variables. Normally, the calling and called procedure can communicate by using global variables, in addition to communicating via parameters. But if the called procedure is moved to a remote machine, the code will fail because the global variables are no longer shared.

These problems are not meant to suggest that RPC is hopeless. In fact, it is widely used, but some restrictions are needed to make it work well in practice.

In terms of transport layer protocols, UDP is a good base on which to implement RPC. Both requests and replies may be sent as a single UDP packet in the simplest case and the operation can be fast. However, an implementation must include other machinery as well. Because the request or the reply may be lost, the client must keep a timer to retransmit the request. Note that a reply serves as an implicit acknowledgement for a request, so the request need not be separately acknowledged. Sometimes the parameters or results may be larger than the maximum UDP packet size, in which case some protocol is needed to deliver large messages. If multiple requests and replies can overlap (as in the case of concurrent programming), an identifier is needed to match the request with the reply.

A higher-level concern is that the operation may not be idempotent (i.e., safe to repeat). The simple case is idempotent operations such as DNS requests and replies. The client can safely retransmit these requests again and again if no replies are forthcoming. It does not matter whether the server never received the request, or it was the reply that was lost. The answer, when it finally arrives, will be the same (assuming the DNS database is not updated in the meantime). However, not all operations are idempotent, for example, because they have important side-effects such as incrementing a counter. RPC for these operations requires stronger semantics so that when the programmer calls a procedure it is not executed multiple times. In this case, it may be necessary to set up a TCP connection and send the request over it rather than using UDP.

6.4.3 Real-Time Transport Protocols

Client-server RPC is one area in which UDP is widely used. Another one is for real-time multimedia applications. In particular, as Internet radio, Internet telephony, music-on-demand, videoconferencing, video-on-demand, and other multimedia applications became more commonplace, people have discovered that each application was reinventing more or less the same real-time transport protocol. It gradually became clear that having a generic real-time transport protocol for multiple applications would be a good idea.

Thus was **RTP** (**Real-time Transport Protocol**) born. It is described in RFC 3550 and is now in widespread use for multimedia applications. We will describe two aspects of real-time transport. The first is the RTP protocol for transporting audio and video data in packets. The second is the processing that takes place, mostly at the receiver, to play out the audio and video at the right time. These functions fit into the protocol stack as shown in Fig. 6-30.

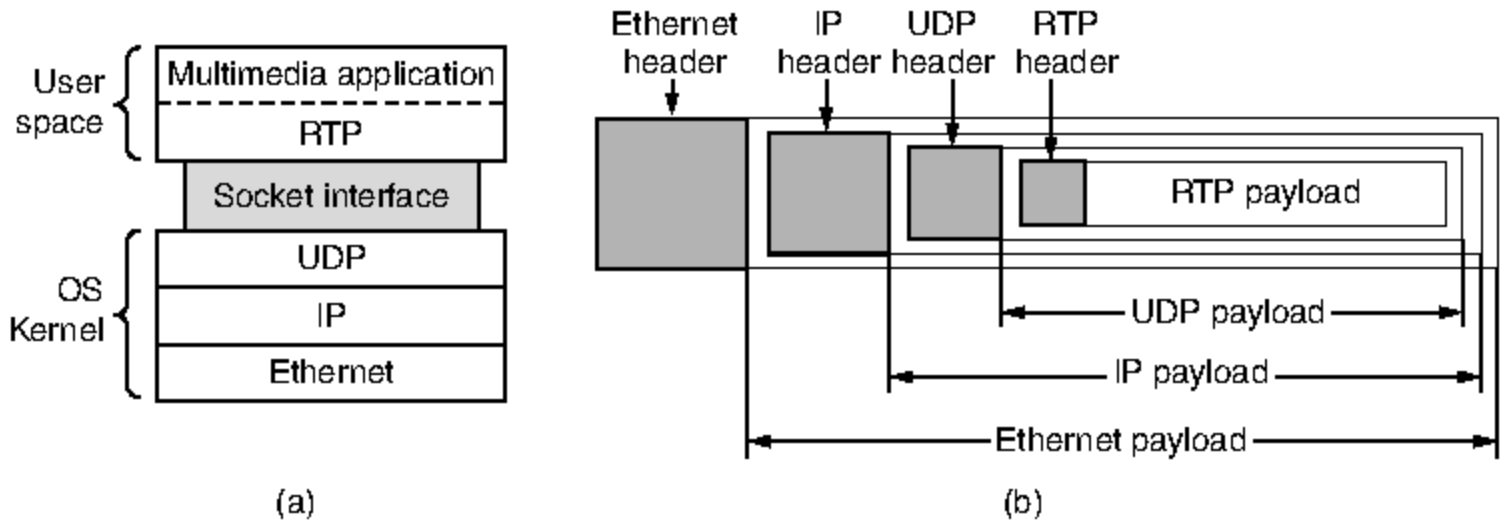


Figure 6-30. (a) The position of RTP in the protocol stack. (b) Packet nesting.

RTP normally runs in user space over UDP (in the operating system). It operates as follows. The multimedia application consists of multiple audio, video, text, and possibly other streams. These are fed into the RTP library, which is in user space along with the application. This library multiplexes the streams and encodes them in RTP packets, which it stuffs into a socket. On the operating system side of the socket, UDP packets are generated to wrap the RTP packets and handed to IP for transmission over a link such as Ethernet. The reverse process happens at the receiver. The multimedia application eventually receives multimedia data from the RTP library. It is responsible for playing out the media. The protocol stack for this situation is shown in Fig. 6-30(a). The packet nesting is shown in Fig. 6-30(b).

As a consequence of this design, it is a little hard to say which layer RTP is in. Since it runs in user space and is linked to the application program, it certainly looks like an application protocol. On the other hand, it is a generic, application-independent protocol that just provides transport facilities, so it also looks like a transport protocol. Probably the best description is that it is a transport protocol that just happens to be implemented in the application layer, which is why we are covering it in this chapter.

RTP—The Real-time Transport Protocol

The basic function of RTP is to multiplex several real-time data streams onto a single stream of UDP packets. The UDP stream can be sent to a single destination (unicasting) or to multiple destinations (multicasting). Because RTP just uses normal UDP, its packets are not treated specially by the routers unless some normal IP quality-of-service features are enabled. In particular, there are no special guarantees about delivery, and packets may be lost, delayed, corrupted, etc.

The RTP format contains several features to help receivers work with multimedia information. Each packet sent in an RTP stream is given a number one

higher than its predecessor. This numbering allows the destination to determine if any packets are missing. If a packet is missing, the best action for the destination to take is up to the application. It may be to skip a video frame if the packets are carrying video data, or to approximate the missing value by interpolation if the packets are carrying audio data. Retransmission is not a practical option since the retransmitted packet would probably arrive too late to be useful. As a consequence, RTP has no acknowledgements, and no mechanism to request retransmissions.

Each RTP payload may contain multiple samples, and they may be coded any way that the application wants. To allow for interworking, RTP defines several profiles (e.g., a single audio stream), and for each profile, multiple encoding formats may be allowed. For example, a single audio stream may be encoded as 8-bit PCM samples at 8 kHz using delta encoding, predictive encoding, GSM encoding, MP3 encoding, and so on. RTP provides a header field in which the source can specify the encoding but is otherwise not involved in how encoding is done.

Another facility many real-time applications need is timestamping. The idea here is to allow the source to associate a timestamp with the first sample in each packet. The timestamps are relative to the start of the stream, so only the differences between timestamps are significant. The absolute values have no meaning. As we will describe shortly, this mechanism allows the destination to do a small amount of buffering and play each sample the right number of milliseconds after the start of the stream, independently of when the packet containing the sample arrived.

Not only does timestamping reduce the effects of variation in network delay, but it also allows multiple streams to be synchronized with each other. For example, a digital television program might have a video stream and two audio streams. The two audio streams could be for stereo broadcasts or for handling films with an original language soundtrack and a soundtrack dubbed into the local language, giving the viewer a choice. Each stream comes from a different physical device, but if they are timestamped from a single counter, they can be played back synchronously, even if the streams are transmitted and/or received somewhat erratically.

The RTP header is illustrated in Fig. 6-31. It consists of three 32-bit words and potentially some extensions. The first word contains the *Version* field, which is already at 2. Let us hope this version is very close to the ultimate version since there is only one code point left (although 3 could be defined as meaning that the real version was in an extension word).

The *P* bit indicates that the packet has been padded to a multiple of 4 bytes. The last padding byte tells how many bytes were added. The *X* bit indicates that an extension header is present. The format and meaning of the extension header are not defined. The only thing that is defined is that the first word of the extension gives the length. This is an escape hatch for any unforeseen requirements.

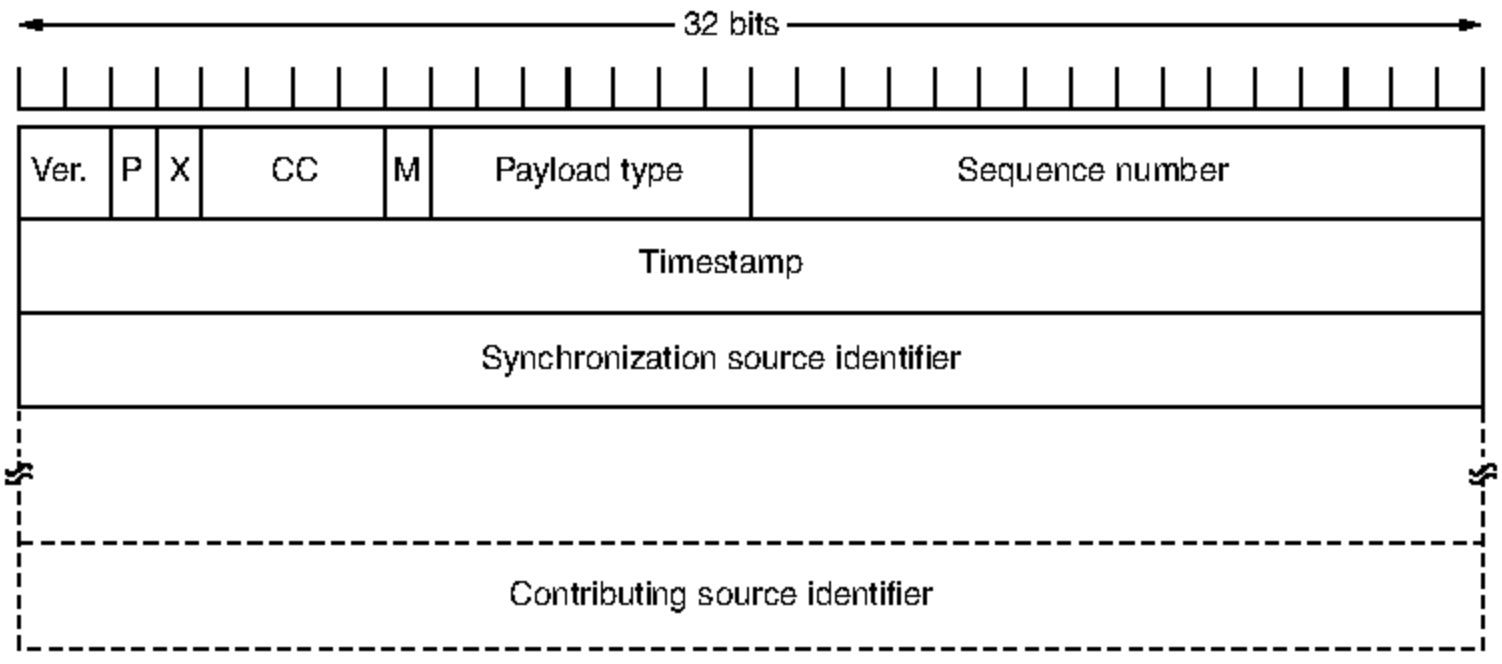


Figure 6-31. The RTP header.

The *CC* field tells how many contributing sources are present, from 0 to 15 (see below). The *M* bit is an application-specific marker bit. It can be used to mark the start of a video frame, the start of a word in an audio channel, or something else that the application understands. The *Payload type* field tells which encoding algorithm has been used (e.g., uncompressed 8-bit audio, MP3, etc.). Since every packet carries this field, the encoding can change during transmission. The *Sequence number* is just a counter that is incremented on each RTP packet sent. It is used to detect lost packets.

The *Timestamp* is produced by the stream's source to note when the first sample in the packet was made. This value can help reduce timing variability called jitter at the receiver by decoupling the playback from the packet arrival time. The *Synchronization source identifier* tells which stream the packet belongs to. It is the method used to multiplex and demultiplex multiple data streams onto a single stream of UDP packets. Finally, the *Contributing source identifiers*, if any, are used when mixers are present in the studio. In that case, the mixer is the synchronizing source, and the streams being mixed are listed here.

RTCP—The Real-time Transport Control Protocol

RTP has a little sister protocol (little sibling protocol?) called **RTCP** (**Real-time Transport Control Protocol**). It is defined along with RTP in RFC 3550 and handles feedback, synchronization, and the user interface. It does not transport any media samples.

The first function can be used to provide feedback on delay, variation in delay or jitter, bandwidth, congestion, and other network properties to the sources. This information can be used by the encoding process to increase the data rate (and give better quality) when the network is functioning well and to cut back the data

rate when there is trouble in the network. By providing continuous feedback, the encoding algorithms can be continuously adapted to provide the best quality possible under the current circumstances. For example, if the bandwidth increases or decreases during the transmission, the encoding may switch from MP3 to 8-bit PCM to delta encoding as required. The *Payload type* field is used to tell the destination what encoding algorithm is used for the current packet, making it possible to vary it on demand.

An issue with providing feedback is that the RTCP reports are sent to all participants. For a multicast application with a large group, the bandwidth used by RTCP would quickly grow large. To prevent this from happening, RTCP senders scale down the rate of their reports to collectively consume no more than, say, 5% of the media bandwidth. To do this, each participant needs to know the media bandwidth, which it learns from the sender, and the number of participants, which it estimates by listening to other RTCP reports.

RTCP also handles interstream synchronization. The problem is that different streams may use different clocks, with different granularities and different drift rates. RTCP can be used to keep them in sync.

Finally, RTCP provides a way for naming the various sources (e.g., in ASCII text). This information can be displayed on the receiver's screen to indicate who is talking at the moment.

More information about RTP can be found in Perkins (2003).

Playout with Buffering and Jitter Control

Once the media information reaches the receiver, it must be played out at the right time. In general, this will not be the time at which the RTP packet arrived at the receiver because packets will take slightly different amounts of time to transit the network. Even if the packets are injected with exactly the right intervals between them at the sender, they will reach the receiver with different relative times. This variation in delay is called **jitter**. Even a small amount of packet jitter can cause distracting media artifacts, such as jerky video frames and unintelligible audio, if the media is simply played out as it arrives.

The solution to this problem is to **buffer** packets at the receiver before they are played out to reduce the jitter. As an example, in Fig. 6-32 we see a stream of packets being delivered with a substantial amount of jitter. Packet 1 is sent from the server at $t = 0$ sec and arrives at the client at $t = 1$ sec. Packet 2 undergoes more delay and takes 2 sec to arrive. As the packets arrive, they are buffered on the client machine.

At $t = 10$ sec, playback begins. At this time, packets 1 through 6 have been buffered so that they can be removed from the buffer at uniform intervals for smooth play. In the general case, it is not necessary to use uniform intervals because the RTP timestamps tell when the media should be played.

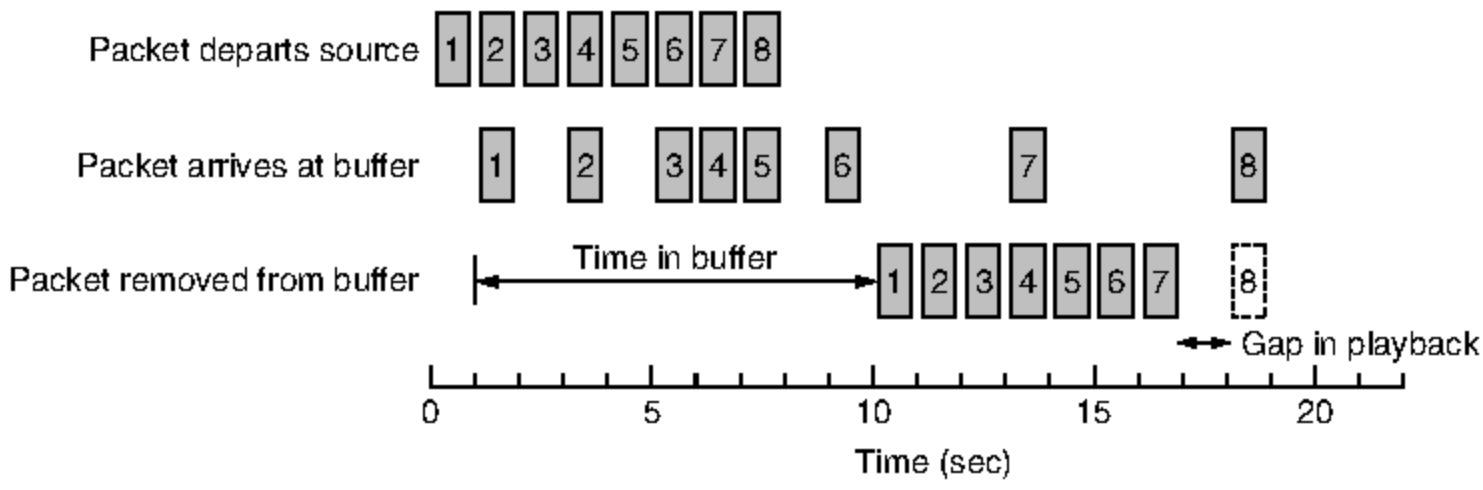


Figure 6-32. Smoothing the output stream by buffering packets.

Unfortunately, we can see that packet 8 has been delayed so much that it is not available when its play slot comes up. There are two options. Packet 8 can be skipped and the player can move on to subsequent packets. Alternatively, playback can stop until packet 8 arrives, creating an annoying gap in the music or movie. In a live media application like a voice-over-IP call, the packet will typically be skipped. Live applications do not work well on hold. In a streaming media application, the player might pause. This problem can be alleviated by delaying the starting time even more, by using a larger buffer. For a streaming audio or video player, buffers of about 10 seconds are often used to ensure that the player receives all of the packets (that are not dropped in the network) in time. For live applications like videoconferencing, short buffers are needed for responsiveness.

A key consideration for smooth playout is the **playback point**, or how long to wait at the receiver for media before playing it out. Deciding how long to wait depends on the jitter. The difference between a low-jitter and high-jitter connection is shown in Fig. 6-33. The average delay may not differ greatly between the two, but if there is high jitter the playback point may need to be much further out to capture 99% of the packets than if there is low jitter.

To pick a good playback point, the application can measure the jitter by looking at the difference between the RTP timestamps and the arrival time. Each difference gives a sample of the delay (plus an arbitrary, fixed offset). However, the delay can change over time due to other, competing traffic and changing routes. To accommodate this change, applications can adapt their playback point while they are running. However, if not done well, changing the playback point can produce an observable glitch to the user. One way to avoid this problem for audio is to adapt the playback point between **talkspurts**, in the gaps in a conversation. No one will notice the difference between a short and slightly longer silence. RTP lets applications set the *M* marker bit to indicate the start of a new talkspurt for this purpose.

If the absolute delay until media is played out is too long, live applications will suffer. Nothing can be done to reduce the propagation delay if a direct path is

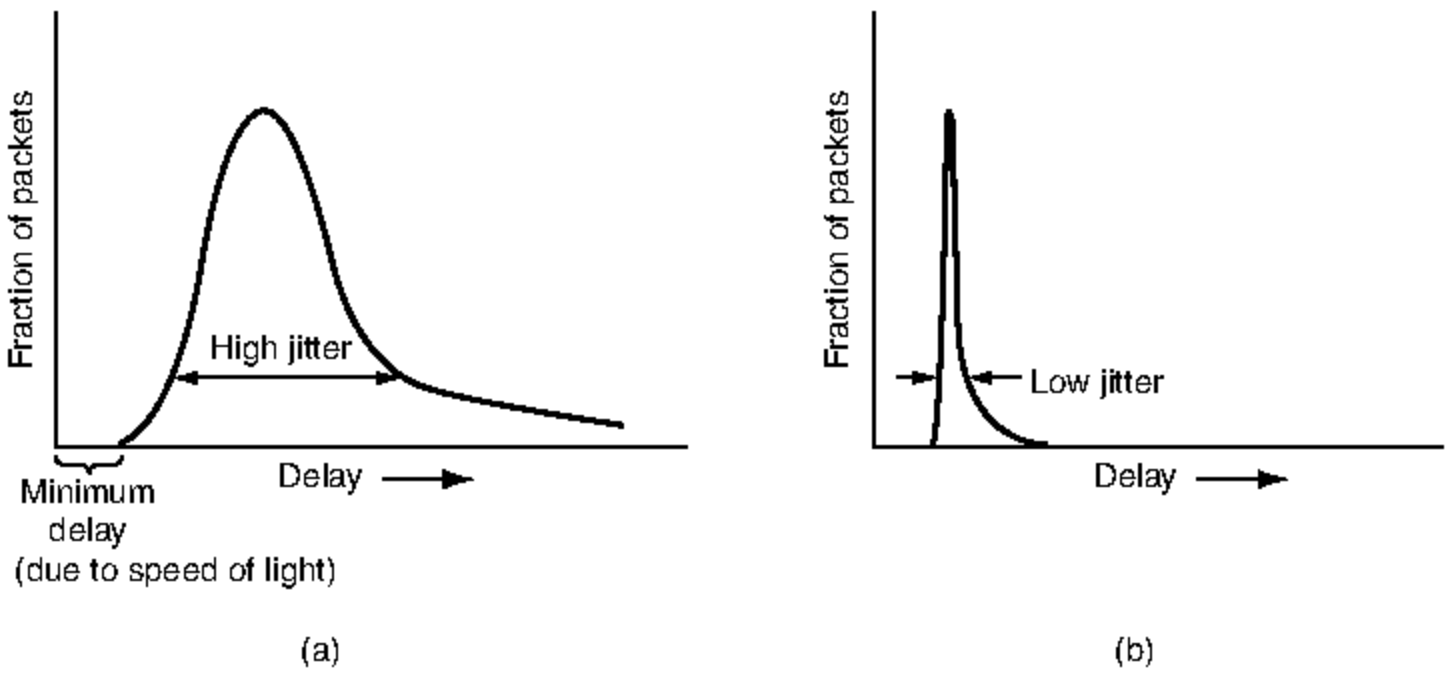


Figure 6-33. (a) High jitter. (b) Low jitter.

already being used. The playback point can be pulled in by simply accepting that a larger fraction of packets will arrive too late to be played. If this is not acceptable, the only way to pull in the playback point is to reduce the jitter by using a better quality of service, for example, the expedited forwarding differentiated service. That is, a better network is needed.

6.5 THE INTERNET TRANSPORT PROTOCOLS: TCP

UDP is a simple protocol and it has some very important uses, such as client-server interactions and multimedia, but for most Internet applications, reliable, sequenced delivery is needed. UDP cannot provide this, so another protocol is required. It is called TCP and is the main workhorse of the Internet. Let us now study it in detail.

6.5.1 Introduction to TCP

TCP (**Transmission Control Protocol**) was specifically designed to provide a reliable end-to-end byte stream over an unreliable internetwork. An 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 in September 1981. As time went on, many improvements have been made, and various errors and inconsistencies have been fixed. To give you a sense of the extent of TCP, the important RFCs are

now RFC 793 plus: clarifications and bug fixes in RFC 1122; extensions for high-performance in RFC 1323; selective acknowledgements in RFC 2018; congestion control in RFC 2581; repurposing of header fields for quality of service in RFC 2873; improved retransmission timers in RFC 2988; and explicit congestion notification in RFC 3168. The full collection is even larger, which led to a guide to the many RFCs, published of course as another RFC document, RFC 4614.

Each machine supporting TCP has a TCP transport entity, either a library procedure, a user process, or most commonly part of the kernel. In all cases, it 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 64 KB (in practice, often 1460 data bytes in order to fit in a single Ethernet frame with the IP and TCP headers), and sends each piece as a separate IP datagram. When 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, nor any indication of how fast datagrams may be sent. It is up to TCP to send datagrams fast enough to make use of the capacity but not cause congestion, and to time out and retransmit any datagrams that are not delivered. 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 good performance with the reliability that most applications want and that IP does not provide.

6.5.2 The TCP Service Model

TCP service is obtained by both the sender and the receiver creating 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. For TCP service to be obtained, a connection must be explicitly established between a socket on one machine and a socket on another machine. The socket calls are listed in Fig. 6-5.

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 1024 are reserved for standard services that can usually only be started by privileged users (e.g., root in UNIX systems). They are called **well-known ports**. For example, any process wishing to remotely retrieve mail from a host can connect to the destination host's port 143 to contact its IMAP

daemon. The list of well-known ports is given at *www.iana.org*. Over 700 have been assigned. A few of the better-known ones are listed in Fig. 6-34.

Port	Protocol	Use
20, 21	FTP	File transfer
22	SSH	Remote login, replacement for Telnet
25	SMTP	Email
80	HTTP	World Wide Web
110	POP-3	Remote email access
143	IMAP	Remote email access
443	HTTPS	Secure Web (HTTP over SSL/TLS)
543	RTSP	Media player control
631	IPP	Printer sharing

Figure 6-34. Some assigned ports.

Other ports from 1024 through 49151 can be registered with IANA for use by unprivileged users, but applications can and do choose their own ports. For example, the BitTorrent peer-to-peer file-sharing application (unofficially) uses ports 6881–6887, but may run on other ports as well.

It would certainly be possible to have the FTP daemon attach itself to port 21 at boot time, the SSH daemon attach itself to port 22 at boot time, and so on. However, doing so would clutter up memory with daemons that were idle most of the time. Instead, what is commonly done is to have a single daemon, called **inetd** (**Internet daemon**) in UNIX, attach itself to multiple ports and wait for the first incoming connection. When that occurs, *inetd* forks off a new process and executes the appropriate daemon in it, letting that daemon handle the request. In this way, the daemons other than *inetd* are only active when there is work for them to do. Inetd learns which ports it is to use from a configuration file. Consequently, the system administrator can set up the system to have permanent daemons on the busiest ports (e.g., port 80) and *inetd* on the rest.

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. 6-35), or some other way. There is no way for the receiver to detect the unit(s) in which the data were written, no matter how hard it tries.

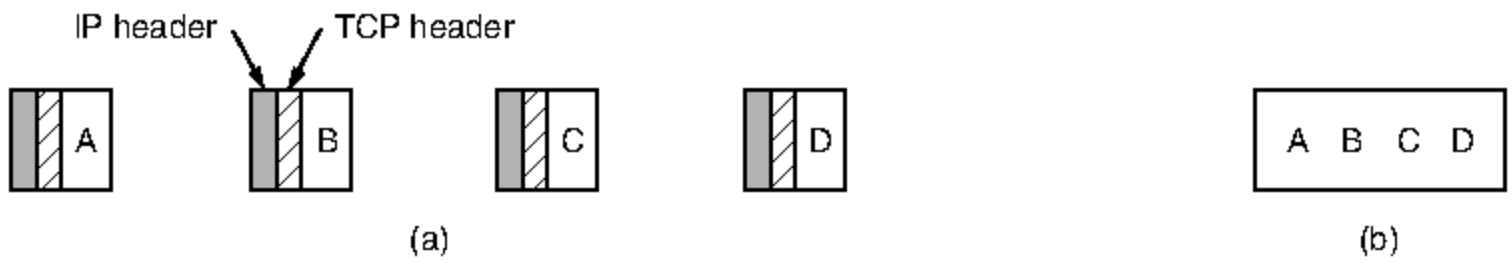


Figure 6-35. (a) Four 512-byte segments sent as separate IP datagrams. (b) The 2048 bytes of data delivered to the application in a single READ call.

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. However, sometimes the application really wants the data to be sent immediately. For example, suppose a user of an interactive game wants to send a stream of updates. It is essential that the updates be sent immediately, not buffered until there is a collection of them. To force data out, TCP has the notion of a PUSH flag that is carried on packets. The original intent was to let applications tell TCP implementations via the PUSH flag not to delay the transmission. However, applications cannot literally set the PUSH flag when they send data. Instead, different operating systems have evolved different options to expedite transmission (e.g., TCP_NODELAY in Windows and Linux).

For Internet archaeologists, we will also mention one interesting feature of TCP service that remains in the protocol but is rarely used: **urgent data**. When an application has high priority data that should be processed immediately, for example, if an interactive user hits the CTRL-C key to break off a remote computation that has already begun, the sending application can put some control information in the data stream and give 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 provides a crude signaling mechanism and leaves everything else up to the application. However, while urgent data is potentially useful, it found no compelling application early on and fell into disuse. Its use is now discouraged because of implementation differences, leaving applications to handle their own signaling. Perhaps future transport protocols will provide better signaling.

6.5.3 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.

A key feature of TCP, and one that dominates the protocol design, is that every byte on a TCP connection has its own 32-bit sequence number. When the Internet began, the lines between routers were mostly 56-kbps leased lines, so a host blasting away at full speed took over 1 week to cycle through the sequence numbers. At modern network speeds, the sequence numbers can be consumed at an alarming rate, as we will see later. Separate 32-bit sequence numbers are carried on packets for the sliding window position in one direction and for acknowledgements in the reverse direction, as discussed below.

The sending and receiving TCP entities exchange data in the form of segments. A **TCP 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 can 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 65,515-byte IP payload. Second, each link has an **MTU** (**Maximum Transfer Unit**). Each segment must fit in the MTU at the sender and receiver so that it can be sent and received in a single, unfragmented packet. In practice, the MTU is generally 1500 bytes (the Ethernet payload size) and thus defines the upper bound on segment size.

However, it is still possible for IP packets carrying TCP segments to be fragmented when passing over a network path for which some link has a small MTU. If this happens, it degrades performance and causes other problems (Kent and Mogul, 1987). Instead, modern TCP implementations perform **path MTU discovery** by using the technique outlined in RFC 1191 that we described in Sec. 5.5.5. This technique uses ICMP error messages to find the smallest MTU for any link on the path. TCP then adjusts the segment size downwards to avoid fragmentation.

The basic protocol used by TCP entities is the sliding window protocol with a dynamic window size. 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 exist, and otherwise without) bearing an acknowledgement number equal to the next sequence number it expects to receive and the remaining window size. 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 sometimes subtle ins and outs, which we will cover below. Segments can 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. The retransmissions may include different byte

ranges than the original transmission, requiring careful administration to keep track of which bytes have been correctly received so far. However, since each byte in the stream has its own unique offset, it can be done.

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.

6.5.4 The TCP Segment Header

Figure 6-36 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 $65,535 - 20 - 20 = 65,495$ data bytes may follow, where the first 20 refer 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.

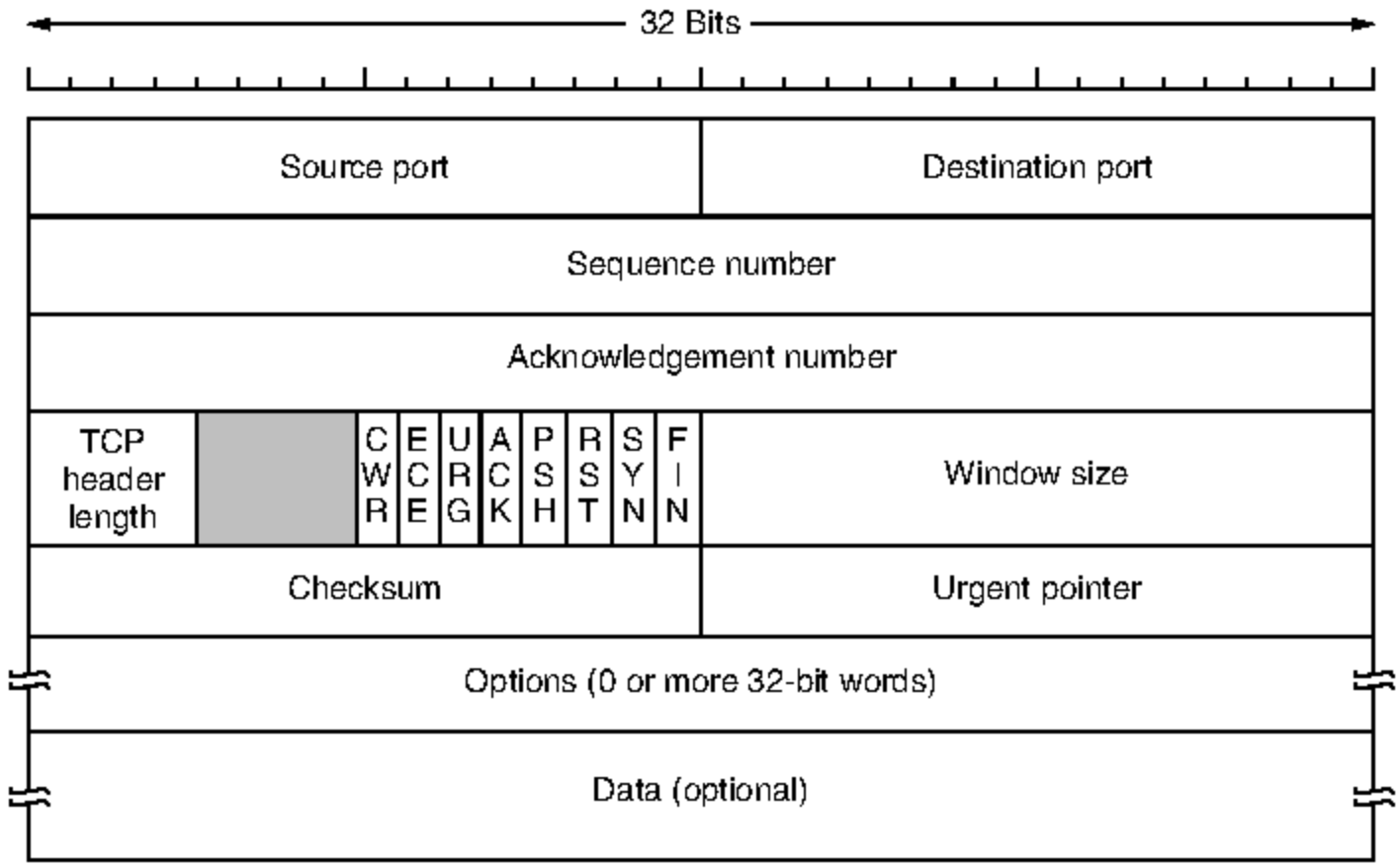


Figure 6-36. The TCP header.

Let us dissect the TCP header field by field. The *Source port* and *Destination port* fields identify the local end points of the connection. A TCP port plus its host's IP address forms a 48-bit unique end point. The source and destination end points together identify the connection. This connection identifier is called a **5 tuple** because it consists of five pieces of information: the protocol (TCP), source IP and source port, and destination IP and destination port.

The *Sequence number* and *Acknowledgement number* fields perform their usual functions. Note that the latter specifies the next in-order byte expected, not the last byte correctly received. It is a **cumulative acknowledgement** because it summarizes the received data with a single number. It does not go beyond lost data. Both are 32 bits 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 4-bit field that is not used. The fact that these bits have remained unused for 30 years (as only 2 of the original reserved 6 bits have been reclaimed) is testimony to how well thought out TCP is. Lesser protocols would have needed these bits to fix bugs in the original design.

Now come eight 1-bit flags. *CWR* and *ECE* are used to signal congestion when ECN (Explicit Congestion Notification) is used, as specified in RFC 3168. *ECE* is set to signal an *ECN-Echo* to a TCP sender to tell it to slow down when the TCP receiver gets a congestion indication from the network. *CWR* is set to signal *Congestion Window Reduced* from the TCP sender to the TCP receiver so that it knows the sender has slowed down and can stop sending the *ECN-Echo*. We discuss the role of ECN in TCP congestion control in Sec. 6.5.10.

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, but it is seldom used.

The *ACK* bit is set to 1 to indicate that the *Acknowledgement number* is valid. This is the case for nearly all packets. 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).

The *RST* bit is used to abruptly 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, however, so it has $SYN = 1$ and $ACK = 1$. In essence, the *SYN* bit is used to denote both 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, the closing 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-sized sliding window. The *Window size* field tells how many bytes may be sent starting at the byte acknowledged. A *Window size* field of 0 is legal and says that the bytes up to and including *Acknowledgement number* − 1 have been received, but that the receiver has not had a chance to consume the data and would like no more data for the moment, thank you. The receiver can later grant permission to send by transmitting a segment with the same *Acknowledgement number* and a nonzero *Window size* field.

In the protocols of Chap. 3, acknowledgements of frames received and permission to send new frames were tied together. This was a consequence of a fixed window size for each protocol. In TCP, acknowledgements and permission to send additional data are completely decoupled. In effect, a receiver can say: "I have received bytes up through k but I do not want any more just now, thank you." This decoupling (in fact, a variable-sized window) gives additional flexibility. We will study it in detail below.

A *Checksum* is also provided for extra reliability. It checksums the header, the data, and a conceptual pseudoheader in exactly the same way as UDP, except that the pseudoheader has the protocol number for TCP (6) and the checksum is mandatory. Please see Sec. 6.4.1 for details.

The *Options* field provides a way to add extra facilities not covered by the regular header. Many options have been defined and several are commonly used. The options are of variable length, fill a multiple of 32 bits by using padding with zeros, and may extend to 40 bytes to accommodate the longest TCP header that can be specified. Some options are carried when a connection is established to negotiate or inform the other side of capabilities. Other options are carried on packets during the lifetime of the connection. Each option has a Type-Length-Value encoding.

A widely used option is the one that allows each host to specify the **MSS** (**Maximum Segment Size**) it is willing to accept. Using large segments is more efficient than using small ones because the 20-byte header can be amortized over more data, but small hosts may not be able to handle big segments. During connection setup, each side can announce its maximum and see its partner's. 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. The maximum segment size in the two directions need not be the same.

For lines with high bandwidth, high delay, or both, the 64-KB window corresponding to a 16-bit field is a problem. For example, on an OC-12 line (of roughly 600 Mbps), it takes less than 1 msec to output a full 64-KB window. If the round-trip propagation delay is 50 msec (which is typical for a transcontinental

fiber), the sender will be idle more than 98% of the time waiting for acknowledgements. A larger window size would allow the sender to keep pumping data out. The **window scale** option allows the sender and receiver to negotiate a window scale factor at the start of a connection. Both sides use the scale factor to shift the *Window size* field up to 14 bits to the left, thus allowing windows of up to 2^{30} bytes. Most TCP implementations support this option.

The **timestamp** option carries a timestamp sent by the sender and echoed by the receiver. It is included in every packet, once its use is established during connection setup, and used to compute round-trip time samples that are used to estimate when a packet has been lost. It is also used as a logical extension of the 32-bit sequence number. On a fast connection, the sequence number may wrap around quickly, leading to possible confusion between old and new data. The **PAWS** (**Protection Against Wrapped Sequence numbers**) scheme discards arriving segments with old timestamps to prevent this problem.

Finally, the **SACK** (**Selective ACKnowledgement**) option lets a receiver tell a sender the ranges of sequence numbers that it has received. It supplements the *Acknowledgement number* and is used after a packet has been lost but subsequent (or duplicate) data has arrived. The new data is not reflected by the *Acknowledgement number* field in the header because that field gives only the next in-order byte that is expected. With SACK, the sender is explicitly aware of what data the receiver has and hence can determine what data should be retransmitted. SACK is defined in RFC 2108 and RFC 2883 and is increasingly used. We describe the use of SACK along with congestion control in Sec. 6.5.10.

6.5.5 TCP Connection Establishment

Connections are established in TCP by means of 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 in that order, 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.

If some process is listening to the port, that process is given the incoming TCP segment. It can 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. 6-37(a). Note that a *SYN* segment consumes 1 byte of sequence space so that it can be acknowledged unambiguously.

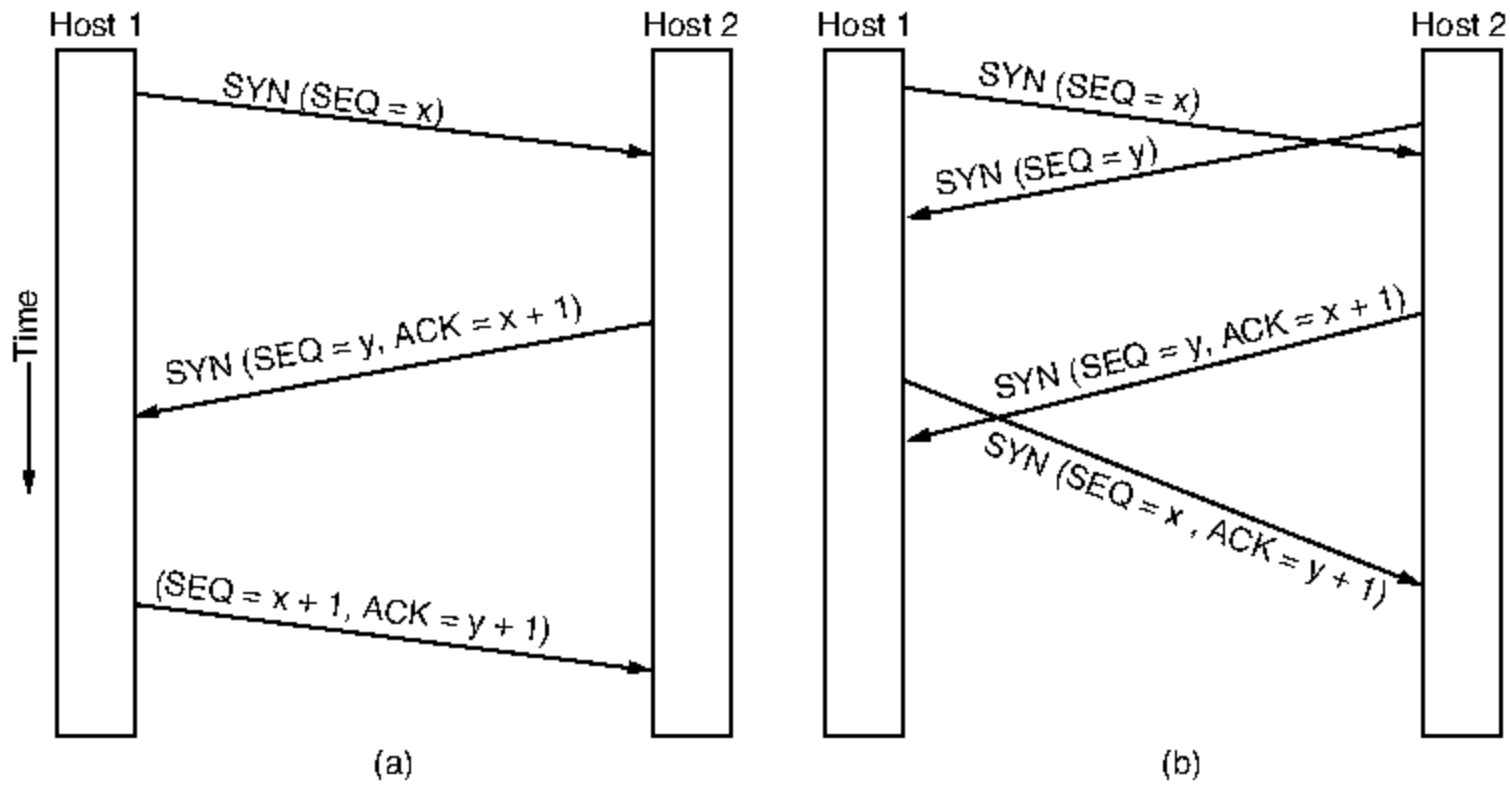


Figure 6-37. (a) TCP connection establishment in the normal case. (b) Simultaneous connection establishment on both sides.

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. 6-37(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).

Recall that the initial sequence number chosen by each host should cycle slowly, rather than be a constant such as 0. This rule is to protect against delayed duplicate packets, as we discussed in Sec 6.2.2. Originally this was accomplished with a clock-based scheme in which the clock ticked every 4 μsec.

However, a vulnerability with implementing the three-way handshake is that the listening process must remember its sequence number as soon it responds with its own *SYN* segment. This means that a malicious sender can tie up resources on a host by sending a stream of *SYN* segments and never following through to complete the connection. This attack is called a **SYN flood**, and it crippled many Web servers in the 1990s.

One way to defend against this attack is to use **SYN cookies**. Instead of remembering the sequence number, a host chooses a cryptographically generated sequence number, puts it on the outgoing segment, and forgets it. If the three-way handshake completes, this sequence number (plus 1) will be returned to the host. It can then regenerate the correct sequence number by running the same cryptographic function, as long as the inputs to that function are known, for example, the other host's IP address and port, and a local secret. This procedure allows the host to check that an acknowledged sequence number is correct without having to

remember the sequence number separately. There are some caveats, such as the inability to handle TCP options, so SYN cookies may be used only when the host is subject to a SYN flood. However, they are an interesting twist on connection establishment. For more information, see RFC 4987 and Lemon (2002).

6.5.6 TCP Connection Release

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 is shut down. There is, in fact, no essential difference between the two hosts releasing sequentially or simultaneously.

To avoid the two-army problem (discussed in Sec. 6.2.3), 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 anymore and will 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.

6.5.7 TCP Connection Management Modeling

The steps required to establish and release connections can be represented in a finite state machine with the 11 states listed in Fig. 6-38. 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. 6-39. The common case of a client actively connecting to a passive server is shown with heavy lines—solid for the client, dotted for the server. The lightface lines are unusual event sequences.

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
TIME 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

Figure 6-38. The states used in the TCP connection management finite state machine.

Each line in Fig. 6-39 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.

One can best understand the diagram by first following the path of a client (the heavy solid line), then later following the path of a server (the heavy dashed line). When an application program 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 shoots off 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 the state *FIN WAIT 2* and one direction of the connection is 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 twice 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.

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*

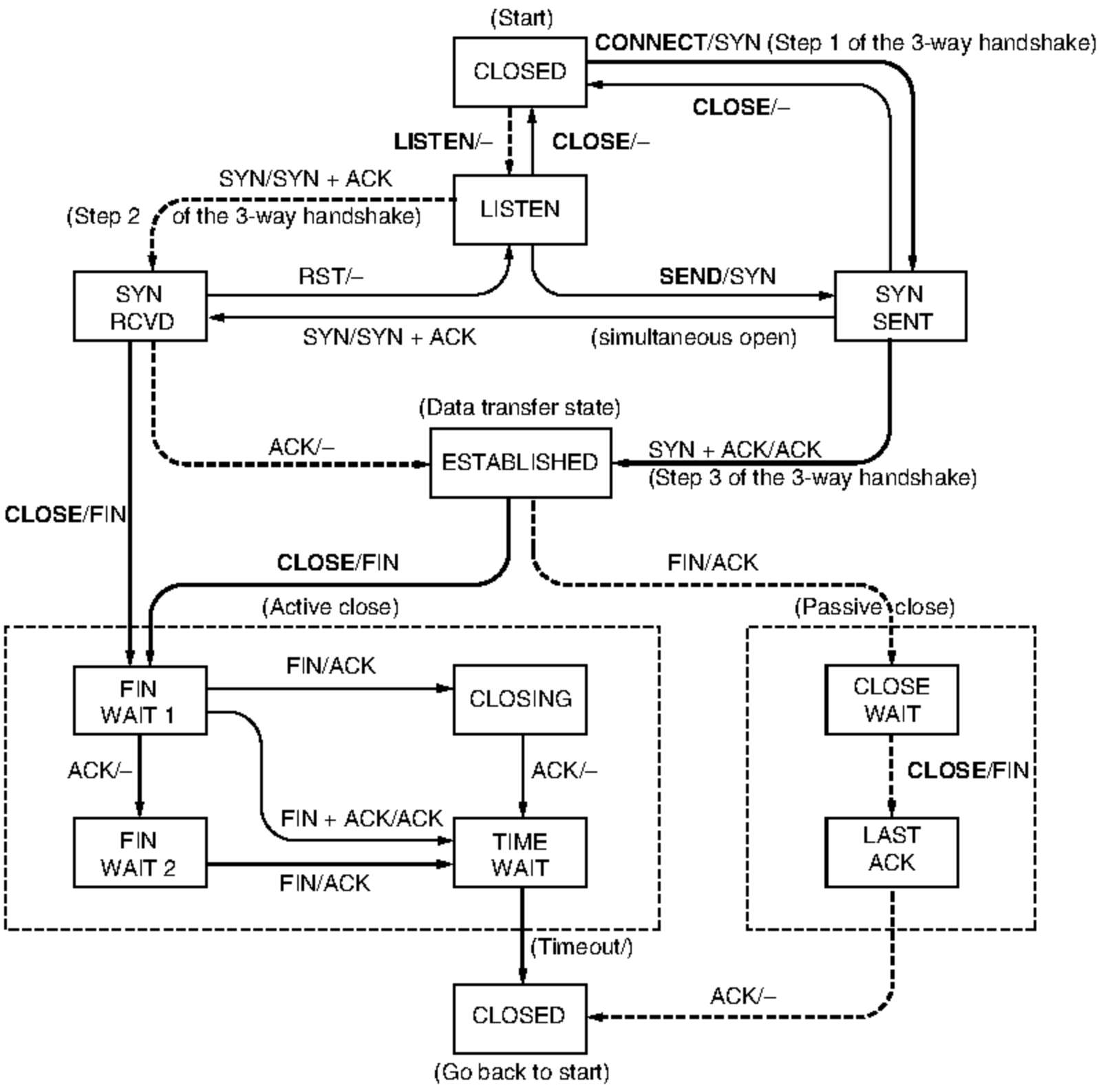


Figure 6-39. TCP connection management finite state machine. The heavy solid line is the normal path for a client. The heavy dashed line is the normal path for a server. The light lines are unusual events. Each transition is labeled with the event causing it and the action resulting from it, separated by a slash.

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 is done transmitting its data, 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.

6.5.8 TCP Sliding Window

As mentioned earlier, window management in TCP decouples the issues of acknowledgement of the correct receipt of segments and receiver buffer allocation. For example, suppose the receiver has a 4096-byte buffer, as shown in Fig. 6-40. 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 bytes 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.

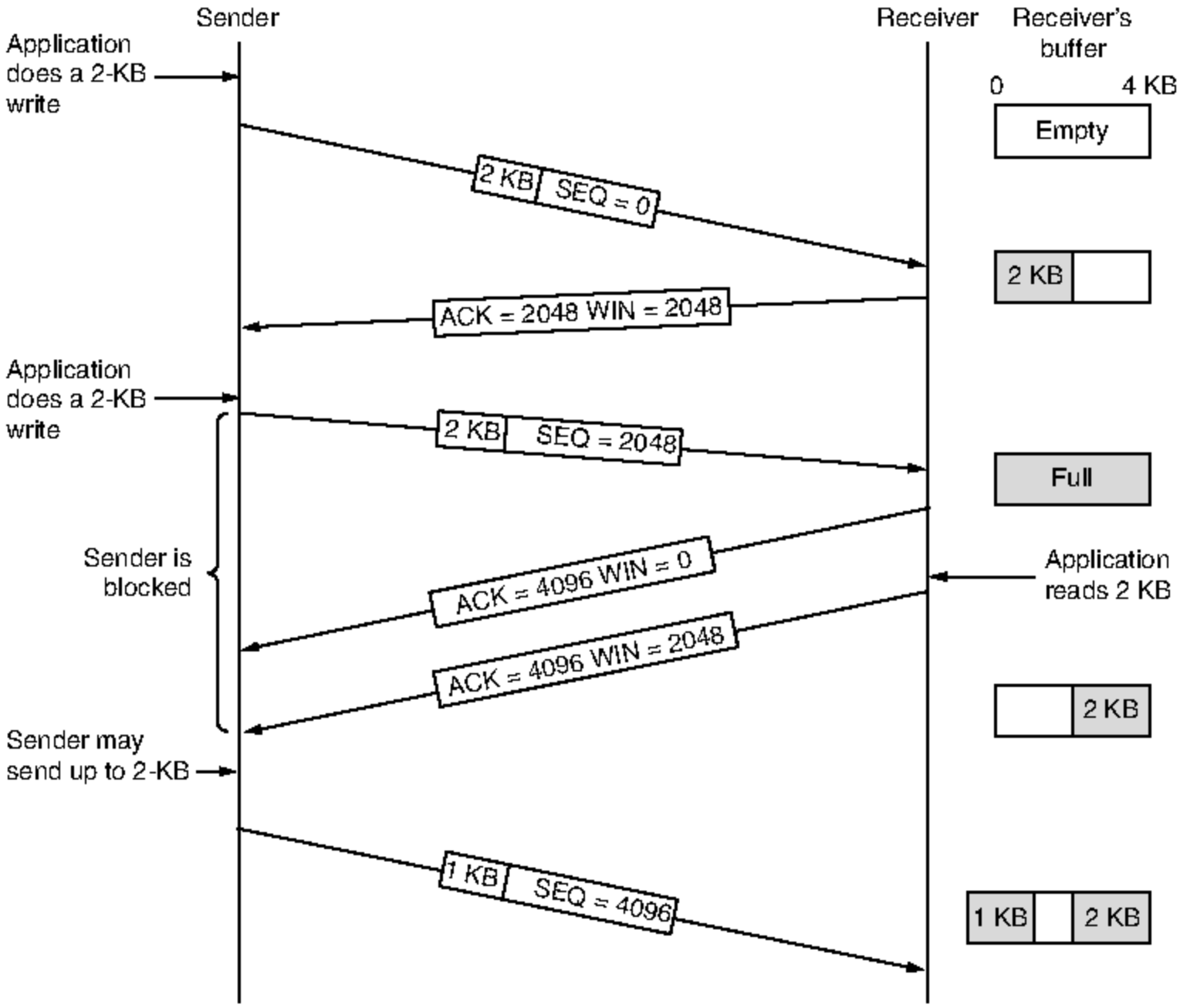


Figure 6-40. Window management in TCP.

Now the sender transmits another 2048 bytes, which are acknowledged, but the advertised window is of size 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 and more data can be sent.

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 force the receiver to reannounce the next byte expected and the window size. This packet is called a **window probe**. The TCP standard explicitly provides this option to prevent deadlock if a window update 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. 6-40, when the first 2 KB of data came in, TCP, knowing that it had a 4-KB window, 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 used to improve performance.

Consider a connection to a remote terminal, for example using SSH or telnet, that reacts on every keystroke. In the worst case, whenever 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 remote terminal 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 remote terminal has processed the character, it echoes the character for local display using 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 called **delayed acknowledgements**. The idea is to delay acknowledgements and window updates for up to 500 msec in the hope of acquiring some data on which to hitch a free ride. Assuming the terminal echoes within 500 msec, only one 41-byte packet now need be sent back by the remote side, cutting the packet count and bandwidth usage in half.

Although delayed acknowledgements reduce the load placed on the network by the receiver, a sender that sends multiple short packets (e.g., 41-byte packets containing 1 byte of data) is still operating inefficiently. 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 in small pieces, just send the first piece and buffer all the rest until the first piece is acknowledged. Then send all the buffered data in one TCP segment and start buffering again until the next segment is acknowledged. That is, only one short packet can be outstanding at any time. If many pieces of data are sent by the application in one round-trip time, Nagle's algorithm will put the many pieces in one segment, greatly reducing the bandwidth used. The algorithm additionally says that a new segment should be sent if enough data have trickled in to fill 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, in interactive games that are run over the Internet, the players typically want a rapid stream of short update packets. Gathering the updates to send them in bursts makes the game respond erratically, which makes for unhappy users. A more subtle problem is that Nagle's algorithm can sometimes interact with delayed acknowledgements to cause a temporary deadlock: the receiver waits for data on which to piggyback an acknowledgement, and the sender waits on the acknowledgement to send more data. This interaction can delay the downloads of Web pages. Because of these problems, Nagle's algorithm can be disabled (which is called the *TCP_NODELAY* option). Mogul and Minshall (2001) discuss this and other solutions.

Another problem that can degrade 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 only 1 byte at a time. To see the problem, look at Fig. 6-41. Initially, the TCP buffer on the receiving side is full (i.e., it has a window of size 0) and the sender knows this. 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 and sets the window to 0. This behavior can go on forever.

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 until its buffer is half empty, whichever is smaller. Furthermore, the sender can also help by not sending tiny segments. Instead, it should wait until it can send a full segment, or at least one containing half of the receiver's buffer size.

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 can also buffer data, so it can block a READ request from the application until it has a large chunk of data for it. Doing so reduces the number of calls to TCP (and the overhead). It also increases the response time, but for noninteractive applications like file transfer, efficiency may be more important than response time to individual requests.

Another issue that the receiver must handle is that segments may arrive out of order. The receiver will buffer the data until it can be passed up to the application

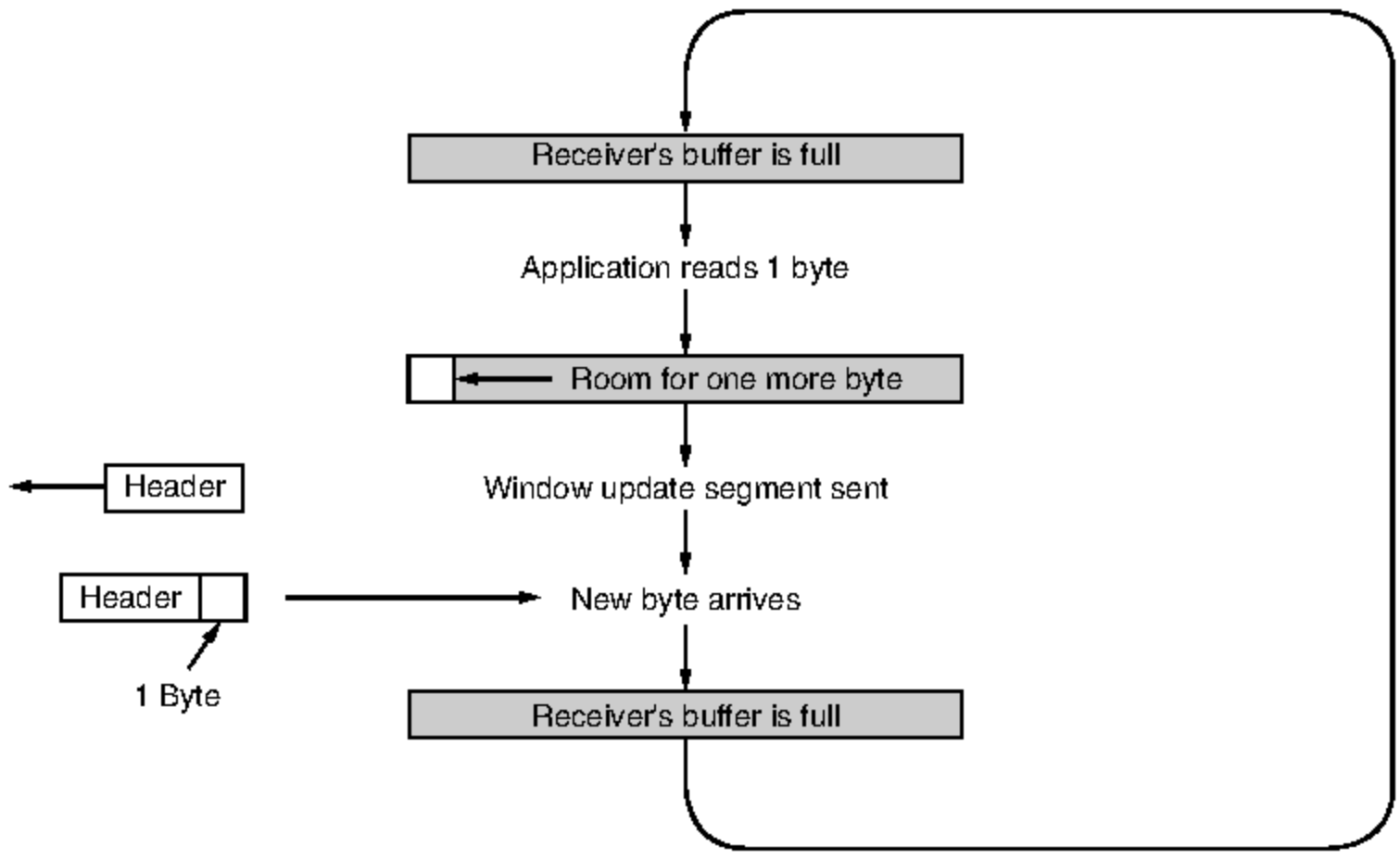


Figure 6-41. Silly window syndrome.

in order. Actually, nothing bad would happen if out-of-order segments were discarded, since they would eventually be retransmitted by the sender, but it would be wasteful.

Acknowledgements can be sent only when all the data up to the byte acknowledged have been received. This is called a **cumulative acknowledgement**. 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. As 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.

6.5.9 TCP Timer Management

TCP uses multiple timers (at least conceptually) to do its work. The most important of these is the **RTO** (**Retransmission TimeOut**). 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 os started again). The question that arises is: how long should the timeout be?

This problem is much more difficult in the transport layer than in data link protocols such as 802.11. In the latter case, the expected delay is measured in

microseconds and 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. 6-42(a). Since acknowledgements are rarely delayed in the data link layer (due to lack of congestion), the absence of an acknowledgement at the expected time generally means either the frame or the acknowledgement has been lost.

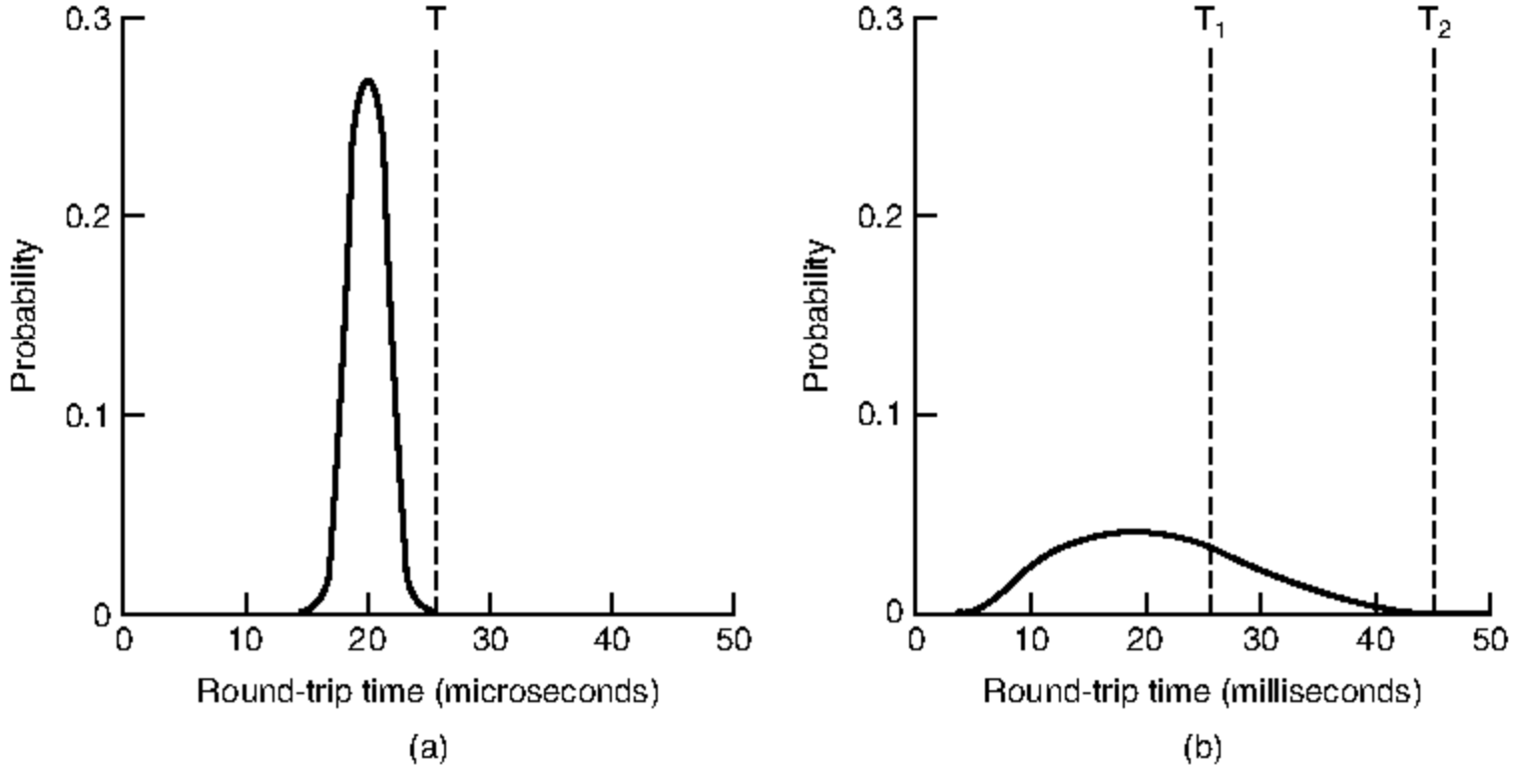


Figure 6-42. (a) Probability density of acknowledgement arrival times in the data link layer. (b) Probability density of acknowledgement arrival times for TCP.

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. 6-42(b) than Fig. 6-42(a). It is larger and more variable. 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. 6-42(b), unnecessary retransmissions will occur, clogging the Internet with useless packets. If it is set too long (e.g., 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 dynamic algorithm that constantly adapts 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, *SRTT* (Smoothed Round-Trip Time), 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 also 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, *R*. It then updates *SRTT* according to the formula

$$SRTT = \alpha\, SRTT + (1 - \alpha)\, R$$

where α is a smoothing factor that determines how quickly the old values are forgotten. Typically, $\alpha = 7/8$. This kind of formula is an **EWMA** (**Exponentially Weighted Moving Average**) or low-pass filter that discards noise in the samples.

Even given a good value of *SRTT*, choosing a suitable retransmission timeout is a nontrivial matter. Initial implementations of TCP used $2xRTT$, but experience showed that a constant value was too inflexible because it failed to respond when the variance went up. In particular, queueing models of random (i.e., Poisson) traffic predict that when the load approaches capacity, the delay becomes large and highly variable. This can lead to the retransmission timer firing and a copy of the packet being retransmitted although the original packet is still transiting the network. It is all the more likely to happen under conditions of high load, which is the worst time at which to send additional packets into the network.

To fix this problem, Jacobson proposed making the timeout value sensitive to the variance in round-trip times as well as the smoothed round-trip time. This change requires keeping track of another smoothed variable, *RTTVAR* (Round-Trip Time VARiation) that is updated using the formula

$$RTTVAR = \beta\, RTTVAR + (1 - \beta)\, |SRTT - R|$$

This is an EWMA as before, and typically $\beta = 3/4$. The retransmission timeout, *RTO*, is set to be

$$RTO = SRTT + 4 \times RTTVAR$$

The choice of the factor 4 is somewhat arbitrary, but multiplication by 4 can be done with a single shift, and less than 1% of all packets come in more than four standard deviations late. Note that *RTTVAR* is not exactly the same as the standard deviation (it is really the mean deviation), but it is close enough in practice. Jacobson's paper is full of clever tricks to compute timeouts using only integer adds, subtracts, and shifts. This economy is not needed for modern hosts, but it has become part of the culture that allows TCP to run on all manner of devices, from supercomputers down to tiny devices. So far nobody has put it on an RFID chip, but someday? Who knows.

More details of how to compute this timeout, including initial settings of the variables, are given in RFC 2988. The retransmission timer is also held to a minimum of 1 second, regardless of the estimates. This is a conservative value chosen to prevent spurious retransmissions based on measurements (Allman and Paxson, 1999).

One problem that occurs with gathering the samples, *R*, of the round-trip time 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 retransmission timeout. Phil Karn discovered this problem the hard way. Karn is an amateur radio enthusiast interested in transmitting TCP/IP packets by ham radio, a notoriously unreliable medium. He made a simple proposal: do not update estimates on any segments that have been retransmitted. Additionally, the timeout is doubled on each successive retransmission until the segments get through the first time. This fix is called **Karn's algorithm** (Karn and Partridge, 1987). Most TCP implementations use it.

The retransmission timer is not the only timer 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 the sender and the receiver are each waiting for the 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 0, 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 whether 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 *TIME 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.

6.5.10 TCP Congestion Control

We have saved one of the key functions of TCP for last: congestion control. When the load offered to any network is more than it can handle, congestion builds up. The Internet is no exception. The network layer detects congestion when queues grow large at routers and tries to manage it, if only by dropping packets. It is up to the transport layer to receive congestion feedback from the network layer and slow down the rate of traffic that it is sending into the network. In the Internet, TCP plays the main role in controlling congestion, as well as the main role in reliable transport. That is why it is such a special protocol.

We covered the general situation of congestion control in Sec. 6.3. One key takeaway was that a transport protocol using an AIMD (Additive Increase Multiplicative Decrease) control law in response to binary congestion signals from the network would converge to a fair and efficient bandwidth allocation. TCP congestion control is based on implementing this approach using a window and with packet loss as the binary signal. To do so, TCP maintains a **congestion window**

whose size is the number of bytes the sender may have in the network at any time. The corresponding rate is the window size divided by the round-trip time of the connection. TCP adjusts the size of the window according to the AIMD rule.

Recall that the congestion window is maintained *in addition* to the flow control window, which specifies the number of bytes that the receiver can buffer. Both windows are tracked in parallel, and the number of bytes that may be sent is the smaller of the two windows. Thus, the effective window is the smaller of what the sender thinks is all right and what the receiver thinks is all right. It takes two to tango. TCP will stop sending data if either the congestion or the flow control window is temporarily full. If the receiver says "send 64 KB" but the sender knows that bursts of more than 32 KB clog the network, it will send 32 KB. On the other hand, if the receiver says "send 64 KB" and the sender knows that bursts of up to 128 KB get through effortlessly, it will send the full 64 KB requested. The flow control window was described earlier, and in what follows we will only describe the congestion window.

Modern congestion control was added to TCP largely through the efforts of Van Jacobson (1988). It is a fascinating story. Starting in 1986, the growing popularity of the early Internet led to the first occurrence of what became known as a **congestion collapse**, a prolonged period during which goodput dropped precipitously (i.e., by more than a factor of 100) due to congestion in the network. Jacobson (and many others) set out to understand what was happening and remedy the situation.

The high-level fix that Jacobson implemented was to approximate an AIMD congestion window. The interesting part, and much of the complexity of TCP congestion control, is how he added this to an existing implementation without changing any of the message formats, which made it instantly deployable. To start, he observed that packet loss is a suitable signal of congestion. This signal comes a little late (as the network is already congested) but it is quite dependable. After all, it is difficult to build a router that does not drop packets when it is overloaded. This fact is unlikely to change. Even when terabyte memories appear to buffer vast numbers of packets, we will probably have terabit/sec networks to fill up those memories.

However, using packet loss as a congestion signal depends on transmission errors being relatively rare. This is not normally the case for wireless links such as 802.11, which is why they include their own retransmission mechanism at the link layer. Because of wireless retransmissions, network layer packet loss due to transmission errors is normally masked on wireless networks. It is also rare on other links because wires and optical fibers typically have low bit-error rates.

All the Internet TCP algorithms assume that lost packets are caused by congestion and monitor timeouts and look for signs of trouble the way miners watch their canaries. A good retransmission timer is needed to detect packet loss signals accurately and in a timely manner. We have already discussed how the TCP retransmission timer includes estimates of the mean and variation in round-trip

times. Fixing this timer, by including the variation factor, was an important step in Jacobson's work. Given a good retransmission timeout, the TCP sender can track the outstanding number of bytes, which are loading the network. It simply looks at the difference between the sequence numbers that are transmitted and acknowledged.

Now it seems that our task is easy. All we need to do is to track the congestion window, using sequence and acknowledgement numbers, and adjust the congestion window using an AIMD rule. As you might have expected, it is more complicated than that. A first consideration is that the way packets are sent into the network, even over short periods of time, must be matched to the network path. Otherwise the traffic will cause congestion. For example, consider a host with a congestion window of 64 KB attached to a 1-Gbps switched Ethernet. If the host sends the entire window at once, this burst of traffic may travel over a slow 1-Mbps ADSL line further along the path. The burst that took only half a millisecond on the 1-Gbps line will clog the 1-Mbps line for half a second, completely disrupting protocols such as voice over IP. This behavior might be a good idea for a protocol designed to cause congestion, but not for a protocol to control it.

However, it turns out that we can use small bursts of packets to our advantage. Fig. 6-43 shows what happens when a sender on a fast network (the 1-Gbps link) sends a small burst of four packets to a receiver on a slow network (the 1-Mbps link) that is the bottleneck or slowest part of the path. Initially the four packets travel over the link as quickly as they can be sent by the sender. At the router, they are queued while being sent because it takes longer to send a packet over the slow link than to receive the next packet over the fast link. But the queue is not large because only a small number of packets were sent at once. Note the increased length of the packets on the slow link. The same packet, of 1 KB say, is now longer because it takes more time to send it on a slow link than on a fast one.

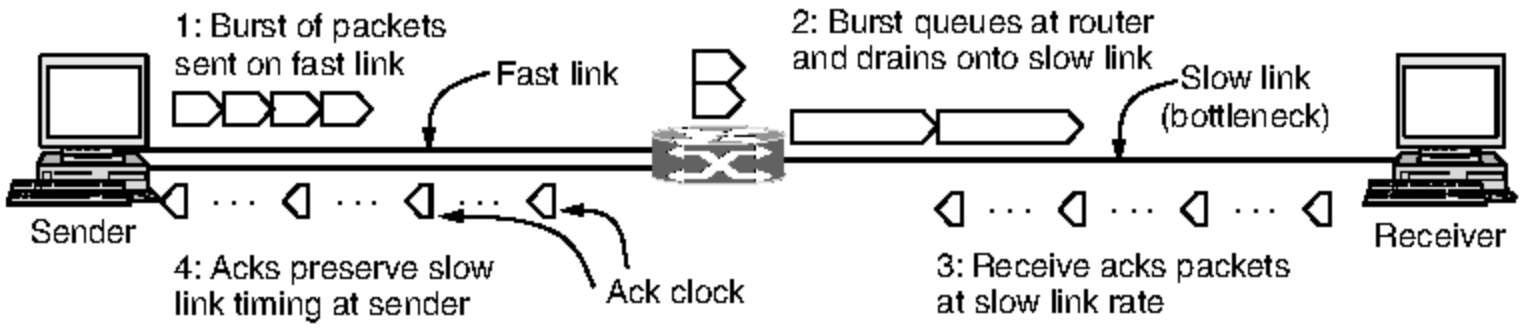


Figure 6-43. A burst of packets from a sender and the returning ack clock.

Eventually the packets get to the receiver, where they are acknowledged. The times for the acknowledgements reflect the times at which the packets arrived at the receiver after crossing the slow link. They are spread out compared to the original packets on the fast link. As these acknowledgements travel over the network and back to the sender they preserve this timing.

The key observation is this: the acknowledgements return to the sender at about the rate that packets can be sent over the slowest link in the path. This is precisely the rate that the sender wants to use. If it injects new packets into the network at this rate, they will be sent as fast as the slow link permits, but they will not queue up and congest any router along the path. This timing is known as an **ack clock**. It is an essential part of TCP. By using an ack clock, TCP smoothes out traffic and avoids unnecessary queues at routers.

A second consideration is that the AIMD rule will take a very long time to reach a good operating point on fast networks if the congestion window is started from a small size. Consider a modest network path that can support 10 Mbps with an RTT of 100 msec. The appropriate congestion window is the bandwidth-delay product, which is 1 Mbit or 100 packets of 1250 bytes each. If the congestion window starts at 1 packet and increases by 1 packet every RTT, it will be 100 RTTs or 10 seconds before the connection is running at about the right rate. That is a long time to wait just to get to the right speed for a transfer. We could reduce this startup time by starting with a larger initial window, say of 50 packets. But this window would be far too large for slow or short links. It would cause congestion if used all at once, as we have just described.

Instead, the solution Jacobson chose to handle both of these considerations is a mix of linear and multiplicative increase. When a connection is established, the sender initializes the congestion window to a small initial value of at most four segments; the details are described in RFC 3390, and the use of four segments is an increase from an earlier initial value of one segment based on experience. The sender then sends the initial window. The packets will take a round-trip time to be acknowledged. For each segment that is acknowledged before the retransmission timer goes off, the sender adds one segment's worth of bytes to the congestion window. Plus, as that segment has been acknowledged, there is now one less segment in the network. The upshot is that every acknowledged segment allows two more segments to be sent. The congestion window is doubling every round-trip time.

This algorithm is called **slow start**, but it is not slow at all—it is exponential growth—except in comparison to the previous algorithm that let an entire flow control window be sent all at once. Slow start is shown in Fig. 6-44. In the first round-trip time, the sender injects one packet into the network (and the receiver receives one packet). Two packets are sent in the next round-trip time, then four packets in the third round-trip time.

Slow-start works well over a range of link speeds and round-trip times, and uses an ack clock to match the rate of sender transmissions to the network path. Take a look at the way acknowledgements return from the sender to the receiver in Fig. 6-44. When the sender gets an acknowledgement, it increases the congestion window by one and immediately sends two packets into the network. (One packet is the increase by one; the other packet is a replacement for the packet that has been acknowledged and left the network. At all times, the number of

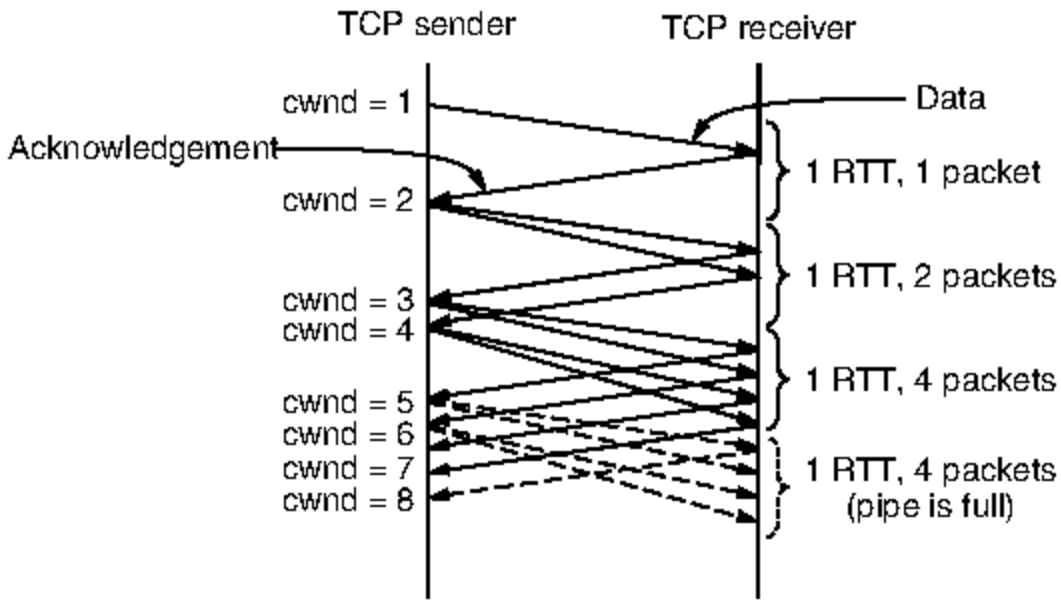


Figure 6-44. Slow start from an initial congestion window of one segment.

unacknowledged packets is given by the congestion window.) However, these two packets will not necessarily arrive at the receiver as closely spaced as when they were sent. For example, suppose the sender is on a 100-Mbps Ethernet. Each packet of 1250 bytes takes 100 μsec to send. So the delay between the packets can be as small as 100 μsec. The situation changes if these packets go across a 1-Mbps ADSL link anywhere along the path. It now takes 10 msec to send the same packet. This means that the minimum spacing between the two packets has grown by a factor of 100. Unless the packets have to wait together in a queue on a later link, the spacing will remain large.

In Fig. 6-44, this effect is shown by enforcing a minimum spacing between data packets arriving at the receiver. The same spacing is kept when the receiver sends acknowledgements, and thus when the sender receives the acknowledgements. If the network path is slow, acknowledgements will come in slowly (after a delay of an RTT). If the network path is fast, acknowledgements will come in quickly (again, after the RTT). All the sender has to do is follow the timing of the ack clock as it injects new packets, which is what slow start does.

Because slow start causes exponential growth, eventually (and sooner rather than later) it will send too many packets into the network too quickly. When this happens, queues will build up in the network. When the queues are full, one or more packets will be lost. After this happens, the TCP sender will time out when an acknowledgement fails to arrive in time. There is evidence of slow start growing too fast in Fig. 6-44. After three RTTs, four packets are in the network. These four packets take an entire RTT to arrive at the receiver. That is, a congestion window of four packets is the right size for this connection. However, as these packets are acknowledged, slow start continues to grow the congestion window, reaching eight packets in another RTT. Only four of these packets can reach the receiver in one RTT, no matter how many are sent. That is, the network pipe is full. Additional packets placed into the network by the sender will build up in

router queues, since they cannot be delivered to the receiver quickly enough. Congestion and packet loss will occur soon.

To keep slow start under control, the sender keeps a threshold for the connection called the **slow start threshold**. Initially this value is set arbitrarily high, to the size of the flow control window, so that it will not limit the connection. TCP keeps increasing the congestion window in slow start until a timeout occurs or the congestion window exceeds the threshold (or the receiver's window is filled).

Whenever a packet loss is detected, for example, by a timeout, the slow start threshold is set to be half of the congestion window and the entire process is restarted. The idea is that the current window is too large because it caused congestion previously that is only now detected by a timeout. Half of the window, which was used successfully at an earlier time, is probably a better estimate for a congestion window that is close to the path capacity but will not cause loss. In our example in Fig. 6-44, growing the congestion window to eight packets may cause loss, while the congestion window of four packets in the previous RTT was the right value. The congestion window is then reset to its small initial value and slow start resumes.

Whenever the slow start threshold is crossed, TCP switches from slow start to additive increase. In this mode, the congestion window is increased by one segment every round-trip time. Like slow start, this is usually implemented with an increase for every segment that is acknowledged, rather than an increase once per RTT. Call the congestion window *cwnd* and the maximum segment size *MSS*. A common approximation is to increase *cwnd* by $(MSS \times MSS)/cwnd$ for each of the $cwnd/MSS$ packets that may be acknowledged. This increase does not need to be fast. The whole idea is for a TCP connection to spend a lot of time with its congestion window close to the optimum value—not so small that throughput will be low, and not so large that congestion will occur.

Additive increase is shown in Fig. 6-45 for the same situation as slow start. At the end of every RTT, the sender's congestion window has grown enough that it can inject an additional packet into the network. Compared to slow start, the linear rate of growth is much slower. It makes little difference for small congestion windows, as is the case here, but a large difference in the time taken to grow the congestion window to 100 segments, for example.

There is something else that we can do to improve performance too. The defect in the scheme so far is waiting for a timeout. Timeouts are relatively long because they must be conservative. After a packet is lost, the receiver cannot acknowledge past it, so the acknowledgement number will stay fixed, and the sender will not be able to send any new packets into the network because its congestion window remains full. This condition can continue for a relatively long period until the timer fires and the lost packet is retransmitted. At that stage, TCP slow starts again.

There is a quick way for the sender to recognize that one of its packets has been lost. As packets beyond the lost packet arrive at the receiver, they trigger

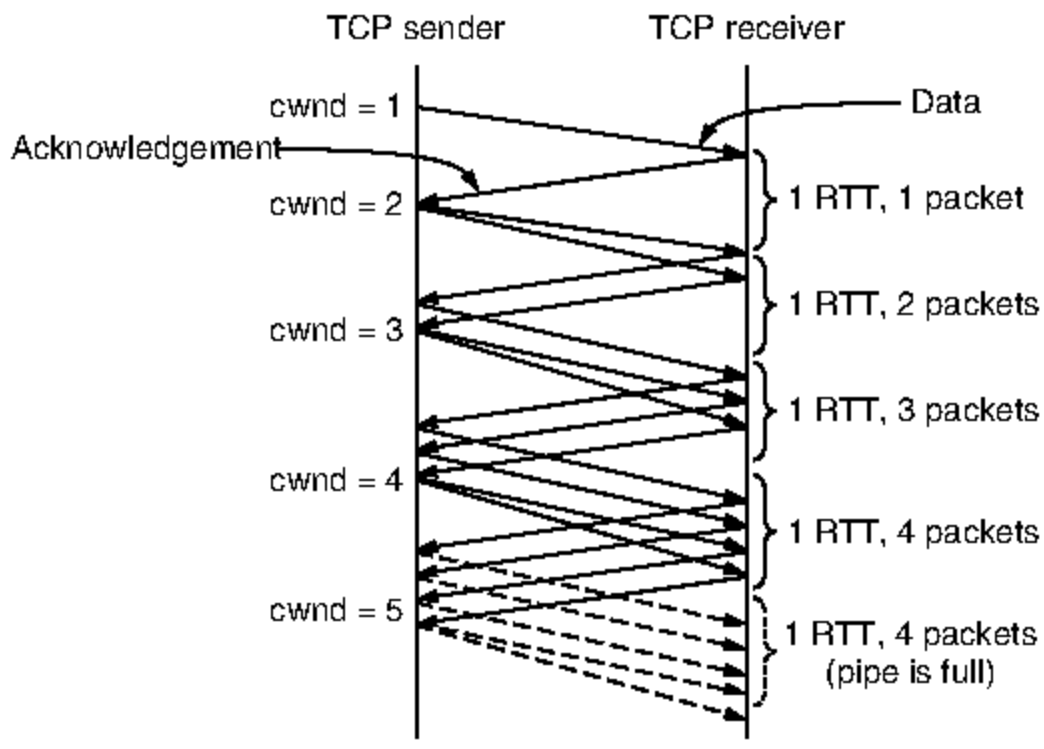


Figure 6-45. Additive increase from an initial congestion window of one segment.

acknowledgements that return to the sender. These acknowledgements bear the same acknowledgement number. They are called **duplicate acknowledgements**. Each time the sender receives a duplicate acknowledgement, it is likely that another packet has arrived at the receiver and the lost packet still has not shown up.

Because packets can take different paths through the network, they can arrive out of order. This will trigger duplicate acknowledgements even though no packets have been lost. However, this is uncommon in the Internet much of the time. When there is reordering across multiple paths, the received packets are usually not reordered too much. Thus, TCP somewhat arbitrarily assumes that three duplicate acknowledgements imply that a packet has been lost. The identity of the lost packet can be inferred from the acknowledgement number as well. It is the very next packet in sequence. This packet can then be retransmitted right away, before the retransmission timeout fires.

This heuristic is called **fast retransmission**. After it fires, the slow start threshold is still set to half the current congestion window, just as with a timeout. Slow start can be restarted by setting the congestion window to one packet. With this window size, a new packet will be sent after the one round-trip time that it takes to acknowledge the retransmitted packet along with all data that had been sent before the loss was detected.

An illustration of the congestion algorithm we have built up so far is shown in Fig. 6-46. This version of TCP is called TCP Tahoe after the 4.2BSD Tahoe release in 1988 in which it was included. The maximum segment size here is 1 KB. Initially, the congestion window was 64 KB, but a timeout occurred, so the threshold is set to 32 KB and the congestion window to 1 KB for transmission 0. The congestion window grows exponentially until it hits the threshold (32 KB). The

window is increased every time a new acknowledgement arrives rather than continuously, which leads to the discrete staircase pattern. After the threshold is passed, the window grows linearly. It is increased by one segment every RTT.

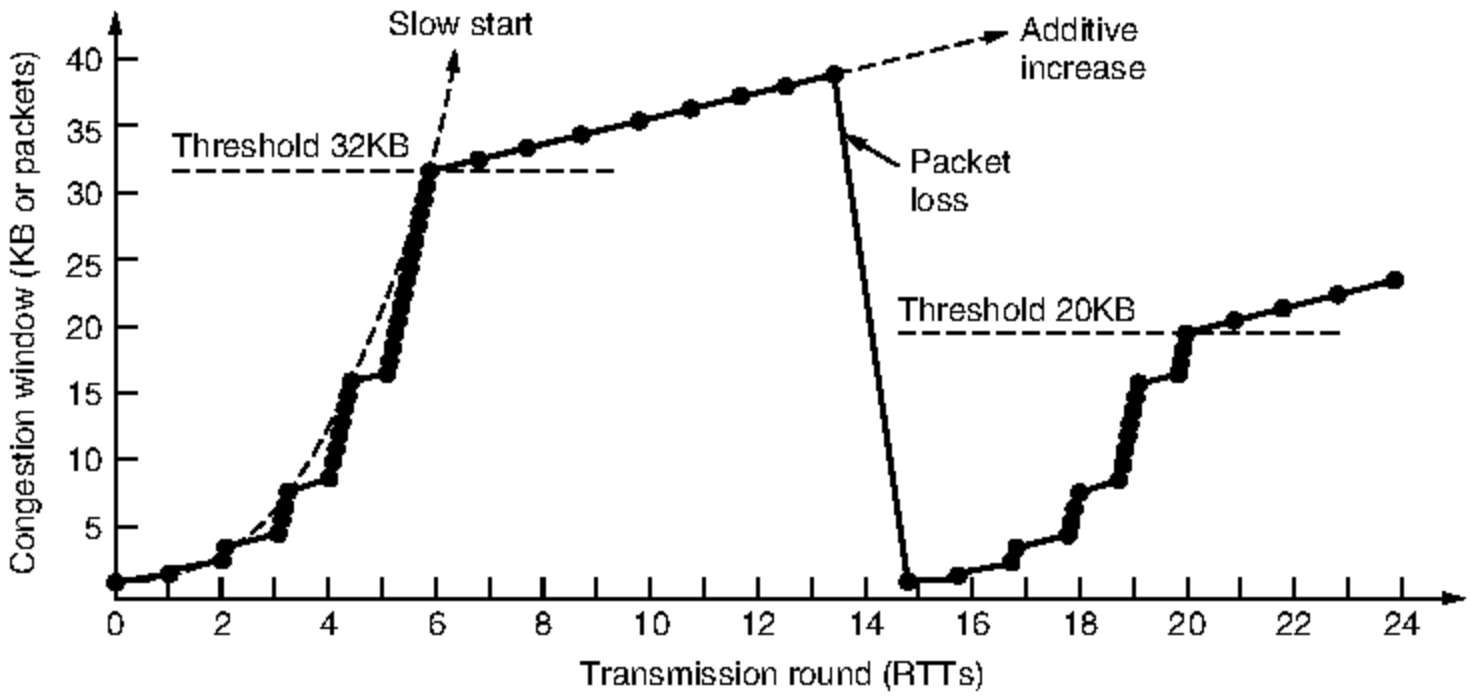


Figure 6-46. Slow start followed by additive increase in TCP Tahoe.

The transmissions in round 13 are unlucky (they should have known), and one of them is lost in the network. This is detected when three duplicate acknowledgements arrive. At that time, the lost packet is retransmitted, the threshold is set to half the current window (by now 40 KB, so half is 20 KB), and slow start is initiated all over again. Restarting with a congestion window of one packet takes one round-trip time for all of the previously transmitted data to leave the network and be acknowledged, including the retransmitted packet. The congestion window grows with slow start as it did previously, until it reaches the new threshold of 20 KB. At that time, the growth becomes linear again. It will continue in this fashion until another packet loss is detected via duplicate acknowledgements or a timeout (or the receiver's window becomes the limit).

TCP Tahoe (which included good retransmission timers) provided a working congestion control algorithm that solved the problem of congestion collapse. Jacobson realized that it is possible to do even better. At the time of the fast retransmission, the connection is running with a congestion window that is too large, but it is still running with a working ack clock. Every time another duplicate acknowledgement arrives, it is likely that another packet has left the network. Using duplicate acknowledgements to count the packets in the network, makes it possible to let some packets exit the network and continue to send a new packet for each additional duplicate acknowledgement.

Fast recovery is the heuristic that implements this behavior. It is a temporary mode that aims to maintain the ack clock running with a congestion window that is the new threshold, or half the value of the congestion window at the time of the

fast retransmission. To do this, duplicate acknowledgements are counted (including the three that triggered fast retransmission) until the number of packets in the network has fallen to the new threshold. This takes about half a round-trip time. From then on, a new packet can be sent for each duplicate acknowledgement that is received. One round-trip time after the fast retransmission, the lost packet will have been acknowledged. At that time, the stream of duplicate acknowledgements will cease and fast recovery mode will be exited. The congestion window will be set to the new slow start threshold and grows by linear increase.

The upshot of this heuristic is that TCP avoids slow start, except when the connection is first started and when a timeout occurs. The latter can still happen when more than one packet is lost and fast retransmission does not recover adequately. Instead of repeated slow starts, the congestion window of a running connection follows a **sawtooth** pattern of additive increase (by one segment every RTT) and multiplicative decrease (by half in one RTT). This is exactly the AIMD rule that we sought to implement.

This sawtooth behavior is shown in Fig. 6-47. It is produced by TCP Reno, named after the 4.3BSD Reno release in 1990 in which it was included. TCP Reno is essentially TCP Tahoe plus fast recovery. After an initial slow start, the congestion window climbs linearly until a packet loss is detected by duplicate acknowledgements. The lost packet is retransmitted and fast recovery is used to keep the ack clock running until the retransmission is acknowledged. At that time, the congestion window is resumed from the new slow start threshold, rather than from 1. This behavior continues indefinitely, and the connection spends most of the time with its congestion window close to the optimum value of the bandwidth-delay product.

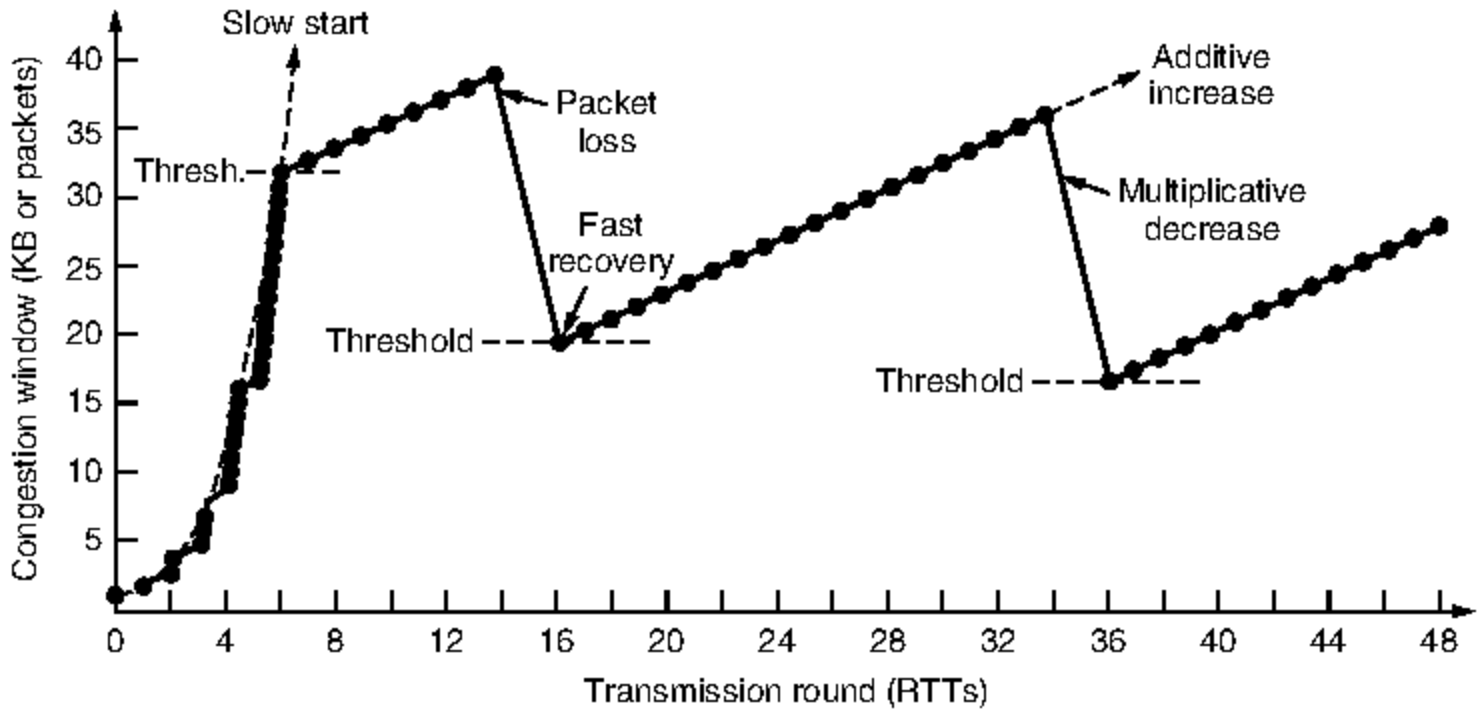


Figure 6-47. Fast recovery and the sawtooth pattern of TCP Reno.

TCP Reno with its mechanisms for adjusting the congestion window has formed the basis for TCP congestion control for more than two decades. Most of

the changes in the intervening years have adjusted these mechanisms in minor ways, for example, by changing the choices of the initial window and removing various ambiguities. Some improvements have been made for recovering from two or more losses in a window of packets. For example, the TCP NewReno version uses a partial advance of the acknowledgement number after a retransmission to find and repair another loss (Hoe, 1996), as described in RFC 3782. Since the mid-1990s, several variations have emerged that follow the principles we have described but use slightly different control laws. For example, Linux uses a variant called CUBIC TCP (Ha et al., 2008) and Windows includes a variant called Compound TCP (Tan et al., 2006).

Two larger changes have also affected TCP implementations. First, much of the complexity of TCP comes from inferring from a stream of duplicate acknowledgements which packets have arrived and which packets have been lost. The cumulative acknowledgement number does not provide this information. A simple fix is the use of **SACK** (**Selective ACKnowledgements**), which lists up to three ranges of bytes that have been received. With this information, the sender can more directly decide what packets to retransmit and track the packets in flight to implement the congestion window.

When the sender and receiver set up a connection, they each send the *SACK permitted* TCP option to signal that they understand selective acknowledgements. Once SACK is enabled for a connection, it works as shown in Fig. 6-48. A receiver uses the TCP *Acknowledgement number* field in the normal manner, as a cumulative acknowledgement of the highest in-order byte that has been received. When it receives packet 3 out of order (because packet 2 was lost), it sends a *SACK option* for the received data along with the (duplicate) cumulative acknowledgement for packet 1. The *SACK option* gives the byte ranges that have been received above the number given by the cumulative acknowledgement. The first range is the packet that triggered the duplicate acknowledgement. The next ranges, if present, are older blocks. Up to three ranges are commonly used. By the time packet 6 is received, two SACK byte ranges are used to indicate that packet 6 and packets 3 to 4 have been received, in addition to all packets up to packet 1. From the information in each *SACK option* that it receives, the sender can decide which packets to retransmit. In this case, retransmitting packets 2 and 5 would be a good idea.

SACK is strictly advisory information. The actual detection of loss using duplicate acknowledgements and adjustments to the congestion window proceed just as before. However, with SACK, TCP can recover more easily from situations in which multiple packets are lost at roughly the same time, since the TCP sender knows which packets have not been received. SACK is now widely deployed. It is described in RFC 2883, and TCP congestion control using SACK is described in RFC 3517.

The second change is the use of ECN (Explicit Congestion Notification) in addition to packet loss as a congestion signal. ECN is an IP layer mechanism to

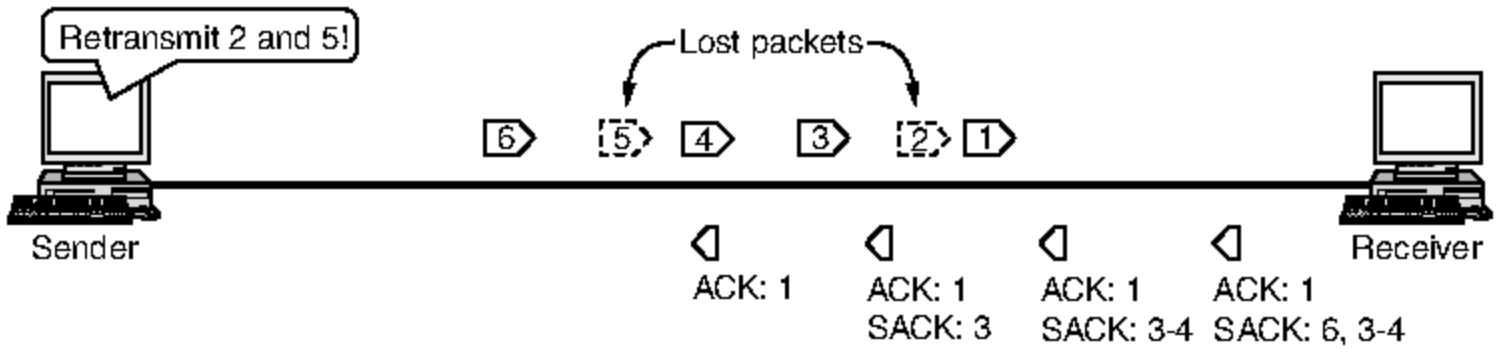


Figure 6-48. Selective acknowledgements.

notify hosts of congestion that we described in Sec. 5.3.4. With it, the TCP receiver can receive congestion signals from IP.

The use of ECN is enabled for a TCP connection when both the sender and receiver indicate that they are capable of using ECN by setting the *ECE* and *CWR* bits during connection establishment. If ECN is used, each packet that carries a TCP segment is flagged in the IP header to show that it can carry an ECN signal. Routers that support ECN will set a congestion signal on packets that can carry ECN flags when congestion is approaching, instead of dropping those packets after congestion has occurred.

The TCP receiver is informed if any packet that arrives carries an ECN congestion signal. The receiver then uses the *ECE* (ECN Echo) flag to signal the TCP sender that its packets have experienced congestion. The sender tells the receiver that it has heard the signal by using the *CWR* (Congestion Window Reduced) flag.

The TCP sender reacts to these congestion notifications in exactly the same way as it does to packet loss that is detected via duplicate acknowledgements. However, the situation is strictly better. Congestion has been detected and no packet was harmed in any way. ECN is described in RFC 3168. It requires both host and router support, and is not yet widely used on the Internet.

For more information on the complete set of congestion control behaviors that are implemented in TCP, see RFC 5681.

6.5.11 The Future of TCP

As the workhorse of the Internet, TCP has been used for many applications and extended over time to give good performance over a wide range of networks. Many versions are deployed with slightly different implementations than the classic algorithms we have described, especially for congestion control and robustness against attacks. It is likely that TCP will continue to evolve with the Internet. We will mention two particular issues.

The first one is that TCP does not provide the transport semantics that all applications want. For example, some applications want to send messages or records whose boundaries need to be preserved. Other applications work with a group of

related conversations, such as a Web browser that transfers several objects from the same server. Still other applications want better control over the network paths that they use. TCP with its standard sockets interface does not meet these needs well. Essentially, the application has the burden of dealing with any problem not solved by TCP. This has led to proposals for new protocols that would provide a slightly different interface. Two examples are SCTP (Stream Control Transmission Protocol), defined in RFC 4960, and SST (Structured Stream Transport) (Ford, 2007). However, whenever someone proposes changing something that has worked so well for so long, there is always a huge battle between the "Users are demanding more features" and "If it ain't broke, don't fix it" camps.

The second issue is congestion control. You may have expected that this is a solved problem after our deliberations and the mechanisms that have been developed over time. Not so. The form of TCP congestion control that we described, and which is widely used, is based on packet losses as a signal of congestion. When Padhye et al. (1998) modeled TCP throughput based on the sawtooth pattern, they found that the packet loss rate must drop off rapidly with increasing speed. To reach a throughput of 1 Gbps with a round-trip time of 100 ms and 1500 byte packets, one packet can be lost approximately every 10 minutes. That is a packet loss rate of 2×10^{-8}, which is incredibly small. It is too infrequent to serve as a good congestion signal, and any other source of loss (e.g., packet transmission error rates of 10^{-7}) can easily dominate it, limiting the throughput.

This relationship has not been a problem in the past, but networks are getting faster and faster, leading many people to revisit congestion control. One possibility is to use an alternate congestion control in which the signal is not packet loss at all. We gave several examples in Sec. 6.2. The signal might be round-trip time, which grows when the network becomes congested, as is used by FAST TCP (Wei et al., 2006). Other approaches are possible too, and time will tell which is the best.

6.6 PERFORMANCE ISSUES

Performance issues are very important in computer networks. When hundreds or thousands of computers are interconnected, 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 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 some rules of thumb gained from hard experience and present examples taken from the real world. We have delayed this discussion until we studied the transport layer because the performance that applications receive

depends on the combined performance of the transport, network and link layers, and to be able to use TCP as an example in various places.

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

1. Performance problems.
2. Measuring network performance.
3. Host design for fast networks.
4. Fast segment processing.
5. Header compression.
6. Protocols for "long fat" networks.

These aspects consider network performance both at the host and across the network, and as networks are increased in speed and size.

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 this and 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 host 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. As an example, if a segment contains a bad parameter (e.g., the port 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 segment is broadcast to 1000 machines: each one might send back an error message. The resulting **broadcast storm** could cripple the network. UDP suffered from this problem until the ICMP protocol was changed to cause hosts to refrain from responding to errors in UDP segments sent to broadcast addresses. Wireless networks must be particularly careful to avoid unchecked broadcast responses because broadcast occurs naturally and the wireless bandwidth is limited.

A second example of synchronous overload is what happens after an electrical power failure. When the power comes back on, all the machines simultaneously start rebooting. A typical reboot sequence might require first going to some (DHCP) 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 in a data center all do this at once, the server will probably collapse under the load.

Even in the absence of synchronous overloads and the presence of sufficient resources, 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, flow control will slow down segment reception and limit performance. This was a problem for many TCP connections as the Internet became faster but the default size of the flow control window stayed fixed at 64 KB.

Another tuning issue is setting timeouts. When a segment is sent, a timer is set to guard against loss of the segment. 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 segment is lost. Other tunable parameters include how long to wait for data on which to piggyback before sending a separate acknowledgement, and how many retransmissions to make before giving up.

Another performance problem that occurs with real-time applications like audio and video is jitter. Having enough bandwidth on average is not sufficient for good performance. Short transmission delays are also required. Consistently achieving short delays demands careful engineering of the load on the network, quality-of-service support at the link and network layers, or both.

6.6.2 Network Performance Measurement

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. Much of the discussion below is based on the seminal work of Mogul (1993).

Measurements can be made in different ways and at many locations (both in the protocol stack and physically). The most basic kind of measurement is to start a timer when beginning some activity and see how long that activity takes. For example, knowing how long it takes for a segment 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 segments). 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. We list a few of them here. 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 segment, but repeat the measurement, say, one million times and take the average. Startup effects, such as the 802.16 NIC or cable modem getting a bandwidth reservation after an idle period, can

slow the first segment, and queueing introduces variability. 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 network conditions 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 11 A.M., and 1 P.M., but no congestion at noon (when all the users are at lunch).

With wireless networks, location is an important variable because of signal propagation. Even a measurement node placed close to a wireless client may not observe the same packets as the client due to differences in the antennas. It is best to take measurements from the wireless client under study to see what it sees. Failing that, it is possible to use techniques to combine the wireless measurements taken at different vantage points to gain a more complete picture of what is going on (Mahajan et al., 2006).

Caching Can Wreak Havoc with Measurements

Repeating a measurement many times will return an unexpectedly fast answer if the protocols use caching mechanisms. For instance, fetching a Web page or looking up a DNS name (to find the IP address) may involve a network exchange the first time, and then return the answer from a local cache without sending any packets over the network. The results from such a measurement are essentially worthless (unless you want to measure cache performance).

Buffering can have a similar effect. TCP/IP performance tests have been known to report that UDP can achieve a performance substantially higher than the network 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 test program thinks they have all been transmitted.

Caution is advised to be absolutely sure that you understand how data can be cached and buffered as part of a network operation.

Be Sure That Nothing Unexpected Is Going On during Your Tests

Making measurements at the same time that some user has decided to run a video conference over your network will often give different results than if there is no video conference. It is best to run tests on an idle network 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 when the automatic backup program begins copying all the disks to tape. Or, there might be heavy traffic for your wonderful Web pages from distant time zones.

Wireless networks are challenging in this respect because it is often not possible to separate them from all sources of interference. Even if there are no other wireless networks sending traffic nearby, someone may microwave popcorn and inadvertently cause interference that degrades 802.11 performance. For these reasons, it is a good practice to monitor the overall network activity so that you can at least realize when something unexpected does happen.

Be Careful When Using a Coarse-Grained Clock

Computer clocks function by incrementing some counter at regular intervals. For example, a millisecond timer adds 1 to a counter every 1 msec. Using such a timer to measure an event that takes less than 1 msec is possible but requires some care. Some computers have more accurate clocks, of course, but there are always shorter events to measure too. Note that clocks are not always as accurate as the precision with which the time is returned when they are read.

To measure the time to make a TCP connection, for example, the clock (say, in milliseconds) should be read out when the transport layer code is entered and again when it is exited. If the true connection setup 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 is added up and divided by one million, the mean time will be accurate to better than 1 μsec.

Be Careful about Extrapolating the Results

Suppose that you make measurements with simulated network loads running from 0 (idle) to 0.4 (40% of capacity). For example, the response time to send a voice-over-IP packet over an 802.11 network might be as shown by the data points and solid line through them in Fig. 6-49. It may be tempting to extrapolate linearly, as shown by the dotted line. However, many queueing results involve a factor of $1/(1-\rho)$, where ρ is the load, so the true values may look more like the dashed line, which rises much faster than linearly when the load gets high. That is, beware contention effects that become much more pronounced at high load.

6.6.3 Host Design for Fast Networks

Measuring and tinkering can 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 redesigned from scratch.

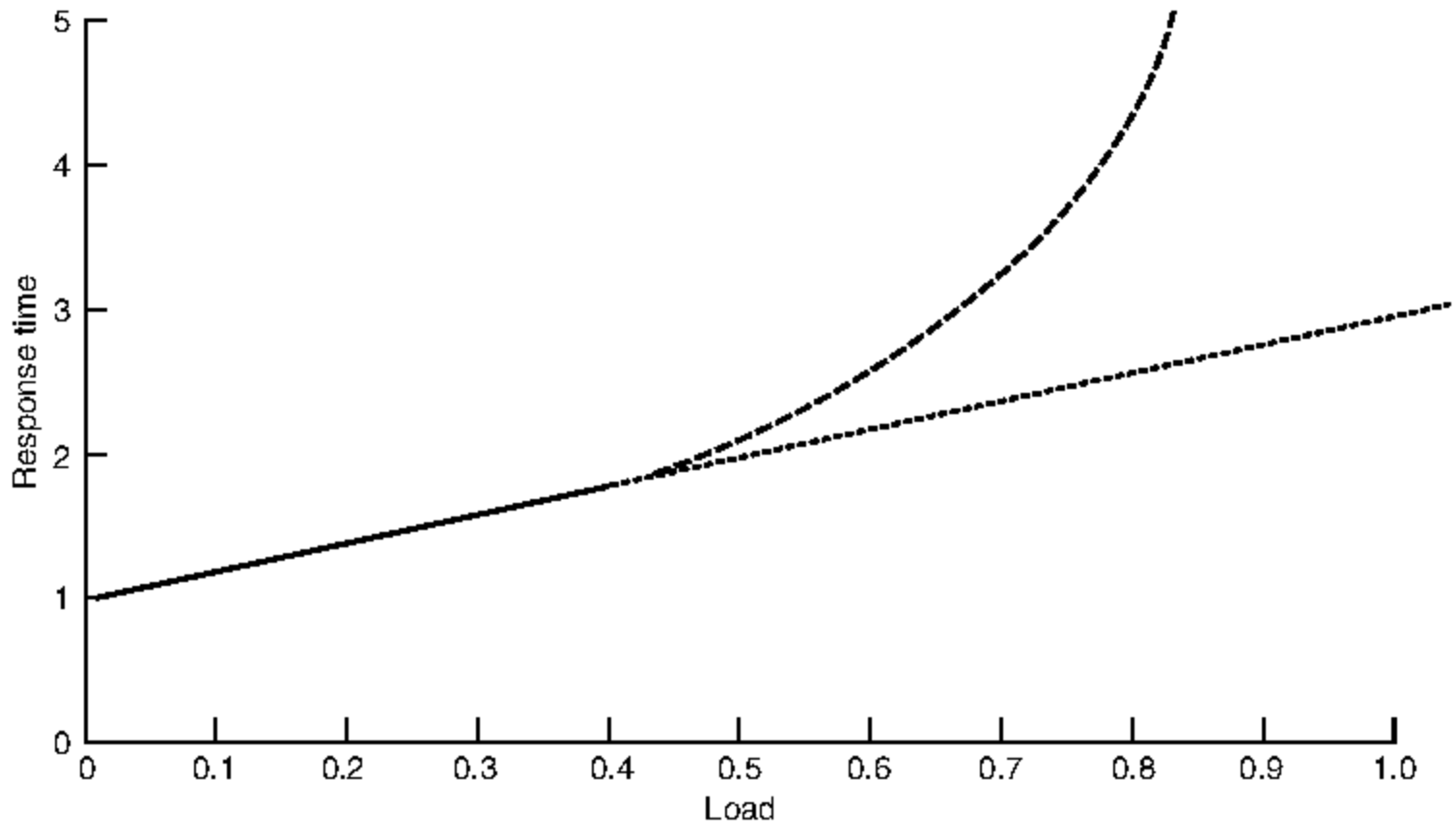


Figure 6-49. Response as a function of load.

In this section, we will present some rules of thumb for software implementation of network protocols on hosts. Surprisingly, experience shows that this is often a performance bottleneck on otherwise fast networks, for two reasons. First, NICs (Network Interface Cards) and routers have already been engineered (with hardware support) to run at "wire speed." This means that they can process packets as quickly as the packets can possibly arrive on the link. Second, the relevant performance is that which applications obtain. It is not the link capacity, but the throughput and delay after network and transport processing.

Reducing software overheads improves performance by increasing throughput and decreasing delay. It can also reduce the energy that is spent on networking, which is an important consideration for mobile computers. Most of these ideas have been common knowledge to network designers for years. They were first stated explicitly by Mogul (1993); our treatment largely follows his. Another relevant source is Metcalfe (1993).

Host Speed Is More Important Than Network Speed

Long experience has shown that in nearly all fast networks, operating system and protocol overhead dominate actual time on the wire. For example, in theory, the minimum RPC time on a 1-Gbps Ethernet is 1 μsec, corresponding to a minimum (512-byte) request followed by a minimum (512-byte) reply. In practice, overcoming the software overhead and getting the RPC time anywhere near there is a substantial achievement. It rarely happens in practice.

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

Reduce Packet Count to Reduce Overhead

Each segment has a certain amount of overhead (e.g., the header) as well as data (e.g., the payload). Bandwidth is required for both components. Processing is also required for both components (e.g., header processing and doing the checksum). When 1 million bytes are being sent, the data cost is the same no matter what the segment size is. However, using 128-byte segments means 32 times as much per-segment overhead as using 4-KB segments. The bandwidth and processing overheads add up fast to reduce throughput.

Per-packet overhead in the lower layers amplifies this effect. Each arriving packet causes a fresh interrupt if the host is keeping up. On a modern pipelined processor, each interrupt breaks the CPU pipeline, interferes with the cache, requires a change to the memory management context, voids the branch prediction table, and forces a substantial number of CPU registers to be saved. An n-fold reduction in segments sent thus reduces the interrupt and packet overhead by a factor of n.

You might say that both people and computers are poor at multitasking. This observation underlies the desire to send MTU packets that are as large as will pass along the network path without fragmentation. Mechanisms such as Nagle's algorithm and Clark's solution are also attempts to avoid sending small packets.

Minimize Data Touching

The most straightforward way to implement a layered protocol stack is with one module for each layer. Unfortunately, this leads to copying (or at least accessing the data on multiple passes) as each layer does its own work. For example, after a packet is received by the NIC, it is typically copied to a kernel buffer. From there, it is copied to a network layer buffer for network layer processing, then to a transport layer buffer for transport layer processing, and finally to the receiving application process. It is not unusual for an incoming packet to be copied three or four times before the segment enclosed in it is delivered.

All this copying can greatly degrade performance because memory operations are an order of magnitude slower than register–register instructions. For example, if 20% of the instructions actually go to memory (i.e., are cache misses), which is likely when touching incoming packets, the average instruction execution time is slowed down by a factor of 2.8 ($0.8 \times 1 + 0.2 \times 10$). Hardware assistance will not help here. The problem is too much copying by the operating system.

A clever operating system will minimize copying by combining the processing of multiple layers. For example, TCP and IP are usually implemented together (as "TCP/IP") so that it is not necessary to copy the payload of the packet as processing switches from network to transport layer. Another common trick is to perform multiple operations within a layer in a single pass over the data. For example, checksums are often computed while copying the data (when it has to be copied) and the newly computed checksum is appended to the end.

Minimize Context Switches

A related rule is that context switches (e.g., from kernel mode to user mode) are deadly. They have the bad properties of interrupts and copying combined. This cost is why transport protocols are often implemented in the kernel. Like reducing packet count, 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 segments 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 some 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. 6-50. All these context switches on each packet are wasteful of CPU time and can have a devastating effect on network performance.

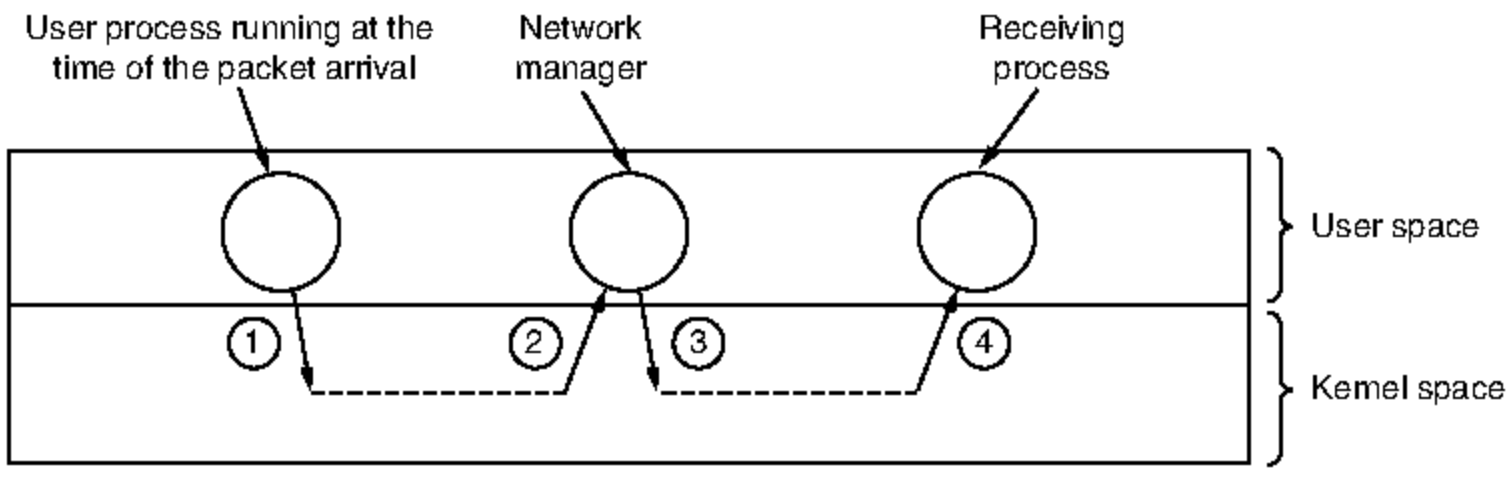


Figure 6-50. Four context switches to handle one packet with a user-space network manager.

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. All of these costs are unnecessary, and recovering from congestion 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 in the future.

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 segment being lost. A timer that goes off when it should not have uses up host resources, wastes bandwidth, and puts extra load on perhaps dozens of routers for no good reason.

6.6.4 Fast Segment Processing

Now that we have covered general rules, we will look at some specific methods for speeding up segment processing. For more information, see Clark et al. (1989), and Chase et al. (2001).

Segment processing overhead has two components: overhead per segment and overhead per byte. Both must be attacked. The key to fast segment processing is to separate out the normal, successful case (one-way data transfer) and handle it specially. Many protocols tend to emphasize what to do when something goes wrong (e.g., a packet getting lost), but to make the protocols run fast, the designer should aim to minimize processing time when everything goes right. Minimizing processing time when an error occurs is secondary.

Although a sequence of special segments is needed to get into the *ESTABLISHED* state, once there, segment 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. 6-51, the sending process traps into the kernel to do the SEND. The first thing the transport entity does is 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 segment is being sent, and enough window space is 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. Typically, this path is taken most of the time.

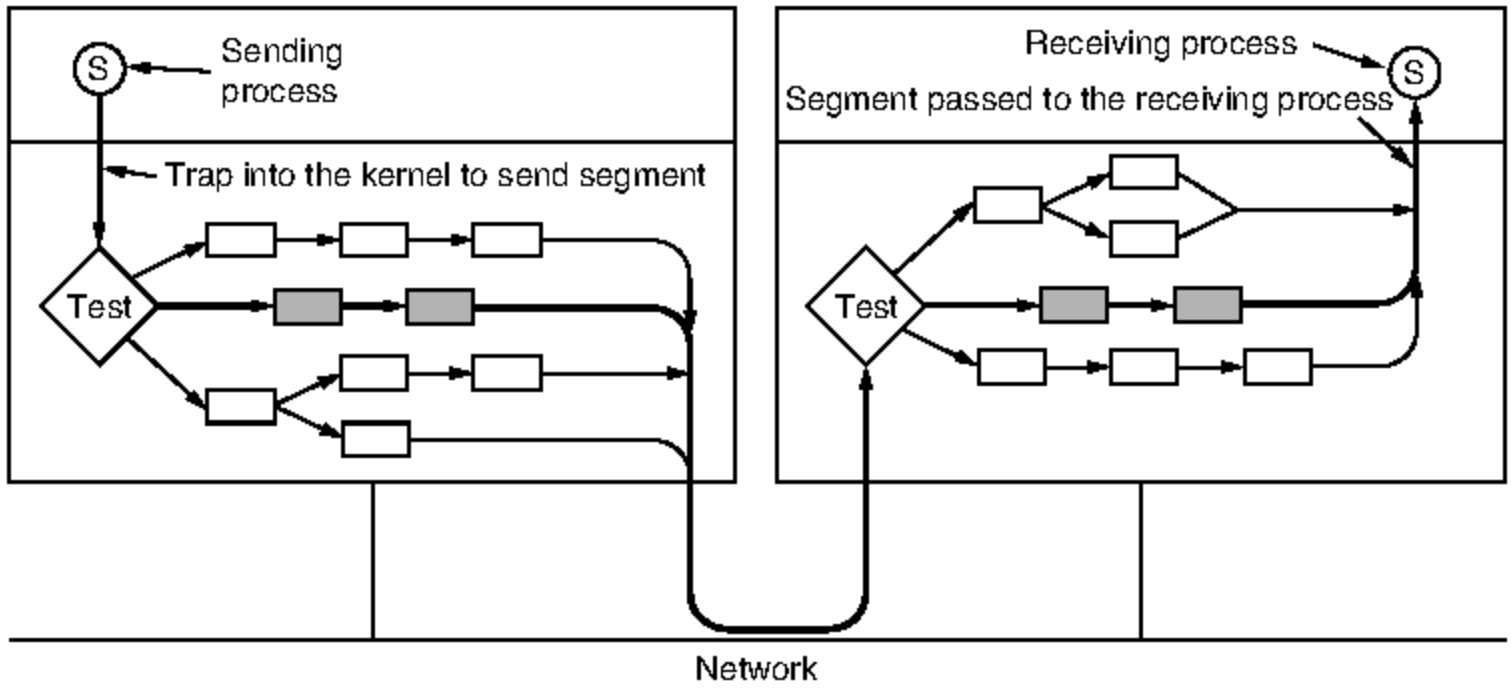


Figure 6-51. The fast path from sender to receiver is shown with a heavy line. The processing steps on this path are shaded.

In the usual case, the headers of consecutive data segments 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 segment to segment are overwritten in the buffer. Frequently, these fields are easily derived from state variables, such as the next sequence number. A pointer to the full segment 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. 6-51). Finally, the network layer gives the resulting packet to the data link layer for transmission.

As an example of how this principle works in practice, let us consider TCP/IP. Fig. 6-52(a) shows the TCP header. The fields that are the same between consecutive segments 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 segment. IP then copies its five-word prototype header [see Fig. 6-52(b)] into the buffer, fills in the *Identification* field, and computes its checksum. The packet is now ready for transmission.

Now let us look at fast path processing on the receiving side of Fig. 6-51. Step 1 is locating the connection record for the incoming segment. For TCP, the

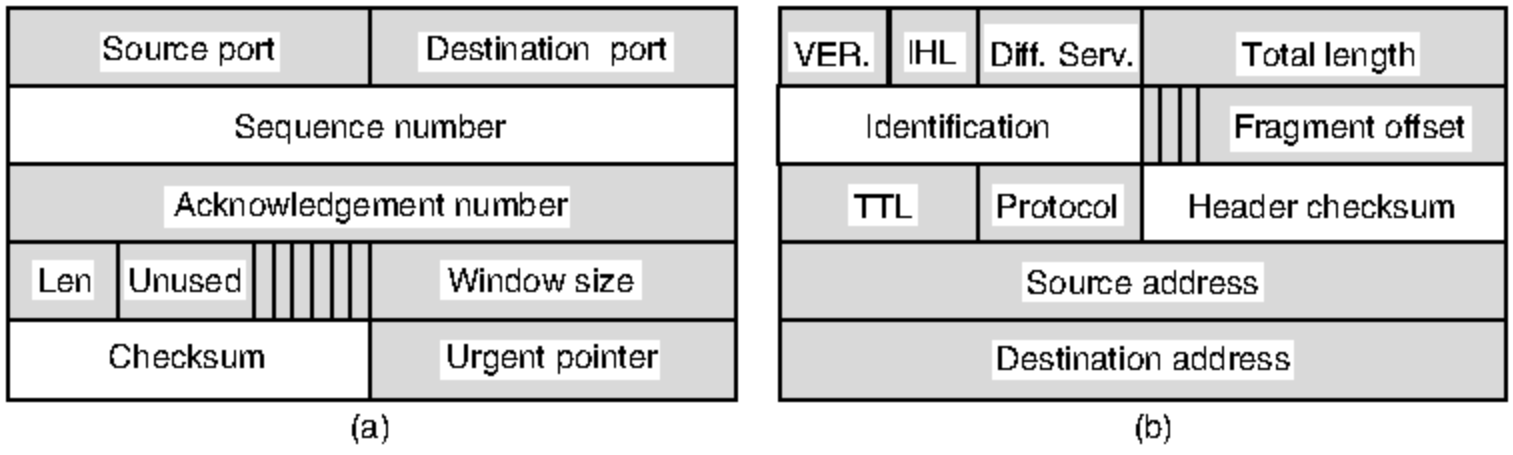


Figure 6-52. (a) TCP header. (b) IP header. In both cases, they are taken from the prototype without change.

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 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%.

The segment is checked to see if it is a normal one: the state is *ESTABLISHED*, neither side is trying to close the connection, the segment 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 then having a special procedure 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% 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 mentioned above. Timer management is important because nearly all timers set do not expire. They are set to guard against segment loss, but most segments and their acknowledgements 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 expiration 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. This way, inserting and deleting timers are expensive operations, with execution times proportional to the length of the list.

A much 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. 6-53. 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. 6-53 cannot accommodate timers beyond $T + 15$.

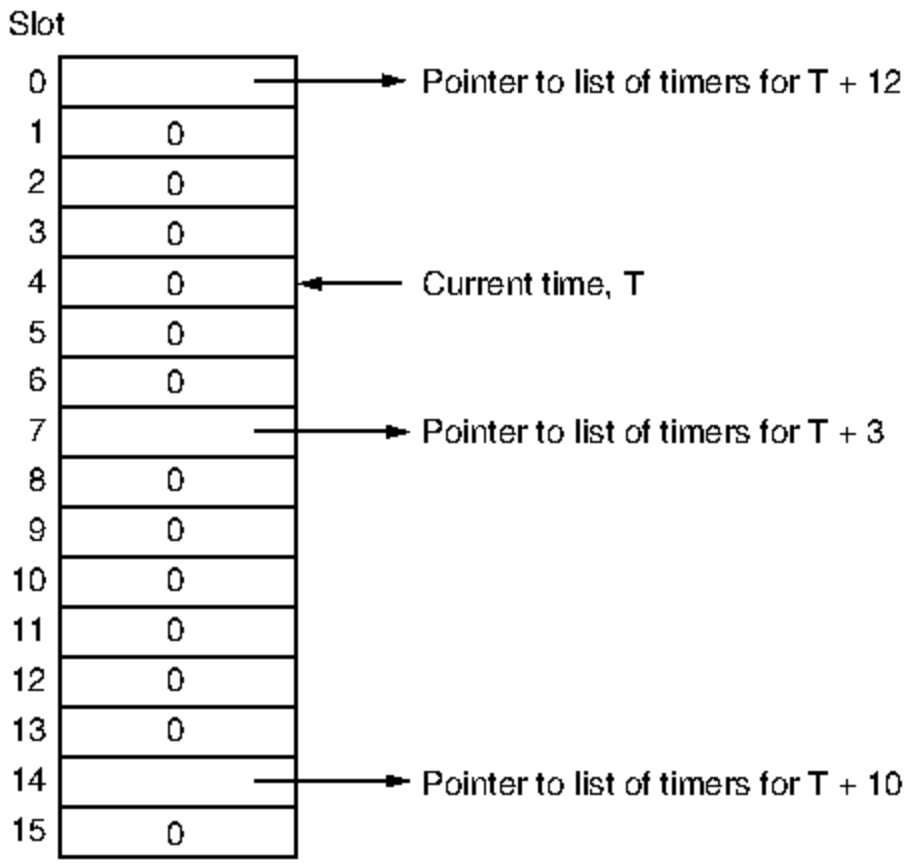


Figure 6-53. A timing wheel.

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 by Varghese and Lauck (1987).

6.6.5 Header Compression

We have been looking at fast networks for too long. There is more out there. Let us now consider performance on wireless and other networks in which bandwidth is limited. Reducing software overhead can help mobile computers run

more efficiently, but it does nothing to improve performance when the network links are the bottleneck.

To use bandwidth well, protocol headers and payloads should be carried with the minimum of bits. For payloads, this means using compact encodings of information, such as images that are in JPEG format rather than a bitmap, or document formats such as PDF that include compression. It also means application-level caching mechanisms, such as Web caches that reduce transfers in the first place.

What about for protocol headers? At the link layer, headers for wireless networks are typically compact because they were designed with scarce bandwidth in mind. For example, 802.16 headers have short connection identifiers instead of longer addresses. However, higher layer protocols such as IP, TCP and UDP come in one version for all link layers, and they are not designed with compact headers. In fact, streamlined processing to reduce software overhead often leads to headers that are not as compact as they could otherwise be (e.g., IPv6 has a more loosely packed headers than IPv4).

The higher-layer headers can be a significant performance hit. Consider, for example, voice-over-IP data that is being carried with the combination of IP, UDP, and RTP. These protocols require 40 bytes of header (20 for IPv4, 8 for UDP, and 12 for RTP). With IPv6 the situation is even worse: 60 bytes, including the 40-byte IPv6 header. The headers can wind up as the majority of the transmitted data and consume more than half the bandwidth.

Header compression is used to reduce the bandwidth taken over links by higher-layer protocol headers. Specially designed schemes are used instead of general purpose methods. This is because headers are short, so they do not compress well individually, and decompression requires all prior data to be received. This will not be the case if a packet is lost.

Header compression obtains large gains by using knowledge of the protocol format. One of the first schemes was designed by Van Jacobson (1990) for compressing TCP/IP headers over slow serial links. It is able to compress a typical TCP/IP header of 40 bytes down to an average of 3 bytes. The trick to this method is hinted at in Fig. 6-52. Many of the header fields do not change from packet to packet. There is no need, for example, to send the same IP TTL or the same TCP port numbers in each and every packet. They can be omitted on the sending side of the link and filled in on the receiving side.

Similarly, other fields change in a predictable manner. For example, barring loss, the TCP sequence number advances with the data. In these cases, the receiver can predict the likely value. The actual number only needs to be carried when it differs from what is expected. Even then, it may be carried as a small change from the previous value, as when the acknowledgement number increases when new data is received in the reverse direction.

With header compression, it is possible to have simple headers in higher-layer protocols and compact encodings over low bandwidth links. **ROHC** (**RObust Header Compression**) is a modern version of header compression that is defined

as a framework in RFC 5795. It is designed to tolerate the loss that can occur on wireless links. There is a profile for each set of protocols to be compressed, such as IP/UDP/RTP. Compressed headers are carried by referring to a context, which is essentially a connection; header fields may easily be predicted for packets of the same connection, but not for packets of different connections. In typical operation, ROHC reduces IP/UDP/RTP headers from 40 bytes to 1 to 3 bytes.

While header compression is mainly targeted at reducing bandwidth needs, it can also be useful for reducing delay. Delay is comprised of propagation delay, which is fixed given a network path, and transmission delay, which depends on the bandwidth and amount of data to be sent. For example, a 1-Mbps link sends 1 bit in 1 μsec. In the case of media over wireless networks, the network is relatively slow so transmission delay may be an important factor in overall delay and consistently low delay is important for quality of service.

Header compression can help by reducing the amount of data that is sent, and hence reducing transmission delay. The same effect can be achieved by sending smaller packets. This will trade increased software overhead for decreased transmission delay. Note that another potential source of delay is queueing delay to access the wireless link. This can also be significant because wireless links are often heavily used as the limited resource in a network. In this case, the wireless link must have quality-of-service mechanisms that give low delay to real-time packets. Header compression alone is not sufficient.

6.6.6 Protocols for Long Fat Networks

Since the 1990s, there have been gigabit networks that transmit data over large distances. Because of the combination of a fast network, or "fat pipe," and long delay, these networks are called **long fat networks**. When these networks arose, people's first reaction was to use the existing protocols on them, but various problems quickly arose. In this section, we will discuss some of the problems with scaling up the speed and delay of network protocols.

The first problem is that many protocols use 32-bit sequence numbers. When the Internet began, the lines between routers were mostly 56-kbps leased lines, so a host blasting away at full speed took over 1 week to cycle through the sequence numbers. To the TCP designers, 2^{32} was a pretty decent approximation of infinity because there was little danger of old packets still being around a week after they were transmitted. With 10-Mbps Ethernet, the wrap time became 57 minutes, much shorter, but still manageable. With a 1-Gbps Ethernet pouring data out onto the Internet, the wrap time is about 34 seconds, well under the 120-sec maximum packet lifetime on the Internet. All of a sudden, 2^{32} is not nearly as good an approximation to infinity since a fast sender can cycle through the sequence space while old packets still exist.

The problem is that many protocol designers simply assumed, without stating it, that the time required 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. Fortunately, it proved possible to extend the effective sequence number by treating the timestamp that can be carried as an option in the TCP header of each packet as the high-order bits. This mechanism is called PAWS (Protection Against Wrapped Sequence numbers) and is described in RFC 1323.

A second problem is that the size of the flow control window must be greatly increased. Consider, for example, sending a 64-KB burst of data from San Diego to Boston in order to fill the receiver's 64-KB buffer. 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. 6-54(a). Only 500 μsec later, in Fig. 6-54(b), all the segments are out on the fiber. The lead segment 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.

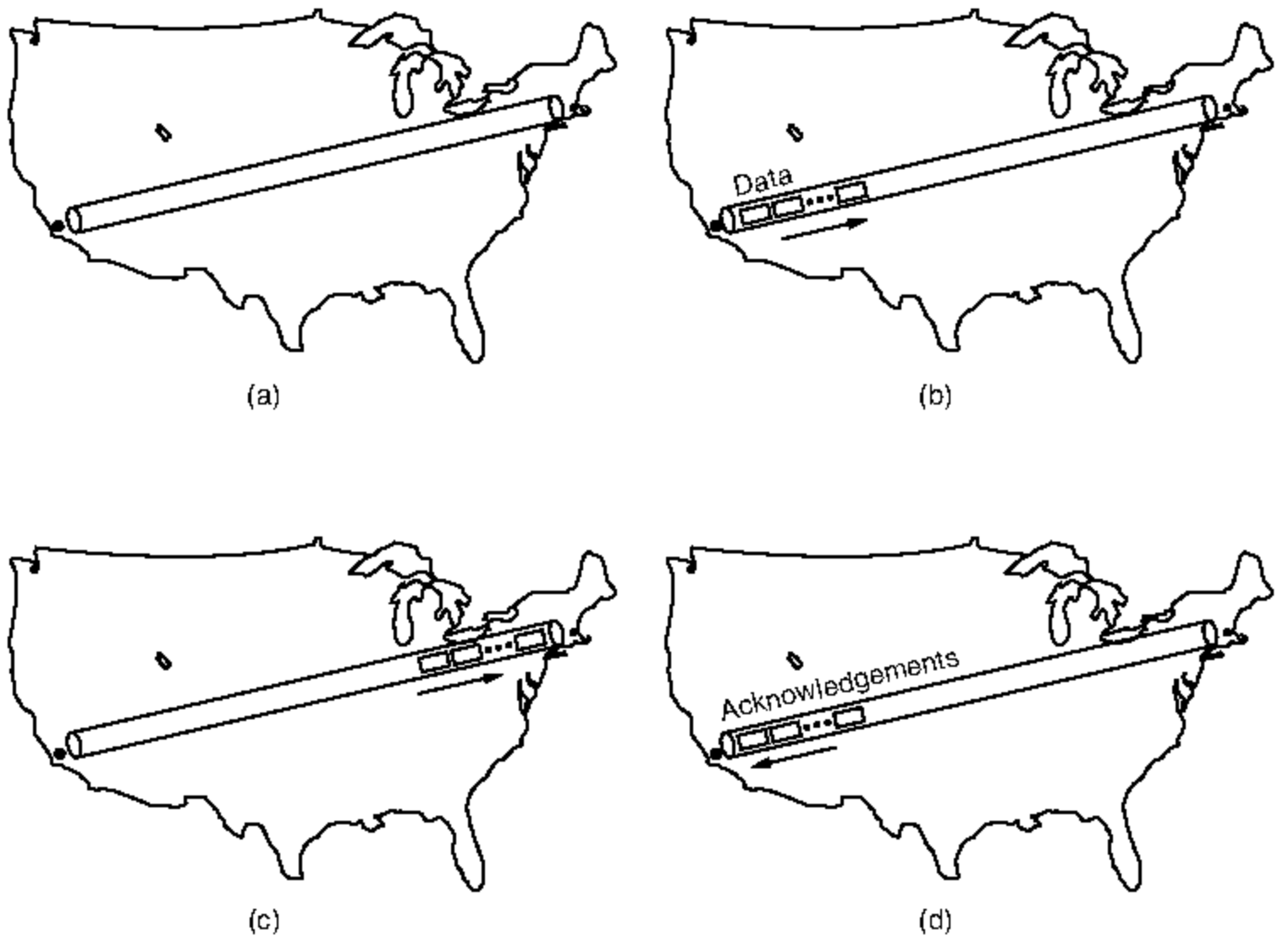


Figure 6-54. The state of transmitting 1 Mbit from San Diego to Boston. (a) At $t = 0$. (b) After 500 μsec. (c) After 20 msec. (d) After 40 msec.

After 20 msec, the lead segment hits Boston, as shown in Fig. 6-54(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 1.25 msec out of 100, the efficiency is about 1.25%. This situation is typical of an older protocols running 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. 6-54, 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 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% efficiency: it is only 1.25% of the pipe's capacity.

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

A third and related problem is that simple retransmission schemes, such as the go-back-n protocol, perform poorly on lines with a large bandwidth-delay product. Consider, the 1-Gbps transcontinental link with a round-trip transmission time of 40 msec. A sender can transmit 5 MB in one round trip. 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 MB worth of packets that came afterward. Clearly, this is a massive waste of resources. More complex protocols such as selective-repeat are needed.

A fourth problem is that gigabit lines are fundamentally different from megabit lines in that long gigabit lines are delay limited rather than bandwidth limited. In Fig. 6-55 we show the time it takes to transfer a 1-Mbit 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 6-55 has unfortunate implications for network protocols. It says that stop-and-wait protocols, such as RPC, have an inherent upper bound on their performance. This limit is dictated by the speed of light. No amount of technological progress in optics will ever improve matters (new laws of physics would help, though). Unless some other use can be found for a gigabit line while a host is waiting for a reply, the gigabit line is no better than a megabit line, just more expensive.

A fifth problem is that communication speeds have improved 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. Compare these numbers to

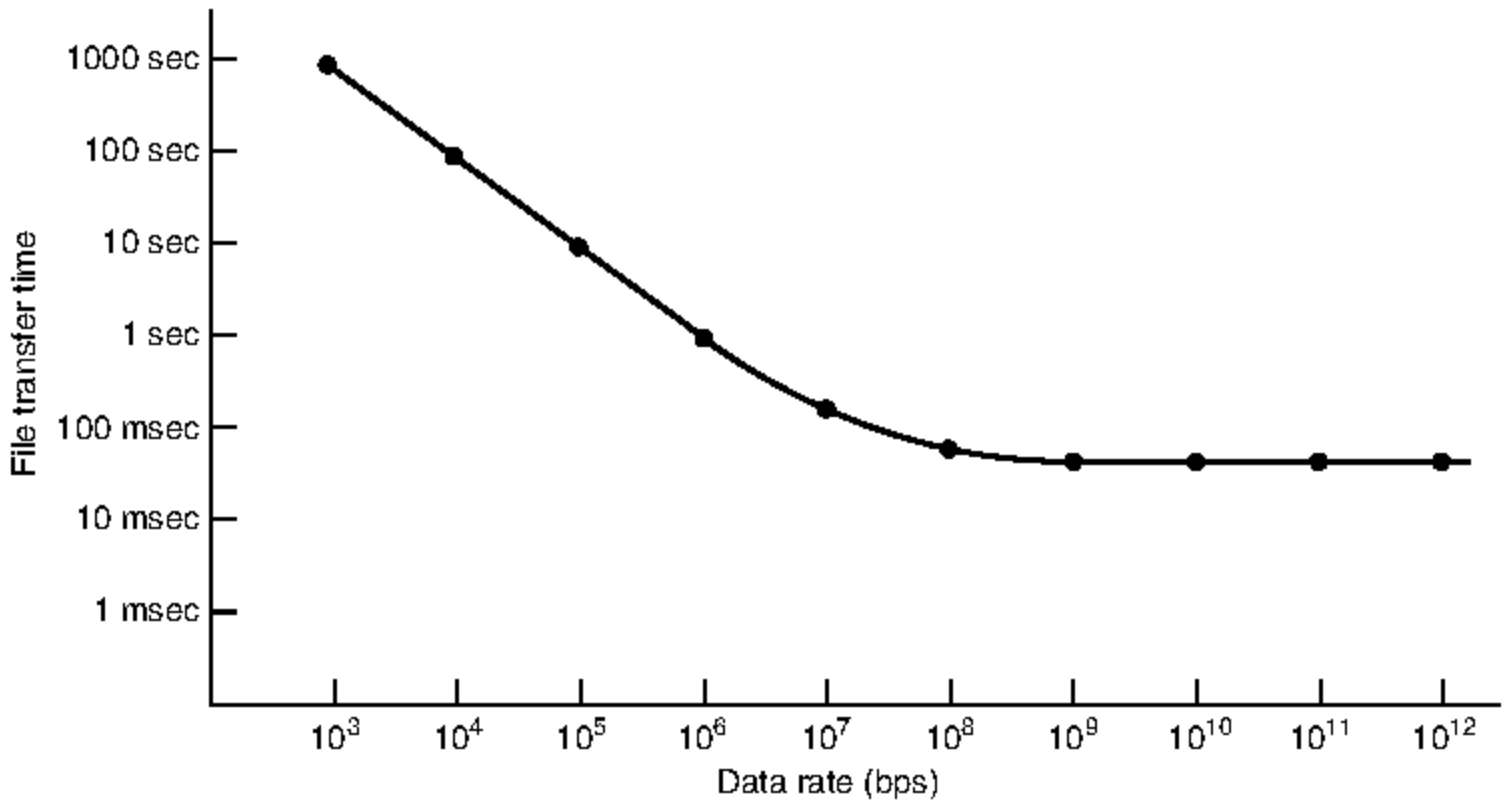


Figure 6-55. Time to transfer and acknowledge a 1-Mbit file over a 4000-km line.

1000-MIPS computers exchanging packets over a 1-Gbps line. The number of instructions per byte has decreased by more than a factor of 10. The exact numbers are debatable depending on dates and scenarios, but the conclusion is this: there is less time available for protocol processing than there used to be, so protocols must become simpler.

Let us now turn from the problems to ways of dealing with them. The basic principle that all high-speed 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. This concern is still valid for wireless networks, but not for gigabit networks. Protocol processing is the problem, so protocols should be designed to minimize it. The IPv6 designers clearly understood this principle.

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 make sure the network coprocessor is cheaper than 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 and avoid races. Usually, the best approach is to make the protocols simple and have the main CPU do the work.

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 fast 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, etc. do not occur.

The maximum data size should be large, to reduce software overhead and permit efficient operation. 1500 bytes is too small for high-speed networks, which is why gigabit Ethernet supports jumbo frames of up to 9 KB and IPv6 supports jumbogram packets in excess of 64 KB.

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 by using a sliding window protocol. Future protocols may switch to rate-based protocols to avoid the (long) delays inherent in the receiver sending window updates to the sender. 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.

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.

6.7 DELAY-TOLERANT NETWORKING

We will finish this chapter by describing a new kind of transport that may one day be an important component of the Internet. TCP and most other transport protocols are based on the assumption that the sender and the receiver are continuously connected by some working path, or else the protocol fails and data cannot be delivered. In some networks there is often no end-to-end path. An example is a space network as LEO (Low-Earth Orbit) satellites pass in and out of range of ground stations. A given satellite may be able to communicate to a ground station only at particular times, and two satellites may never be able to communicate with each other at any time, even via a ground station, because one of the satellites

may always be out of range. Other example networks involve submarines, buses, mobile phones, and other devices with computers for which there is intermittent connectivity due to mobility or extreme conditions.

In these occasionally connected networks, data can still be communicated by storing them at nodes and forwarding them later when there is a working link. This technique is called **message switching**. Eventually the data will be relayed to the destination. A network whose architecture is based on this approach is called a **DTN** (**Delay-Tolerant Network**, or a **Disruption-Tolerant Network**).

Work on DTNs started in 2002 when IETF set up a research group on the topic. The inspiration for DTNs came from an unlikely source: efforts to send packets in space. Space networks must deal with intermittent communication and very long delays. Kevin Fall observed that the ideas for these Interplanetary Internets could be applied to networks on Earth in which intermittent connectivity was the norm (Fall, 2003). This model gives a useful generalization of the Internet in which storage and delays can occur during communication. Data delivery is akin to delivery in the postal system, or electronic mail, rather than packet switching at routers.

Since 2002, the DTN architecture has been refined, and the applications of the DTN model have grown. As a mainstream application, consider large datasets of many terabytes that are produced by scientific experiments, media events, or Web-based services and need to be copied to datacenters at different locations around the world. Operators would like to send this bulk traffic at off-peak times to make use of bandwidth that has already been paid for but is not being used, and are willing to tolerate some delay. It is like doing the backups at night when other applications are not making heavy use of the network. The problem is that, for global services, the off-peak times are different at locations around the world. There may be little overlap in the times when datacenters in Boston and Perth have off-peak network bandwidth because night for one city is day for the other.

However, DTN models allow for storage and delays during transfer. With this model, it becomes possible to send the dataset from Boston to Amsterdam using off-peak bandwidth, as the cities have time zones that are only 6 hours apart. The dataset is then stored in Amsterdam until there is off-peak bandwidth between Amsterdam and Perth. It is then sent to Perth to complete the transfer. Laoutaris et al. (2009) have studied this model and find that it can provide substantial capacity at little cost, and that the use of a DTN model often doubles that capacity compared with a traditional end-to-end model.

In what follows, we will describe the IETF DTN architecture and protocols.

6.7.1 DTN Architecture

The main assumption in the Internet that DTNs seek to relax is that an end-to-end path between a source and a destination exists for the entire duration of a communication session. When this is not the case, the normal Internet protocols

fail. DTNs get around the lack of end-to-end connectivity with an architecture that is based on message switching, as shown in Fig. 6-56. It is also intended to tolerate links with low reliability and large delays. The architecture is specified in RFC 4838.

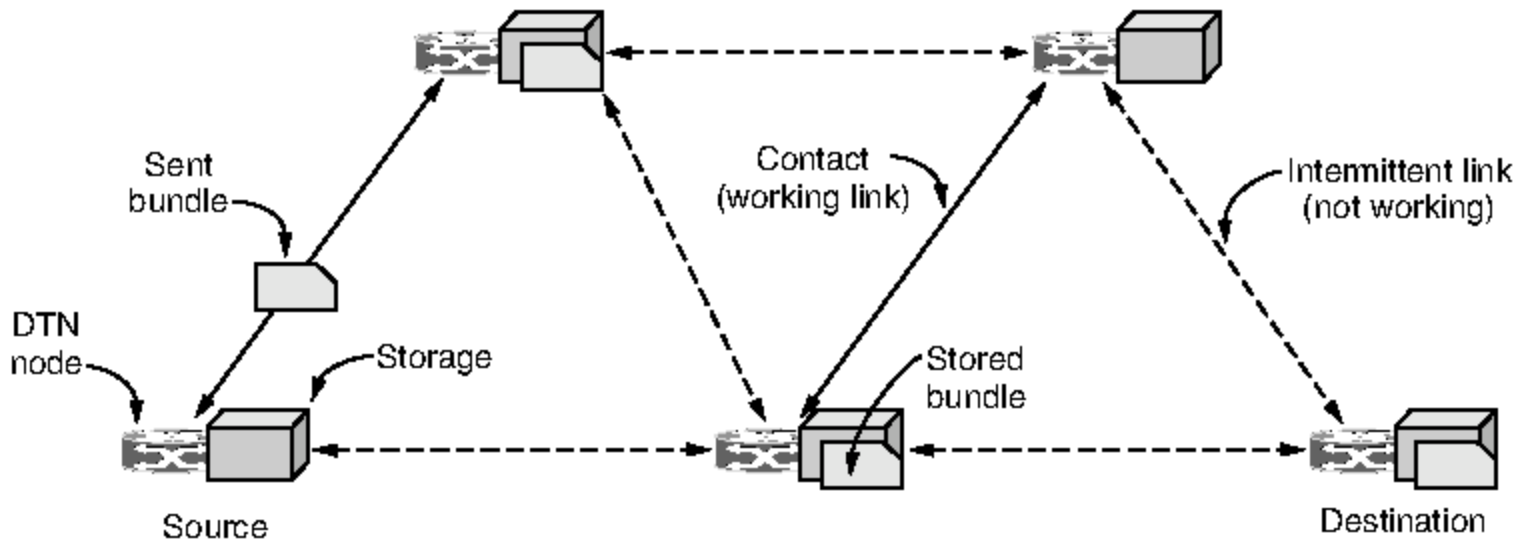


Figure 6-56. Delay-tolerant networking architecture.

In DTN terminology, a message is called a **bundle**. DTN nodes are equipped with storage, typically persistent storage such as a disk or flash memory. They store bundles until links become available and then forward the bundles. The links work intermittently. Fig. 6-56 shows five intermittent links that are not currently working, and two links that are working. A working link is called a **contact**. Fig. 6-56 also shows bundles stored at two DTN nodes awaiting contacts to send the bundles onward. In this way, the bundles are relayed via contacts from the source to their destination.

The storing and forwarding of bundles at DTN nodes sounds similar to the queueing and forwarding of packets at routers, but there are qualitative differences. In routers in the Internet, queueing occurs for milliseconds or at most seconds. At DTN nodes, bundles may be stored for hours, until a bus arrives in town, while an airplane completes a flight, until a sensor node harvests enough solar energy to run, until a sleeping computer wakes up, and so forth. These examples also point to a second difference, which is that nodes may move (with a bus or plane) while they hold stored data, and this movement may even be a key part of data delivery. Routers in the Internet are not allowed to move. The whole process of moving bundles might be better known as "store-carry-forward."

As an example, consider the scenario shown in Fig. 6-57 that was the first use of DTN protocols in space (Wood et al., 2008). The source of bundles is an LEO satellite that is recording Earth images as part of the Disaster Monitoring Constellation of satellites. The images must be returned to the collection point. However, the satellite has only intermittent contact with three ground stations as it orbits the Earth. It comes into contact with each ground station in turn. Each of the satellite, ground stations, and collection point act as a DTN node. At each contact, a

bundle (or a portion of a bundle) is sent to a ground station. The bundles are then sent over a backhaul terrestrial network to the collection point to complete the transfer.

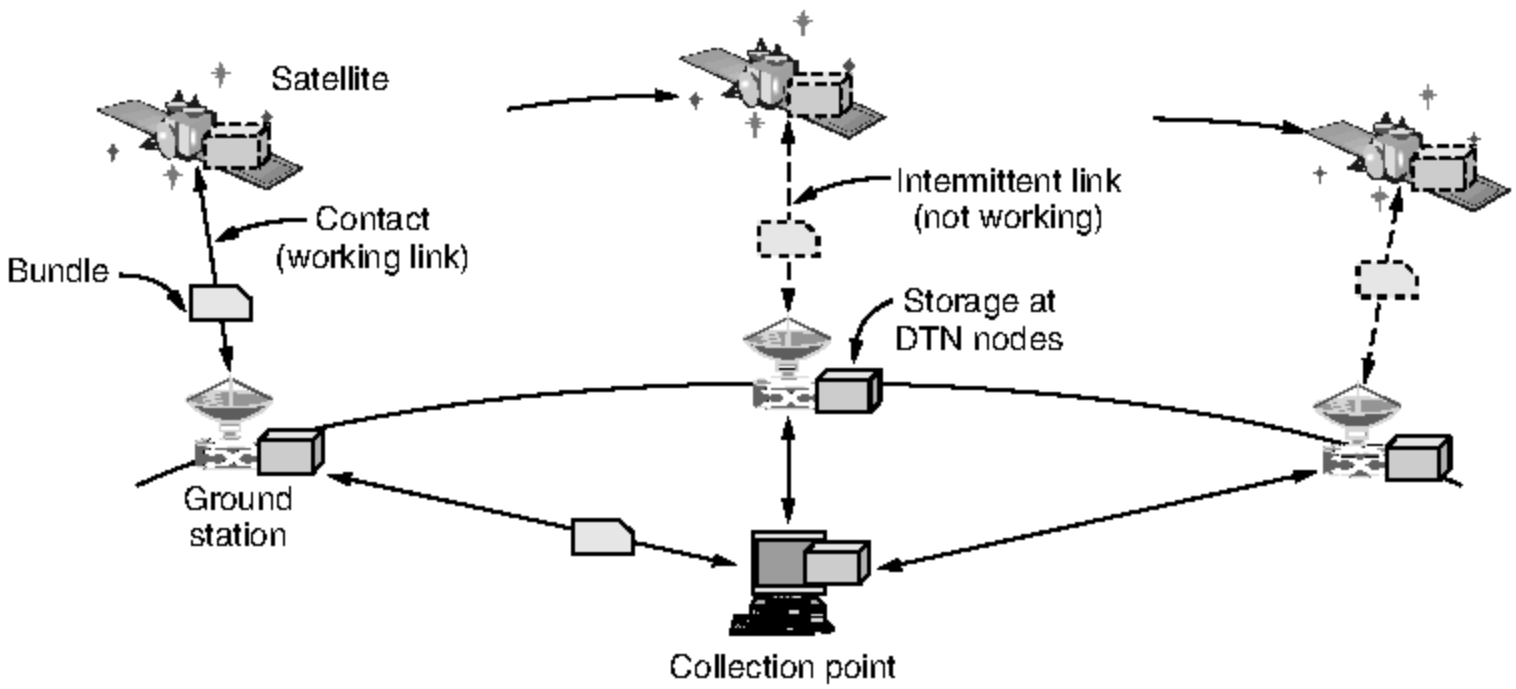


Figure 6-57. Use of a DTN in space.

The primary advantage of the DTN architecture in this example is that it naturally fits the situation of the satellite needing to store images because there is no connectivity at the time the image is taken. There are two further advantages. First, there may be no single contact long enough to send the images. However, they can be spread across the contacts with three ground stations. Second, the use of the link between the satellite and ground station is decoupled from the link over the backhaul network. This means that the satellite download is not limited by a slow terrestrial link. It can proceed at full speed, with the bundle stored at the ground station until it can be relayed to the collection point.

An important issue that is not specified by the architecture is how to find good routes via DTN nodes. A route in this path to use. Good routes depend on the nature of the architecture describes when to send data, and also which contacts. Some contacts are known ahead of time. A good example is the motion of heavenly bodies in the space example. For the space experiment, it was known ahead of time when contacts would occur, that the contact intervals ranged from 5 to 14 minutes per pass with each ground station, and that the downlink capacity was 8.134 Mbps. Given this knowledge, the transport of a bundle of images can be planned ahead of time.

In other cases, the contacts can be predicted, but with less certainty. Examples include buses that make contact with each other in mostly regular ways, due to a timetable, yet with some variation, and the times and amount of off-peak bandwidth in ISP networks, which are predicted from past data. At the other extreme, the contacts are occasional and random. One example is carrying data from user

to user on mobile phones depending on which users make contact with each other during the day. When there is unpredictability in contacts, one routing strategy is to send copies of the bundle along different paths in the hope that one of the copies is delivered to the destination before the lifetime is reached.

6.7.2 The Bundle Protocol

To take a closer look at the operation of DTNs, we will now look at the IETF protocols. DTNs are an emerging kind of network, and experimental DTNs have used different protocols, as there is no requirement that the IETF protocols be used. However, they are at least a good place to start and highlight many of the key issues.

The DTN protocol stack is shown in Fig. 6-58. The key protocol is the **Bundle protocol**, which is specified in RFC 5050. It is responsible for accepting messages from the application and sending them as one or more bundles via store-carry-forward operations to the destination DTN node. It is also apparent from Fig. 6-58 that the Bundle protocol runs above the level of TCP/IP. In other words, TCP/IP may be used over each contact to move bundles between DTN nodes. This positioning raises the issue of whether the Bundle protocol is a transport layer protocol or an application layer protocol. Just as with RTP, we take the position that, despite running over a transport protocol, the Bundle protocol is providing a transport service to many different applications, and so we cover DTNs in this chapter.

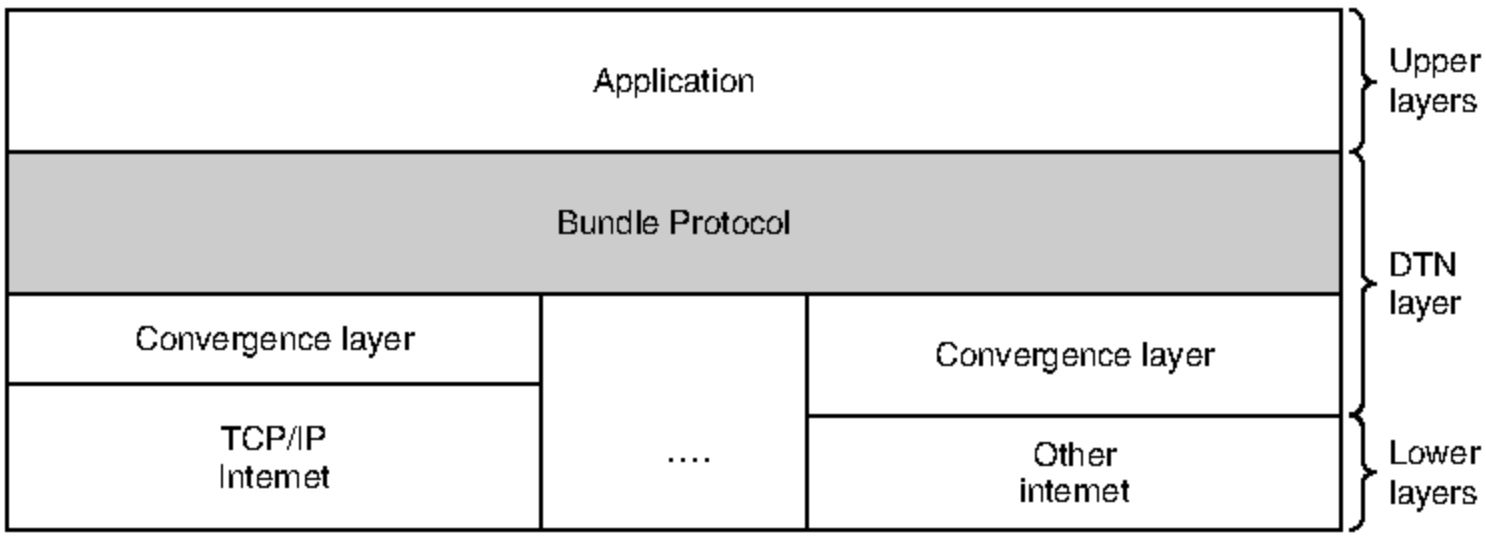


Figure 6-58. Delay-tolerant networking protocol stack.

In Fig. 6-58, we see that the Bundle protocol may be run over other kinds of protocols such as UDP, or even other kinds of internets. For example, in a space network the links may have very long delays. The round-trip time between Earth and Mars can easily be 20 minutes depending on the relative position of the planets. Imagine how well TCP acknowledgements and retransmissions will work over that link, especially for relatively short messages. Not well at all. Instead,

another protocol that uses error-correcting codes might be used. Or in sensor networks that are very resource constrained, a more lightweight protocol than TCP may be used.

Since the Bundle protocol is fixed, yet it is intended to run over a variety of transports, there is must be a gap in functionality between the protocols. That gap is the reason for the inclusion of a convergence layer in Fig. 6-58. The convergence layer is just a glue layer that matches the interfaces of the protocols that it joins. By definition there is a different convergence layer for each different lower layer transport. Convergence layers are commonly found in standards to join new and existing protocols.

The format of Bundle protocol messages is shown in Fig. 6-59. The different fields in these messages tell us some of the key issues that are handled by the Bundle protocol.

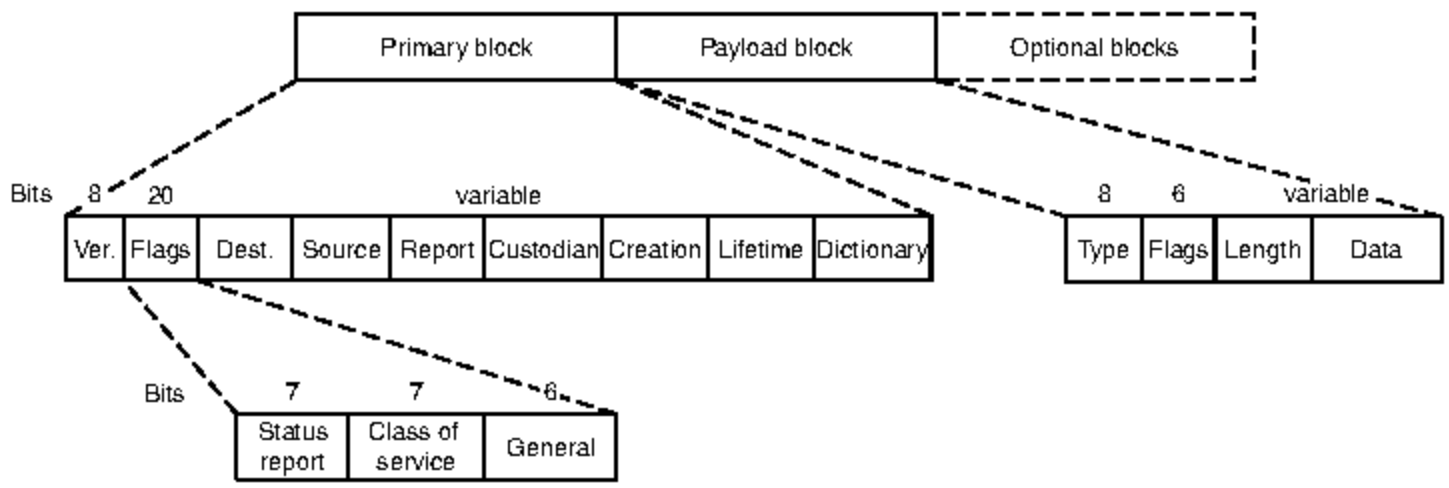


Figure 6-59. Bundle protocol message format.

Each message consists of a primary block, which can be thought of as a header, a payload block for the data, and optionally other blocks, for example to carry security parameters. The primary block begins with a *Version* field (currently 6) followed by a *Flags* field. Among other functions, the flags encode a class of service to let a source mark its bundles as higher or lower priority, and other handling requests such as whether the destination should acknowledge the bundle.

Then come addresses, which highlight three interesting parts of the design. As well as a *Destination* and *Source* identifier field, there is a *Custodian* identifier. The custodian is the party responsible for seeing that the bundle is delivered. In the Internet, the source node is usually the custodian, as it is the node that retransmits if the data is not ultimately delivered to the destination. However, in a DTN, the source node may not always be connected and may have no way of knowing whether the data has been delivered. DTNs deal with this problem using the notion of **custody transfer**, in which another node, closer to the destination, can assume responsibility for seeing the data safely delivered. For example, if a bundle is stored on an airplane for forwarding at a later time and location, the airplane may become the custodian of the bundle.

The second interesting aspect is that these identifiers are *not* IP addresses. Because the Bundle protocol is intended to work across a variety of transports and internets, it defines its own identifiers. These identifiers are really more like high-level names, such as Web page URLs, than low-level addresses, such as IP addresses. They give DTNs an aspect of application-level routing, such as email delivery or the distribution of software updates.

The third interesting aspect is the way the identifiers are encoded. There is also a *Report* identifier for diagnostic messages. All of the identifiers are encoded as references to a variable length *Dictionary* field. This provides compression when the custodian or report nodes are the same as the source or the destination. In fact, much of the message format has been designed with both extensibility and efficiency in mind by using a compact representation of variable length fields. The compact representation is important for wireless links and resource-constrained nodes such as in a sensor network.

Next comes a *Creation* field carrying the time at which the bundle was created, along with a sequence number from the source for ordering, plus a *Lifetime* field that tells the time at which the bundle data is no longer useful. These fields exist because data may be stored for a long period at DTN nodes and there must be some way to remove stale data from the network. Unlike the Internet, they require that DTN nodes have loosely synchronized clocks.

The primary block is completed with the *Dictionary* field. Then comes the payload block. This block starts with a short *Type* field that identifies it as a payload, followed by a small set of *Flags* that describe processing options. Then comes the *Data* field, preceded by a *Length* field. Finally, there may be other, optional blocks, such as a block that carries security parameters.

Many aspects of DTNs are being explored in the research community. Good strategies for routing depend on the nature of the contacts, as was mentioned above. Storing data inside the network raises other issues. Now congestion control must consider storage at nodes as another kind of resource that can be depleted. The lack of end-to-end communication also exacerbates security problems. Before a DTN node takes custody of a bundle, it may want to know that the sender is authorized to use the network and that the bundle is probably wanted by the destination. Solutions to these problems will depend on the kind of DTN, as space networks are different from sensor networks.

6.8 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. A common transport layer interface is the one provided by Berkeley sockets.

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. It must handle all the service primitives, manage connections and timers, allocate bandwidth with congestion control, and run a variable-sized sliding window for flow control.

Congestion control should allocate all of the available bandwidth between competing flows fairly, and it should track changes in the usage of the network. The AIMD control law converges to a fair and efficient allocation.

The Internet has two main transport protocols: UDP and TCP. UDP is a connectionless protocol that is mainly a wrapper for IP packets with the additional feature of multiplexing and demultiplexing multiple processes using a single IP address. UDP can be used for client-server interactions, for example, using RPC. It can also be used for building real-time protocols such as RTP.

The main Internet transport protocol is TCP. It provides a reliable, bidirectional, congestion-controlled byte stream with a 20-byte header on all segments. A great deal of work has gone into optimizing TCP performance, using algorithms from Nagle, Clark, Jacobson, Karn, and others.

Network performance is typically dominated by protocol and segment processing overhead, and this situation gets worse at higher speeds. Protocols should be designed to minimize the number of segments and work for large bandwidth-delay paths. For gigabit networks, simple protocols and streamlined processing are called for.

Delay-tolerant networking provides a delivery service across networks that have occasional connectivity or long delays across links. Intermediate nodes store, carry, and forward bundles of information so that it is eventually delivered, even if there is no working path from sender to receiver at any time.

PROBLEMS

1. In our example transport primitives of Fig. 6-2, 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. Primitives of transport service assume asymmetry between the two end points during connection establishment, one end (server) executes LISTEN while the other end (client) executes CONNECT. However, in peer to peer applications such file sharing

systems, e.g. BitTorrent, all end points are peers. There is no server or client functionality. How can transport service primitives may be used to build such peer to peer applications?

3. In the underlying model of Fig. 6-4, 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. 6-4?

4. In both parts of Fig. 6-6, there is a comment that the value of *SERVER_PORT* must be the same in both client and server. Why is this so important?

5. In the Internet File Server example (Figure 6-6), can the connect( ) system call on the client fail for any reason other than listen queue being full on the server? Assume that the network is perfect.

6. One criteria for deciding whether to have a server active all the time or have it start on demand using a process server is how frequently the service provided is used. Can you think of any other criteria for making this decision?

7. 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?

8. 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?

9. 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.

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

11. 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?

12. In Figure 6-20, suppose a new flow E is added that takes a path from $R1$ to $R2$ to $R6$. How does the max-min bandwidth allocation change for the five flows?

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

14. Some other policies for fairness in congestion control are Additive Increase Additive Decrease (AIAD), Multiplicative Increase Additive Decrease (MIAD), and Multiplicative Increase Multiplicative Decrease (MIMD). Discuss these three policies in terms of convergence and stability.

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

16. Consider a simple application-level protocol built on top of UDP that allows a client to retrieve a file from a remote server residing at a well-known address. The client first sends a request with a file name, and the server responds with a sequence of data packets containing different parts of the requested file. To ensure reliability and sequenced delivery, client and server use a stop-and-wait protocol. Ignoring the obvious performance issue, do you see a problem with this protocol? Think carefully about the possibility of processes crashing.

17. 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?

18. 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?

19. Both UDP and TCP use port numbers to identify the destination entity when delivering a message. Give two reasons why these protocols invented a new abstract ID (port numbers), instead of using process IDs, which already existed when these protocols were designed.

20. Several RPC implementations provide an option to the client to use RPC implemented over UDP or RPC implemented over TCP. Under what conditions will a client prefer to use RPC over UDP and under what conditions will he prefer to use RPC over TCP?

21. Consider two networks, $N1$ and $N2$, that have the same average delay between a source A and a destination D. In $N1$, the delay experienced by different packets is unformly distributed with maximum delay being 10 seconds, while in $N2$, 99% of the packets experience less than one second delay with no limit on maximum delay. Discuss how RTP may be used in these two cases to transmit live audio/video stream.

22. What is the total size of the minimum TCP MTU, including TCP and IP overhead but not including data link layer overhead?

23. 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?

24. RTP is used to transmit CD-quality audio, which makes a pair of 16-bit samples 44,100 times/sec, one sample for each of the stereo channels. How many packets per second must RTP transmit?

25. Would it be possible to place the RTP code in the operating system kernel, along with the UDP code? Explain your answer.

26. 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?

27. In Fig. 6-36 we saw that in addition to the 32-bit *acknowledgement* field, there is an *ACK* bit in the fourth word. Does this really add anything? Why or why not?

28. The maximum payload of a TCP segment is 65,495 bytes. Why was such a strange number chosen?

29. Describe two ways to get into the *SYN RCVD* state of Fig. 6-39.

30. 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?

31. Suppose that the TCP congestion window is set to 18 KB 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.

32. 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 using the Jacobson algorithm? Use $\alpha = 0.9$.

33. A TCP machine is sending full 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?

34. What is the fastest line speed at which a host can blast out 1500-byte TCP payloads with a 120-sec maximum packet lifetime without having the sequence numbers wrap around? Take TCP, IP, and Ethernet overhead into consideration. Assume that Ethernet frames may be sent continuously.

35. To address the limitations of IP version 4, a major effort had to be undertaken via IETF that resulted in the design of IP version 6 and there are still is significant reluctance in the adoption of this new version. However, no such major effort is needed to address the limitations of TCP. Explain why this is the case.

36. In a network whose max segment is 128 bytes, max segment lifetime is 30 sec, and has 8-bit sequence numbers, what is the maximum data rate per connection?

37. Suppose that you are measuring the time to receive a segment. When an interrupt occurs, you read out the system clock in milliseconds. When the segment 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 segment?

38. A CPU executes instructions at the rate of 1000 MIPS. Data can be copied 64 bits at a time, with each word copied costing 10 instructions. If an coming packet has to be copied four times, 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 1000-MIPS rate.

39. 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.

40. In Sec. 6.6.5, we calculated that a gigabit line dumps 80,000 packets/sec on the host, giving it only 6250 instructions to process it and leaving half the CPU time for applications. This calculation assumed a 1500-byte packet. Redo the calculation for an ARPANET-sized packet (128 bytes). In both cases, assume that the packet sizes given include all overhead.

41. 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?

42. Calculate the bandwidth-delay product for the following networks: (1) T1 (1.5 Mbps), (2) Ethernet (10 Mbps), (3) T3 (45 Mbps), and (4) STS-3 (155 Mbps). Assume an RTT of 100 msec. Recall that a TCP header has 16 bits reserved for Window Size. What are its implications in light of your calculations?

43. What is the bandwidth-delay product for a 50-Mbps channel on a geostationary satellite? If the packets are all 1500 bytes (including overhead), how big should the window be in packets?

44. The file server of Fig. 6-6 is far from perfect and could use a few improvements. Make the following modifications.

(a) Give the client a third argument that specifies a byte range.
(b) Add a client flag –w that allows the file to be written to the server.

45. One common function that all network protocols need is to manipulate messages. Recall that protocols manipulate messages by adding/striping headers. Some protocols may break a single message into multiple fragments, and later join these multiple fragments back into a single message. To this end, design and implement a message management library that provides support for creating a new message, attaching a header to a message, stripping a header from a message, breaking a message into two messages, combining two messages into a single message, and saving a copy of a message. Your implementation must minimize data copying from one buffer to another as much as possible. It is critical that the operations that manipulate messages do not touch the data in a message, but rather, only manipulate pointers.

46. Design and implement a chat system that allows multiple groups of users to chat. A chat coordinator resides at a well-known network address, uses UDP for communication with chat clients, sets up chat servers for each chat session, and maintains a chat session directory. There is one chat server per chat session. A chat server uses TCP for communication with clients. A chat client allows users to start, join, and leave a chat session. Design and implement the coordinator, server, and client code.

7

THE APPLICATION LAYER

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

However, even in the application layer there is a need for support protocols, to allow the applications to function. Accordingly, we will look at an important one of these before starting with the applications themselves. The item in question is DNS, which handles naming within the Internet. After that, we will examine three real applications: electronic mail, the World Wide Web, and multimedia. We will finish the chapter by saying more about content distribution, including by peer-to-peer networks.

7.1 DNS—THE DOMAIN NAME SYSTEM

Although programs theoretically could refer to Web pages, mailboxes, and other resources by using the network (e.g., IP) addresses of the computers on which they are stored, these addresses are hard for people to remember. Also, browsing a company's Web pages from *128.111.24.41* means that if the company moves the Web server to a different machine with a different IP address, everyone needs to be told the new IP address. Consequently, high-level, readable names were introduced in order to decouple machine names from machine addresses. In

this way, the company's Web server might be known as *www.cs.washington.edu* regardless of its IP address. Nevertheless, since the network itself understands only numerical addresses, some mechanism is required to convert the names to network addresses. In the following sections, we will study how this mapping is accomplished in the Internet.

Way back in the ARPANET days, there was simply a file, *hosts.txt*, that listed all the computer names 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, well before many millions of PCs were connected to the Internet, everyone involved with it 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 importantly, host name conflicts would occur constantly unless names were centrally managed, something unthinkable in a huge international network due to the load and latency. To solve these problems, **DNS** (**Domain Name System**) was invented in 1983. It has been a key part of the Internet ever since.

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 to IP addresses but can also be used for other purposes. DNS is defined in RFCs 1034, 1035, 2181, and further elaborated in many others.

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. We saw an example of a resolver, *gethostbyname*, in Fig. 6-6. The resolver sends a query containing the name to a local DNS server, which looks up the name and returns a response containing the IP address to the resolver, which then returns it to the caller. The query and response messages are sent as UDP packets. Armed with the IP address, the program can then establish a TCP connection with the host or send it UDP packets.

7.1.1 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, street address, and name of the addressee. Using this kind of hierarchical addressing ensures that 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.

For the Internet, the top of the naming hierarchy is managed by an organization called **ICANN** (**Internet Corporation for Assigned Names and Numbers**). ICANN was created for this purpose in 1998, as part of the maturing of the Internet to a worldwide, economic concern. Conceptually, the Internet is divided into

over 250 **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. 7-1. 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 contain thousands of hosts.

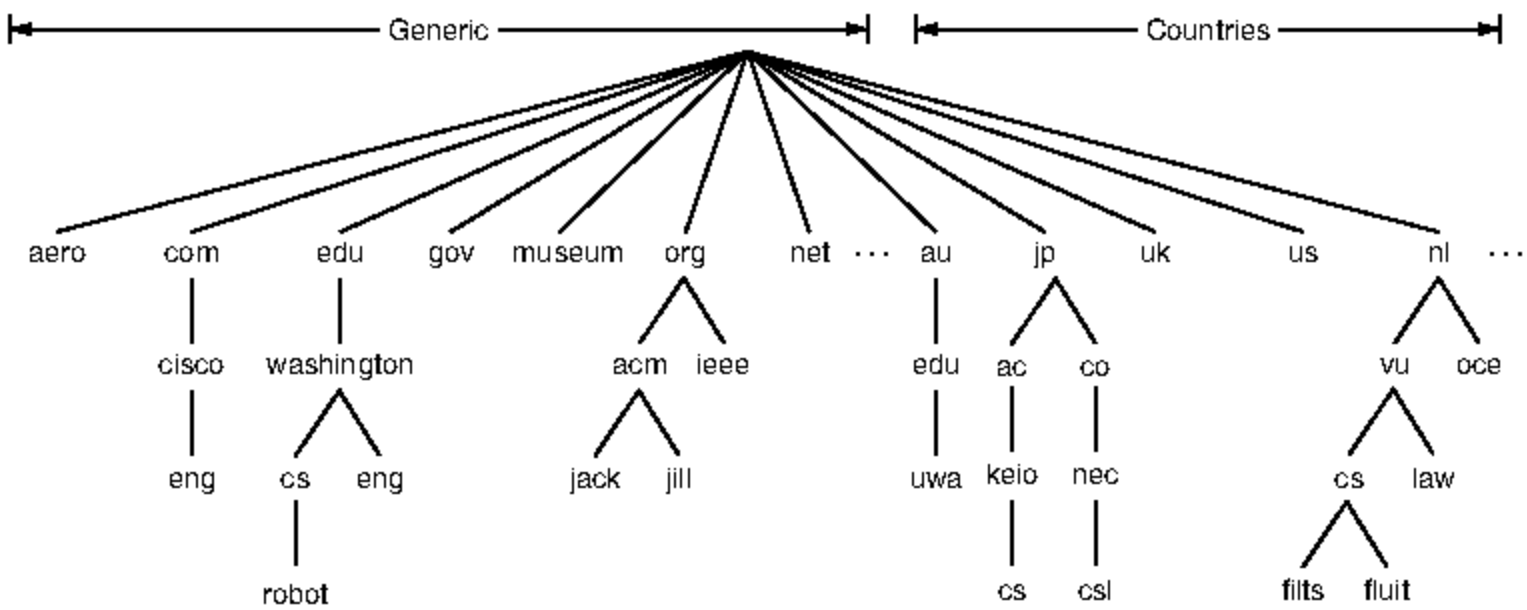


Figure 7-1. A portion of the Internet domain name space.

The top-level domains come in two flavors: generic and countries. The generic domains, listed in Fig. 7-2, include original domains from the 1980s and domains introduced via applications to ICANN. Other generic top-level domains will be added in the future.

The country domains include one entry for every country, as defined in ISO 3166. Internationalized country domain names that use non-Latin alphabets were introduced in 2010. These domains let people name hosts in Arabic, Cyrillic, Chinese, or other languages.

Getting a second-level domain, such as *name-of-company.com*, is easy. The top-level domains are run by **registrars** appointed by ICANN. Getting a name merely requires going to a corresponding registrar (for *com* in this case) to check if the desired name is available and not somebody else's trademark. If there are no problems, the requester pays the registrar a small annual fee and gets the name.

However, as the Internet has become more commercial and more international, it has also become more contentious, especially in matters related to naming. This controversy includes ICANN itself. For example, the creation of the *xxx* domain took several years and court cases to resolve. Is voluntarily placing adult content in its own domain a good or a bad thing? (Some people did not want adult content available at all on the Internet while others wanted to put it all in one domain so nanny filters could easily find and block it from children). Some of the domains self-organize, while others have restrictions on who can obtain a name, as noted in Fig. 7-2. But what restrictions are appropriate? Take the *pro* domain,

Domain	Intended use	Start date	Restricted?
com	Commercial	1985	No
edu	Educational institutions	1985	Yes
gov	Government	1985	Yes
int	International organizations	1988	Yes
mil	Military	1985	Yes
net	Network providers	1985	No
org	Non-profit organizations	1985	No
aero	Air transport	2001	Yes
biz	Businesses	2001	No
coop	Cooperatives	2001	Yes
info	Informational	2002	No
museum	Museums	2002	Yes
name	People	2002	No
pro	Professionals	2002	Yes
cat	Catalan	2005	Yes
jobs	Employment	2005	Yes
mobi	Mobile devices	2005	Yes
tel	Contact details	2005	Yes
travel	Travel industry	2005	Yes
xxx	Sex industry	2010	No

Figure 7-2. Generic top-level domains.

for example. It is for qualified professionals. But who is a professional? Doctors and lawyers clearly are professionals. But what about freelance photographers, piano teachers, magicians, plumbers, barbers, exterminators, tattoo artists, mercenaries, and prostitutes? Are these occupations eligible? According to whom?

There is also money in names. Tuvalu (the country) sold a lease on its *tv* domain for $50 million, all because the country code is well-suited to advertising television sites. Virtually every common (English) word has been taken in the *com* domain, along with the most common misspellings. Try household articles, animals, plants, body parts, etc. The practice of registering a domain only to turn around and sell it off to an interested party at a much higher price even has a name. It is called **cybersquatting**. Many companies that were slow off the mark when the Internet era began found their obvious domain names already taken when they tried to acquire them. In general, as long as no trademarks are being violated and no fraud is involved, it is first-come, first-served with names. Nevertheless, policies to resolve naming disputes are still being refined.

Each domain is named by the path upward from it to the (unnamed) root. The components are separated by periods (pronounced "dot"). Thus, the engineering department at Cisco might be *eng.cisco.com.*, rather than a UNIX-style name such as */com/cisco/eng*. Notice that this hierarchical naming means that *eng.cisco.com.* does not conflict with a potential use of *eng* in *eng.washington.edu.*, which might be used by the English department at the University of Washington.

Domain names can be either absolute or relative. An absolute domain name always ends with a period (e.g., *eng.cisco.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*, *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 either generic or country domains. For example, *cs.washington.edu* could equally well be listed under the *us* country domain as *cs.washington.wa.us*. In practice, however, most organizations in the United States are under generic domains, and most outside the United States are under the domain of their country. There is no rule against registering under multiple top-level domains. Large companies often do so (e.g., *sony.com*, *sony.net*, and *sony.nl*).

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.washington.edu* (University of Washington, in the U.S.).
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 the University of Washington and wants to be known as *vlsi.cs.washington.edu*, it has to get permission from whoever manages *cs.washington.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* (if that is still available). 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, the hosts in both buildings will normally belong to the same domain.

7.1.2 Domain 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. These records are the DNS database. 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 primary 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  Class  Type  Value
```

The *Domain_name* tells the domain to which this record applies. Normally, many records exist for each domain and each copy of the database holds information about multiple domains. This field is thus the primary search key used to satisfy queries. The order of the records in the database is not significant.

The *Time_to_live* field gives an indication of how stable the record is. Information that is highly stable is assigned a large value, such as 86400 (the number of seconds in 1 day). Information that is highly volatile is assigned a small value, such as 60 (1 minute). We will come back to this point later when we have discussed caching.

The third field of every resource record is the *Class*. For Internet information, it is always *IN*. For non-Internet information, other codes can be used, but in practice these are rarely seen.

The *Type* field tells what kind of record this is. There are many kinds of DNS records. The important types are listed in Fig. 7-3.

An *SOA* record provides the name of the primary source of information about the name server's zone (described below), the email address of its administrator, a unique serial number, and various flags and timeouts.

The most important record type is the *A* (Address) record. It holds a 32-bit IPv4 address of an interface for some host. The corresponding *AAAA*, or "quad A," record holds a 128-bit IPv6 address. Every Internet host must have at least one IP address so that other machines can communicate with it. Some hosts have two or more network interfaces, in which case they will have two or more type *A* or *AAAA* resource records. Consequently, DNS can return multiple addresses for a single name.

A common record type is the *MX* record. It specifies the name of the host prepared to accept email for the specified domain. It is used because not every

Type	Meaning	Value
SOA	Start of authority	Parameters for this zone
A	IPv4 address of a host	32-Bit integer
AAAA	IPv6 address of a host	128-Bit integer
MX	Mail exchange	Priority, domain willing to accept email
NS	Name server	Name of a server for this domain
CNAME	Canonical name	Domain name
PTR	Pointer	Alias for an IP address
SPF	Sender policy framework	Text encoding of mail sending policy
SRV	Service	Host that provides it
TXT	Text	Descriptive ASCII text

Figure 7-3. The principal DNS resource record types.

machine is prepared to accept email. If someone wants to send email to, for example, *bill@microsoft.com*, the sending host needs to find some mail server located at *microsoft.com* that is willing to accept email. The *MX* record can provide this information.

Another important record type is the *NS* record. It specifies a name server for the domain or subdomain. This is a host that has a copy of the database for a domain. It is used as part of the process to look up names, which we will describe shortly.

CNAME records allow aliases to be created. For example, a person familiar with Internet naming in general and wanting to send a message to user *paul* in the computer science department at M.I.T. might guess that *paul@cs.mit.edu* will work. Actually, this address will not work, because the domain for M.I.T.'s computer science department is *csail.mit.edu*. However, as a service to people who do not know this, M.I.T. could create a *CNAME* entry to point people and programs in the right direction. An entry like this one might do the job:

```
cs.mit.edu   86400   IN   CNAME   csail.mit.edu
```

Like *CNAME*, *PTR* points to another name. However, unlike *CNAME*, which is really just a macro definition (i.e., a mechanism to replace one string by another), *PTR* is a regular DNS data type whose interpretation depends on the context in which it is found. In practice, it is nearly always used to associate a name with an IP address to allow lookups of the IP address and return the name of the corresponding machine. These are called **reverse lookups**.

SRV is a newer type of record that allows a host to be identified for a given service in a domain. For example, the Web server for *cs.washington.edu* could be identified as *cockatoo.cs.washington.edu*. This record generalizes the *MX* record that performs the same task but it is just for mail servers.

SPF is also a newer type of record. It lets a domain encode information about what machines in the domain will send mail to the rest of the Internet. This helps receiving machines check that mail is valid. If mail is being received from a machine that calls itself *dodgy* but the domain records say that mail will only be sent out of the domain by a machine called *smtp*, chances are that the mail is forged junk mail.

Last on the list, *TXT* records were originally provided to allow domains to identify themselves in arbitrary ways. Nowadays, they usually encode machine-readable information, typically the *SPF* information.

Finally, we have the *Value* field. This field can be a number, a domain name, or an ASCII string. The semantics depend on the record type. A short description of the *Value* fields for each of the principal record types is given in Fig. 7-3.

For an example of the kind of information one might find in the DNS database of a domain, see Fig. 7-4. This figure depicts part of a (hypothetical) database for the *cs.vu.nl* domain shown in Fig. 7-1. The database contains seven types of resource records.

```
; Authoritative data for cs.vu.nl
cs.vu.nl.        86400   IN   SOA      star boss (9527,7200,7200,241920,86400)
cs.vu.nl.        86400   IN   MX       1 zephyr
cs.vu.nl.        86400   IN   MX       2 top
cs.vu.nl.        86400   IN   NS       star
star             86400   IN   A        130.37.56.205
zephyr           86400   IN   A        130.37.20.10
top              86400   IN   A        130.37.20.11
www              86400   IN   CNAME    star.cs.vu.nl
ftp              86400   IN   CNAME    zephyr.cs.vu.nl

flits            86400   IN   A        130.37.16.112
flits            86400   IN   A        192.31.231.165
flits            86400   IN   MX       1 flits
flits            86400   IN   MX       2 zephyr
flits            86400   IN   MX       3 top

rowboat                  IN   A        130.37.56.201
                         IN   MX       1 rowboat
                         IN   MX       2 zephyr

little-sister            IN   A        130.37.62.23

laserjet                 IN   A        192.31.231.216
```

Figure 7-4. A portion of a possible DNS database for *cs.vu.nl*.

The first noncomment line of Fig. 7-4 gives some basic information about the domain, which will not concern us further. Then come two entries giving the first

and second places to try to deliver email sent to *person@cs.vu.nl*. The *zephyr* (a specific machine) should be tried first. If that fails, the *top* should be tried as the next choice. The next line identifies the name server for the domain as *star*.

After the blank line (added for readability) come lines giving the IP addresses for the *star*, *zephyr*, and *top*. These are followed by an alias, *www.cs.vu.nl*, so that this address can be used without designating a specific machine. Creating this alias allows *cs.vu.nl* to change its World Wide Web server without invalidating the address people use to get to it. A similar argument holds for *ftp.cs.vu.nl*.

The section for the machine *flits* lists two IP addresses and three choices are given for handling email sent to *flits.cs.vu.nl*. First choice is naturally the *flits* itself, but if it is down, the *zephyr* and *top* are the second and third choices.

The next three lines contain a typical entry for a computer, in this case, *rowboat.cs.vu.nl*. The information provided contains the IP address and the primary and secondary mail drops. Then comes an entry for a computer that is not capable of receiving mail itself, followed by an entry that is likely for a printer that is connected to the Internet.

7.1.3 Name Servers

In theory at least, a single name server could contain the entire DNS database and respond to all queries about it. In practice, this server would be so overloaded as to be useless. Furthermore, if it ever went down, the entire Internet would be crippled.

To avoid the problems associated with having only a single source of information, the DNS name space is divided into nonoverlapping **zones**. One possible way to divide the name space of Fig. 7-1 is shown in Fig. 7-5. Each circled zone contains some part of the tree.

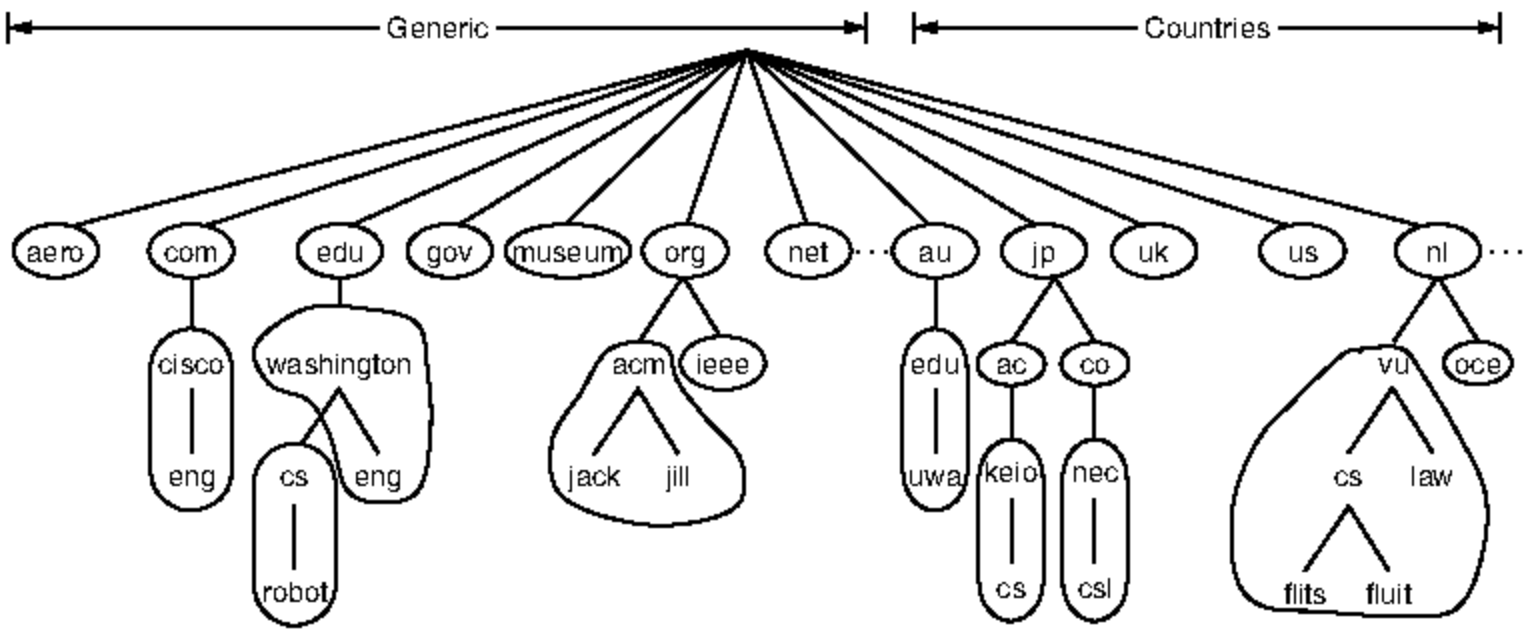


Figure 7-5. Part of the DNS name space divided into zones (which are circled).

Where the zone boundaries are placed within a zone is up to that zone's administrator. This decision is made in large part based on how many name servers are desired, and where. For example, in Fig. 7-5, the University of Washington has a zone for *washington.edu* that handles *eng.washington.edu* but does not handle *cs.washington.edu*. That is a separate zone with its own name servers. Such a decision might be made when a department such as English does not wish to run its own name server, but a department such as Computer Science does.

Each zone is also associated with one or more name servers. These are hosts that hold the database for the zone. Normally, a zone will have one primary name server, which gets its information from a file on its disk, and one or more secondary name servers, which get their information from the primary name server. To improve reliability, some of the name servers can be located outside the zone.

The process of looking up a name and finding an address is called **name resolution**. When a resolver has a query about a domain name, it passes the query to a local name server. If the domain being sought falls under the jurisdiction of the name server, such as *top.cs.vu.nl* falling under *cs.vu.nl*, it returns the authoritative resource records. An **authoritative record** is one that comes from the authority that manages the record and is thus always correct. Authoritative records are in contrast to **cached records**, which may be out of date.

What happens when the domain is remote, such as when *flits.cs.vu.nl* wants to find the IP address of *robot.cs.washington.edu* at UW (University of Washington)? In this case, and if there is no cached information about the domain available locally, the name server begins a remote query. This query follows the process shown in Fig. 7-6. Step 1 shows the query that is sent to the local name server. The query contains the domain name sought, the type (*A*), and the class(*IN*).

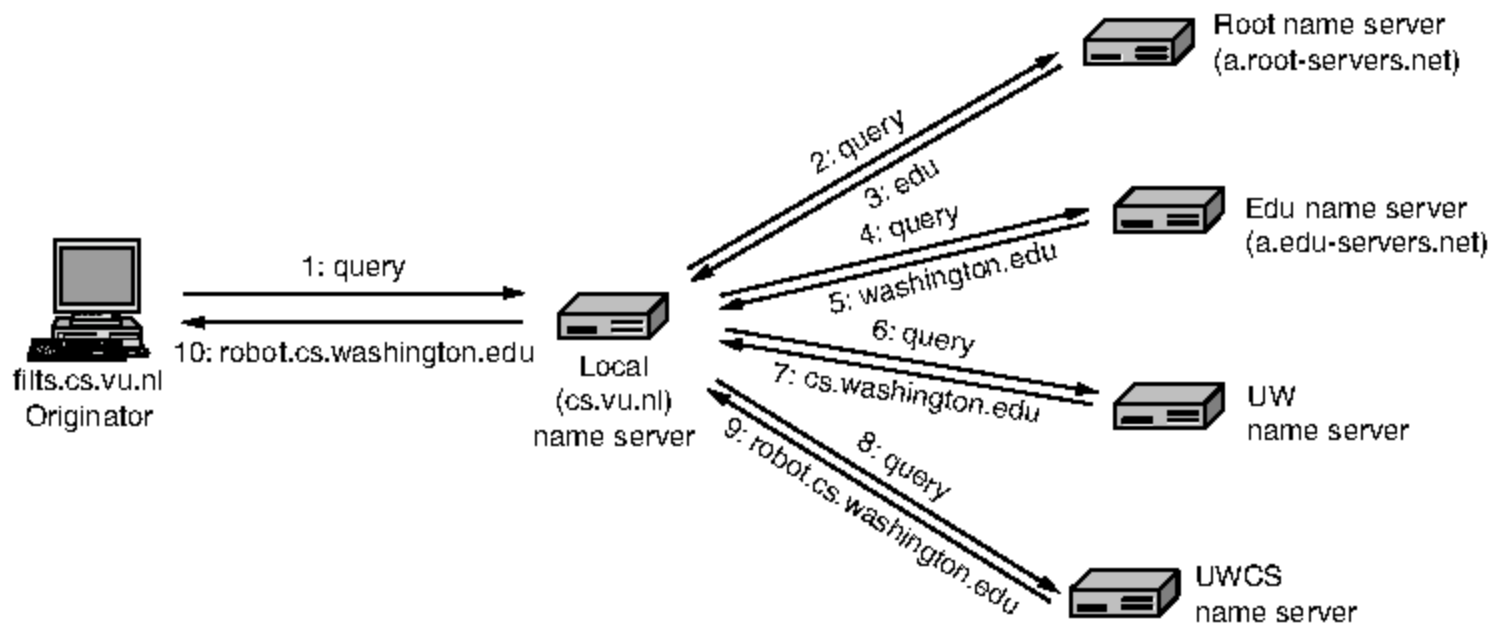


Figure 7-6. Example of a resolver looking up a remote name in 10 steps.

The next step is to start at the top of the name hierarchy by asking one of the **root name servers**. These name servers have information about each top-level

domain. This is shown as step 2 in Fig. 7-6. To contact a root server, each name server must have information about one or more root name servers. This information is normally present in a system configuration file that is loaded into the DNS cache when the DNS server is started. It is simply a list of *NS* records for the root and the corresponding *A* records.

There are 13 root DNS servers, unimaginatively called *a-root-servers.net* through *m.root-servers.net*. Each root server could logically be a single computer. However, since the entire Internet depends on the root servers, they are powerful and heavily replicated computers. Most of the servers are present in multiple geographical locations and reached using anycast routing, in which a packet is delivered to the nearest instance of a destination address; we described anycast in Chap. 5 The replication improves reliability and performance.

The root name server is unlikely to know the address of a machine at UW, and probably does not know the name server for UW either. But it must know the name server for the *edu* domain, in which *cs.washington.edu* is located. It returns the name and IP address for that part of the answer in step 3.

The local name server then continues its quest. It sends the entire query to the *edu* name server (*a.edu-servers.net*). That name server returns the name server for UW. This is shown in steps 4 and 5. Closer now, the local name server sends the query to the UW name server (step 6). If the domain name being sought was in the English department, the answer would be found, as the UW zone includes the English department. But the Computer Science department has chosen to run its own name server. The query returns the name and IP address of the UW Computer Science name server (step 7).

Finally, the local name server queries the UW Computer Science name server (step 8). This server is authoritative for the domain *cs.washington.edu*, so it must have the answer. It returns the final answer (step 9), which the local name server forwards as a response to *flits.cs.vu.nl* (step 10). The name has been resolved.

You can explore this process using standard tools such as the *dig* program that is installed on most UNIX systems. For example, typing

```
dig@a.edu-servers.net robot.cs.washington.edu
```

will send a query for *robot.cs.washington.edu* to the *a.edu-servers.net* name server and print out the result. This will show you the information obtained in step 4 in the example above, and you will learn the name and IP address of the UW name servers.

There are three technical points to discuss about this long scenario. First, two different query mechanisms are at work in Fig. 7-6. When the host *flits.cs.vu.nl* sends its query to the local name server, that name server handles the resolution on behalf of *flits* until it has the desired answer to return. It does *not* return partial answers. They might be helpful, but they are not what the query was seeking. This mechanism is called a **recursive query**.

On the other hand, the root name server (and each subsequent name server) does not recursively continue the query for the local name server. It just returns a partial answer and moves on to the next query. The local name server is responsible for continuing the resolution by issuing further queries. This mechanism is called an **iterative query**.

One name resolution can involve both mechanisms, as this example showed. A recursive query may always seem preferable, but many name servers (especially the root) will not handle them. They are too busy. Iterative queries put the burden on the originator. The rationale for the local name server supporting a recursive query is that it is providing a service to hosts in its domain. Those hosts do not have to be configured to run a full name server, just to reach the local one.

The second point is caching. All of the answers, including all the partial answers returned, are cached. In this way, if another *cs.vu.nl* host queries for *robot.cs.washington.edu* the answer will already be known. Even better, if a host queries for a different host in the same domain, say *galah.cs.washington.edu*, the query can be sent directly to the authoritative name server. Similarly, queries for other domains in *washington.edu* can start directly from the *washington.edu* name server. Using cached answers greatly reduces the steps in a query and improves performance. The original scenario we sketched is in fact the worst case that occurs when no useful information is cached.

However, cached answers are not authoritative, since changes made at *cs.washington.edu* will not be propagated to all the caches in the world that may know about it. For this reason, cache entries should not live too long. This is the reason that the *Time_to_live* field is included in each resource record. It tells remote name servers how long to cache records. If a certain machine has had the same IP address for years, it may be safe to cache that information for 1 day. For more volatile information, it might be safer to purge the records after a few seconds or a minute.

The third issue is the transport protocol that is used for the queries and responses. It is UDP. DNS messages are sent in UDP packets with a simple format for queries, answers, and name servers that can be used to continue the resolution. We will not go into the details of this format. If no response arrives within a short time, the DNS client repeats the query, trying another server for the domain after a small number of retries. This process is designed to handle the case of the server being down as well as the query or response packet getting lost. A 16-bit identifier is included in each query and copied to the response so that a name server can match answers to the corresponding query, even if multiple queries are outstanding at the same time.

Even though its purpose is simple, it should be clear that DNS is a large and complex distributed system that is comprised of millions of name servers that work together. It forms a key link between human-readable domain names and the IP addresses of machines. It includes replication and caching for performance and reliability and is designed to be highly robust.

We have not covered security, but as you might imagine, the ability to change the name-to-address mapping can have devastating consequences if done maliciously. For that reason, security extensions called DNSSEC have been developed for DNS. We will describe them in Chap. 8.

There is also application demand to use names in more flexible ways, for example, by naming content and resolving to the IP address of a nearby host that has the content. This fits the model of searching for and downloading a movie. It is the movie that matters, not the computer that has a copy of it, so all that is wanted is the IP address of any nearby computer that has a copy of the movie. Content distribution networks are one way to accomplish this mapping. We will describe how they build on the DNS later in this chapter, in Sec. 7.5.

7.2 ELECTRONIC MAIL

Electronic mail, or more commonly **email**, has been around for over three decades. Faster and cheaper than paper mail, email has been a popular application since the early days of the Internet. Before 1990, it was mostly used in academia. During the 1990s, it became known to the public at large and grew exponentially, to the point where the number of emails sent per day now is vastly more than the number of **snail mail** (i.e., paper) letters. Other forms of network communication, such as instant messaging and voice-over-IP calls have expanded greatly in use over the past decade, but email remains the workhorse of Internet communication. It is widely used within industry for intracompany communication, for example, to allow far-flung employees all over the world to cooperate on complex projects. Unfortunately, like paper mail, the majority of email—some 9 out of 10 messages—is junk mail or **spam** (McAfee, 2010).

Email, like most other forms of communication, has developed its own conventions and styles. It is very informal and has a low threshold of use. People who would never dream of calling up or even writing a letter to a Very Important Person do not hesitate for a second to send a sloppily written email to him or her. By eliminating most cues associated with rank, age, and gender, email debates often focus on content, not status. With email, a brilliant idea from a summer student can have more impact than a dumb one from an executive vice president.

Email is full of jargon such as BTW (By The Way), ROTFL (Rolling On The Floor Laughing), and IMHO (In My Humble Opinion). Many people also use little ASCII symbols called **smileys**, starting with the ubiquitous ":-)". Rotate the book 90 degrees clockwise if this symbol is unfamiliar. This symbol and other **emoticons** help to convey the tone of the message. They have spread to other terse forms of communication, such as instant messaging.

The email protocols have evolved during the period of their use, too. The first email systems simply consisted of file transfer protocols, with the convention that the first line of each message (i.e., file) contained the recipient's address. As time

went on, email diverged from file transfer and many features were added, such as the ability to send one message to a list of recipients. Multimedia capabilities became important in the 1990s to send messages with images and other non-text material. Programs for reading email became much more sophisticated too, shifting from text-based to graphical user interfaces and adding the ability for users to access their mail from their laptops wherever they happen to be. Finally, with the prevalence of spam, mail readers and the mail transfer protocols must now pay attention to finding and removing unwanted email.

In our description of email, we will focus on the way that mail messages are moved between users, rather than the look and feel of mail reader programs. Nevertheless, after describing the overall architecture, we will begin with the user-facing part of the email system, as it is familiar to most readers.

7.2.1 Architecture and Services

In this section, we will provide an overview of how email systems are organized and what they can do. The architecture of the email system is shown in Fig. 7-7. It consists of two kinds of subsystems: the **user agents**, which allow people to read and send email, and the **message transfer agents**, which move the messages from the source to the destination. We will also refer to message transfer agents informally as **mail servers**.

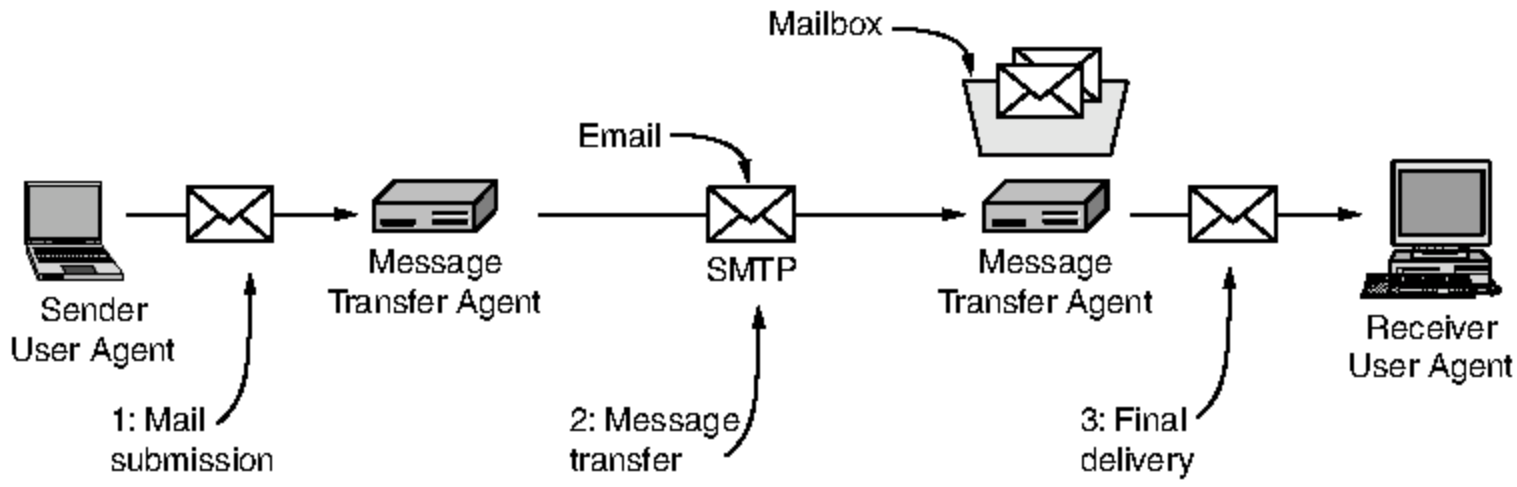


Figure 7-7. Architecture of the email system.

The user agent is a program that provides a graphical interface, or sometimes a text- and command-based interface that lets users interact with the email system. It includes a means to compose messages and replies to messages, display incoming messages, and organize messages by filing, searching, and discarding them. The act of sending new messages into the mail system for delivery is called **mail submission**.

Some of the user agent processing may be done automatically, anticipating what the user wants. For example, incoming mail may be filtered to extract or

deprioritize messages that are likely spam. Some user agents include advanced features, such as arranging for automatic email responses ("I'm having a wonderful vacation and it will be a while before I get back to you"). A user agent runs on the same computer on which a user reads her mail. It is just another program and may be run only some of the time.

The message transfer agents are typically system processes. They run in the background on mail server machines and are intended to be always available. Their job is to automatically move email through the system from the originator to the recipient with **SMTP** (**Simple Mail Transfer Protocol**). This is the message transfer step.

SMTP was originally specified as RFC 821 and revised to become the current RFC 5321. It sends mail over connections and reports back the delivery status and any errors. Numerous applications exist in which confirmation of delivery is important and may even have legal significance ("Well, Your Honor, my email system is just not very reliable, so I guess the electronic subpoena just got lost somewhere'').

Message transfer agents also implement **mailing lists**, in which an identical copy of a message is delivered to everyone on a list of email addresses. Other advanced features are carbon copies, blind carbon copies, high-priority email, secret (i.e., encrypted) email, alternative recipients if the primary one is not currently available, and the ability for assistants to read and answer their bosses' email.

Linking user agents and message transfer agents are the concepts of mailboxes and a standard format for email messages. **Mailboxes** store the email that is received for a user. They are maintained by mail servers. User agents simply present users with a view of the contents of their mailboxes. To do this, the user agents send the mail servers commands to manipulate the mailboxes, inspecting their contents, deleting messages, and so on. The retrieval of mail is the final delivery (step 3) in Fig. 7-7. With this architecture, one user may use different user agents on multiple computers to access one mailbox.

Mail is sent between message transfer agents in a standard format. The original format, RFC 822, has been revised to the current RFC 5322 and extended with support for multimedia content and international text. This scheme is called MIME and will be discussed later. People still refer to Internet email as RFC 822, though.

A key idea in the message format is the distinction between the **envelope** and its contents. The envelope encapsulates the message. It contains all the information needed for transporting the message, such as the destination address, priority, and security level, all of which are distinct from the message itself. The message transport agents use the envelope for routing, just as the post office does.

The message inside the envelope consists of two separate parts: the **header** and the **body**. The header contains control information for the user agents. The body is entirely for the human recipient. None of the agents care much about it. Envelopes and messages are illustrated in Fig. 7-8.

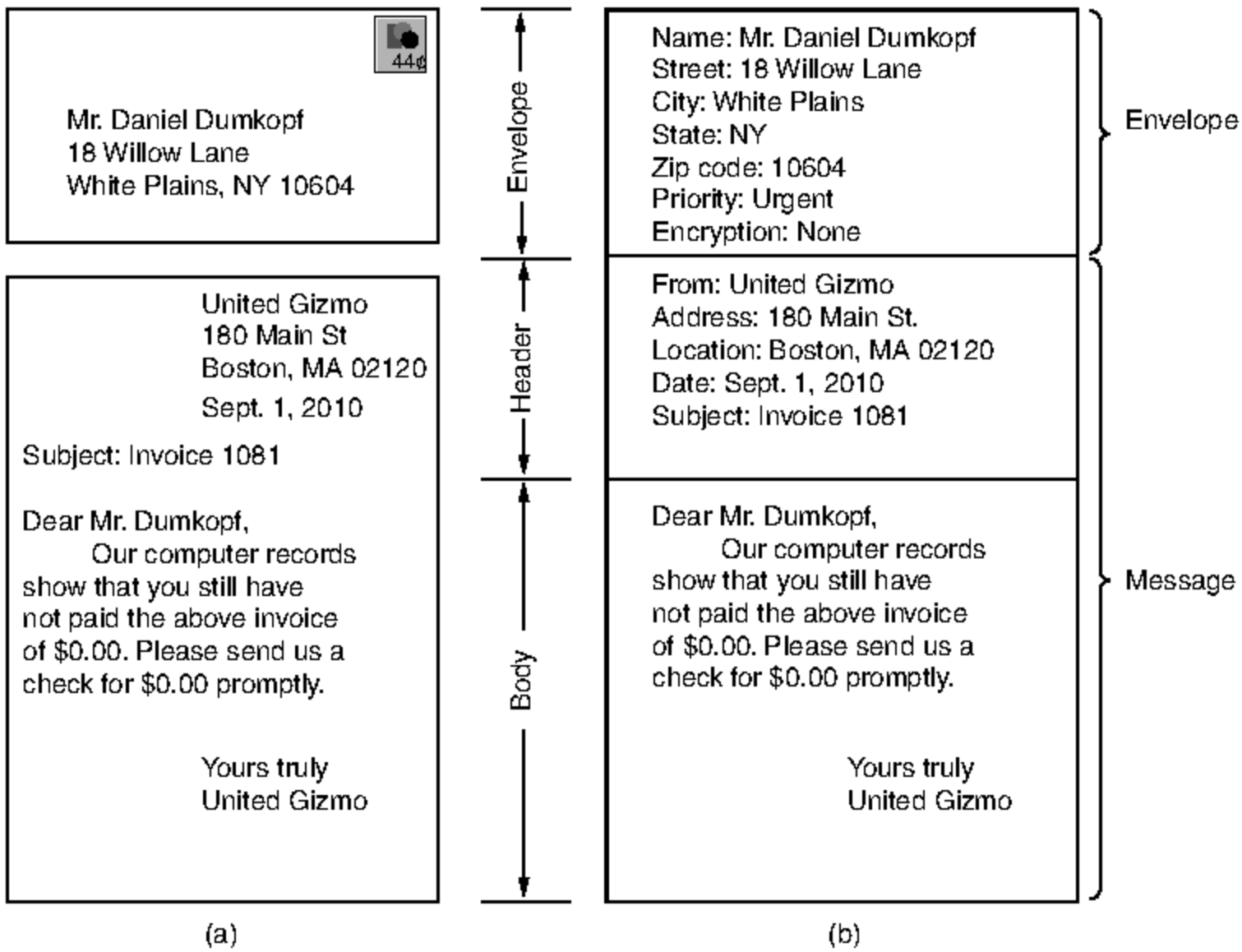


Figure 7-8. Envelopes and messages. (a) Paper mail. (b) Electronic mail.

We will examine the pieces of this architecture in more detail by looking at the steps that are involved in sending email from one user to another. This journey starts with the user agent.

7.2.2 The User Agent

A user agent is a program (sometimes called an **email reader**) that accepts a variety of commands for composing, receiving, and replying to messages, as well as for manipulating mailboxes. There are many popular user agents, including Google gmail, Microsoft Outlook, Mozilla Thunderbird, and Apple Mail. They can vary greatly in their appearance. Most user agents have a menu- or icon-driven graphical interface that requires a mouse, or a touch interface on smaller mobile devices. Older user agents, such as Elm, mh, and Pine, provide text-based interfaces and expect one-character commands from the keyboard. Functionally, these are the same, at least for text messages.

The typical elements of a user agent interface are shown in Fig. 7-9. Your mail reader is likely to be much flashier, but probably has equivalent functions.

When a user agent is started, it will usually present a summary of the messages in the user's mailbox. Often, the summary will have one line for each message in some sorted order. It highlights key fields of the message that are extracted from the message envelope or header.

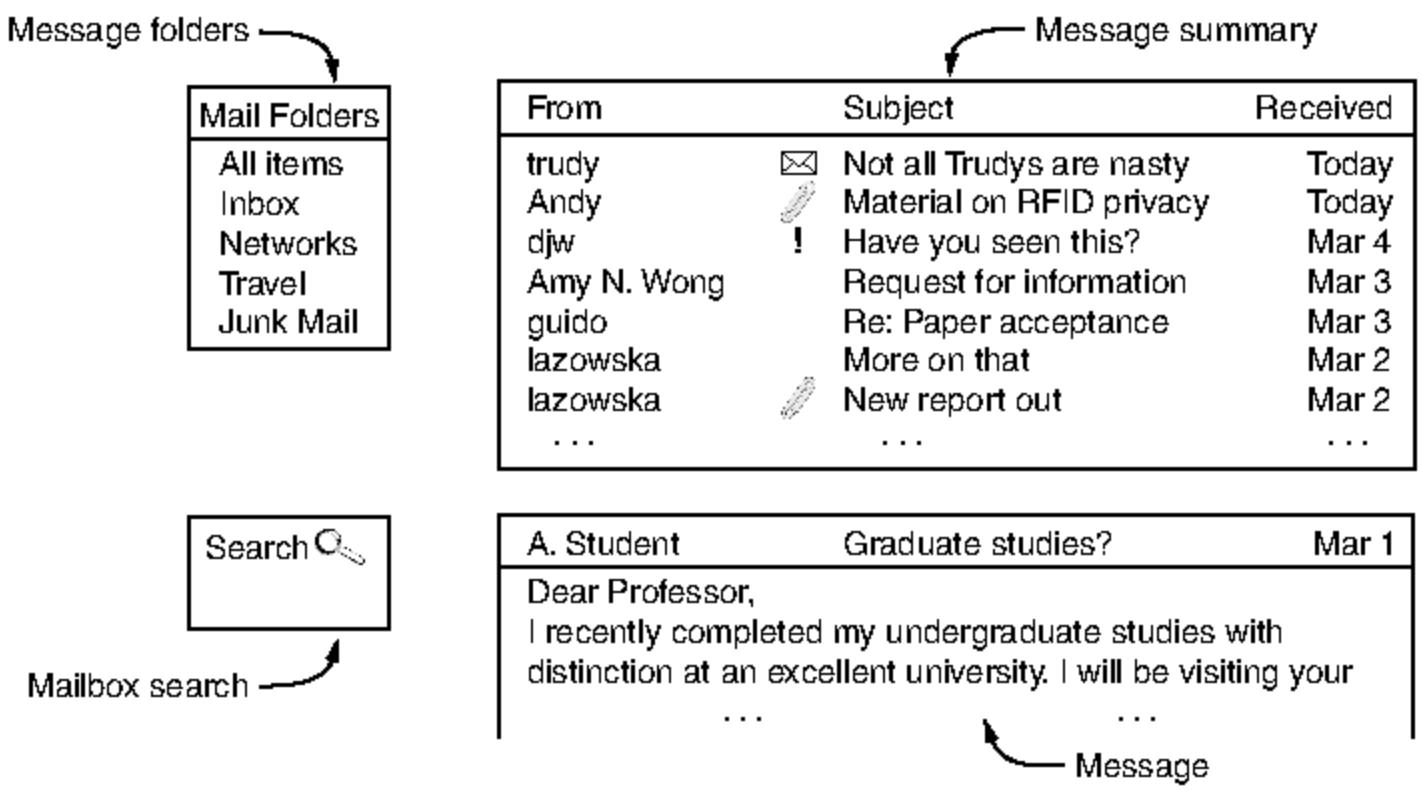


Figure 7-9. Typical elements of the user agent interface.

Seven summary lines are shown in the example of Fig. 7-9. The lines use the *From*, *Subject*, and *Received* fields, in that order, to display who sent the message, what it is about, and when it was received. All the information is formatted in a user-friendly way rather than displaying the literal contents of the message fields, but it is based on the message fields. Thus, people who fail to include a *Subject* field often discover that responses to their emails tend not to get the highest priority.

Many other fields or indications are possible. The icons next to the message subjects in Fig. 7-9 might indicate, for example, unread mail (the envelope), attached material (the paperclip), and important mail, at least as judged by the sender (the exclamation point).

Many sorting orders are also possible. The most common is to order messages based on the time that they were received, most recent first, with some indication as to whether the message is new or has already been read by the user. The fields in the summary and the sort order can be customized by the user according to her preferences.

User agents must also be able to display incoming messages as needed so that people can read their email. Often a short preview of a message is provided, as in Fig. 7-9, to help users decide when to read further. Previews may use small icons or images to describe the contents of the message. Other presentation processing

includes reformatting messages to fit the display, and translating or converting contents to more convenient formats (e.g., digitized speech to recognized text).

After a message has been read, the user can decide what to do with it. This is called **message disposition**. Options include deleting the message, sending a reply, forwarding the message to another user, and keeping the message for later reference. Most user agents can manage one mailbox for incoming mail with multiple folders for saved mail. The folders allow the user to save message according to sender, topic, or some other category.

Filing can be done automatically by the user agent as well, before the user reads the messages. A common example is that the fields and contents of messages are inspected and used, along with feedback from the user about previous messages, to determine if a message is likely to be spam. Many ISPs and companies run software that labels mail as important or spam so that the user agent can file it in the corresponding mailbox. The ISP and company have the advantage of seeing mail for many users and may have lists of known spammers. If hundreds of users have just received a similar message, it is probably spam. By presorting incoming mail as "probably legitimate" and "probably spam," the user agent can save users a fair amount of work separating the good stuff from the junk.

And the most popular spam? It is generated by collections of compromised computers called **botnets** and its content depends on where you live. Fake diplomas are topical in Asia, and cheap drugs and other dubious product offers are topical in the U.S. Unclaimed Nigerian bank accounts still abound. Pills for enlarging various body parts are common everywhere.

Other filing rules can be constructed by users. Each rule specifies a condition and an action. For example, a rule could say that any message received from the boss goes to one folder for immediate reading and any message from a particular mailing list goes to another folder for later reading. Several folders are shown in Fig. 7-9. The most important folders are the Inbox, for incoming mail not filed elsewhere, and Junk Mail, for messages that are thought to be spam.

As well as explicit constructs like folders, user agents now provide rich capabilities to search the mailbox. This feature is also shown in Fig. 7-9. Search capabilities let users find messages quickly, such as the message about "where to buy Vegemite" that someone sent in the last month.

Email has come a long way from the days when it was just file transfer. Providers now routinely support mailboxes with up to 1 GB of stored mail that details a user's interactions over a long period of time. The sophisticated mail handling of user agents with search and automatic forms of processing is what makes it possible to manage these large volumes of email. For people who send and receive thousands of messages a year, these tools are invaluable.

Another useful feature is the ability to automatically respond to messages in some way. One response is to forward incoming email to a different address, for example, a computer operated by a commercial paging service that pages the user

by using radio or satellite and displays the *Subject:* line on his pager. These **auto-responders** must run in the mail server because the user agent may not run all the time and may only occasionally retrieve email. Because of these factors, the user agent cannot provide a true automatic response. However, the interface for automatic responses is usually presented by the user agent.

A different example of an automatic response is a **vacation agent**. This is a program that examines each incoming message and sends the sender an insipid reply such as: "Hi. I'm on vacation. I'll be back on the 24th of August. Talk to you then." Such replies can also specify how to handle urgent matters in the interim, other people to contact for specific problems, etc. Most vacation agents keep track of whom they have sent canned replies to and refrain from sending the same person a second reply. There are pitfalls with these agents, however. For example, it is not advisable to send a canned reply to a large mailing list.

Let us now turn to the scenario of one user sending a message to another user. One of the basic features user agents support that we have not yet discussed is mail composition. It involves creating messages and answers to messages and sending these messages into the rest of the mail system for delivery. Although any text editor can be used to create the body of the message, editors are usually integrated with the user agent so that it can provide assistance with addressing and the numerous header fields attached to each message. For example, when answering a message, the email system can extract the originator's address from the incoming email and automatically insert it into the proper place in the reply. Other common features are appending a **signature block** to the bottom of a message, correcting spelling, and computing digital signatures that show the message is valid.

Messages that are sent into the mail system have a standard format that must be created from the information supplied to the user agent. The most important part of the message for transfer is the envelope, and the most important part of the envelope is the destination address. This address must be in a format that the message transfer agents can deal with.

The expected form of an address is *user@dns-address*. Since we studied DNS earlier in this chapter, we will not repeat that material here. However, it is worth noting that other forms of addressing exist. In particular, **X.400** addresses look radically different from DNS addresses.

X.400 is an ISO standard for message-handling systems that was at one time a competitor to SMTP. SMTP won out handily, though X.400 systems are still used, mostly outside of the U.S. X.400 addresses are composed of *attribute=value* pairs separated by slashes, for example,

```
/C=US/ST=MASSACHUSETTS/L=CAMBRIDGE/PA=360 MEMORIAL DR./CN=KEN SMITH/
```

This address specifies a country, state, locality, personal address, and common name (Ken Smith). Many other attributes are possible, so you can send email to

someone whose exact email address you do not know, provided you know enough other attributes (e.g., company and job title).

Although X.400 names are considerably less convenient than DNS names, the issue is moot for user agents because they have user-friendly aliases (sometimes called nicknames) that allow users to enter or select a person's name and get the correct email address. Consequently, it is usually not necessary to actually type in these strange strings.

A final point we will touch on for sending mail is mailing lists, which let users send the same message to a list of people with a single command. There are two choices for how the mailing list is maintained. It might be maintained locally, by the user agent. In this case, the user agent can just send a separate message to each intended recipient.

Alternatively, the list may be maintained remotely at a message transfer agent. Messages will then be expanded in the message transfer system, which has the effect of allowing multiple users to send to the list. For example, if a group of bird watchers has a mailing list called *birders* installed on the transfer agent *meadowlark.arizona.edu*, any message sent to *birders@meadowlark.arizona.edu* will be routed to the University of Arizona and expanded into individual messages to all the mailing list members, wherever in the world they may be. Users of this mailing list cannot tell that it is a mailing list. It could just as well be the personal mailbox of Prof. Gabriel O. Birders.

7.2.3 Message Formats

Now we turn from the user interface to the format of the email messages themselves. Messages sent by the user agent must be placed in a standard format to be handled by the message transfer agents. First we will look at basic ASCII email using RFC 5322, which is the latest revision of the original Internet message format as described in RFC 822. After that, we will look at multimedia extensions to the basic format.

RFC 5322—The Internet Message Format

Messages consist of a primitive envelope (described as part of SMTP in RFC 5321), some number of header fields, a blank line, and then the message body. Each header field (logically) consists of a single line of ASCII text containing the field name, a colon, and, for most fields, a value. The original RFC 822 was designed decades ago and did not clearly distinguish the envelope fields from the header fields. Although it has been revised to RFC 5322, completely redoing it was not possible due to its widespread usage. In normal usage, the user agent builds a message and passes it to the message transfer agent, which then uses some of the header fields to construct the actual envelope, a somewhat old-fashioned mixing of message and envelope.

The principal header fields related to message transport are listed in Fig. 7-10. The *To:* field gives the DNS address of the primary recipient. Having multiple recipients is also allowed. The *Cc:* field gives the addresses of any secondary recipients. In terms of delivery, there is no distinction between the primary and secondary recipients. It is entirely a psychological difference that may be important to the people involved but is not important to the mail system. The term *Cc:* (Carbon copy) is a bit dated, since computers do not use carbon paper, but it is well established. The *Bcc:* (Blind carbon copy) field is like the *Cc:* field, except that this line is deleted from all the copies sent to the primary and secondary recipients. This feature allows people to send copies to third parties without the primary and secondary recipients knowing this.

Header	Meaning
To:	Email address(es) of primary recipient(s)
Cc:	Email address(es) of secondary recipient(s)
Bcc:	Email address(es) for blind carbon copies
From:	Person or people who created the message
Sender:	Email address of the actual sender
Received:	Line added by each transfer agent along the route
Return-Path:	Can be used to identify a path back to the sender

Figure 7-10. RFC 5322 header fields related to message transport.

The next two fields, *From:* and *Sender:*, tell who wrote and sent the message, respectively. These need not be the same. For example, a business executive may write a message, but her assistant may be the one who actually transmits it. In this case, the executive would be listed in the *From:* field and the assistant in the *Sender:* field. The *From:* field is required, but the *Sender:* field may be omitted if it is the same as the *From:* field. These fields are needed in case the message is undeliverable and must be returned to the sender.

A line containing *Received:* is added by each message transfer agent along the way. The line contains the agent's identity, the date and time the message was received, and other information that can be used for debugging the routing system.

The *Return-Path:* field is added by the final message transfer agent and was intended to tell how to get back to the sender. In theory, this information can be gathered from all the *Received:* headers (except for the name of the sender's mailbox), but it is rarely filled in as such and typically just contains the sender's address.

In addition to the fields of Fig. 7-10, RFC 5322 messages may also contain a variety of header fields used by the user agents or human recipients. The most common ones are listed in Fig. 7-11. Most of these are self-explanatory, so we will not go into all of them in much detail.

Header	Meaning
Date:	The date and time the message was sent
Reply-To:	Email address to which replies should be sent
Message-Id:	Unique number for referencing this message later
In-Reply-To:	Message-Id of the message to which this is a reply
References:	Other relevant Message-Ids
Keywords:	User-chosen keywords
Subject:	Short summary of the message for the one-line display

Figure 7-11. Some fields used in the RFC 5322 message header.

The *Reply-To:* field is sometimes used when neither the person composing the message nor the person sending the message wants to see the reply. For example, a marketing manager may write an email message telling customers about a new product. The message is sent by an assistant, but the *Reply-To:* field lists the head of the sales department, who can answer questions and take orders. This field is also useful when the sender has two email accounts and wants the reply to go to the other one.

The *Message-Id:* is an automatically generated number that is used to link messages together (e.g., when used in the *In-Reply-To:* field) and to prevent duplicate delivery.

The RFC 5322 document explicitly says that users are allowed to invent optional headers for their own private use. By convention since RFC 822, these headers start with the string *X-*. It is guaranteed that no future headers will use names starting with *X-*, to avoid conflicts between official and private headers. Sometimes wiseguy undergraduates make up fields like *X-Fruit-of-the-Day:* or *X-Disease-of-the-Week:*, which are legal, although not always illuminating.

After the headers comes the message body. Users can put whatever they want here. Some people terminate their messages with elaborate signatures, including quotations from greater and lesser authorities, political statements, and disclaimers of all kinds (e.g., The XYZ Corporation is not responsible for my opinions; in fact, it cannot even comprehend them).

MIME—The Multipurpose Internet Mail Extensions

In the early days of the ARPANET, email consisted exclusively of text messages written in English and expressed in ASCII. For this environment, the early RFC 822 format did the job completely: it specified the headers but left the content entirely up to the users. In the 1990s, the worldwide use of the Internet and demand to send richer content through the mail system meant that this approach was no longer adequate. The problems included sending and receiving messages

in languages with accents (e.g., French and German), non-Latin alphabets (e.g., Hebrew and Russian), or no alphabets (e.g., Chinese and Japanese), as well as sending messages not containing text at all (e.g., audio, images, or binary documents and programs).

The solution was the development of **MIME** (**Multipurpose Internet Mail Extensions**). It is widely used for mail messages that are sent across the Internet, as well as to describe content for other applications such as Web browsing. MIME is described in RFCs 2045–2047, 4288, 4289, and 2049.

The basic idea of MIME is to continue to use the RFC 822 format (the precursor to RFC 5322 the time MIME was proposed) but to add structure to the message body and define encoding rules for the transfer of non-ASCII messages. Not deviating from RFC 822 allowed MIME messages to be sent using the existing mail transfer agents and protocols (based on RFC 821 then, and RFC 5321 now). All that had to be changed were the sending and receiving programs, which users could do for themselves.

MIME defines five new message headers, as shown in Fig. 7-12. The first of these simply tells the user agent receiving the message that it is dealing with a MIME message, and which version of MIME it uses. Any message not containing a *MIME-Version:* header is assumed to be an English plaintext message (or at least one using only ASCII characters) and is processed as such.

Header	Meaning
MIME-Version:	Identifies the MIME version
Content-Description:	Human-readable string telling what is in the message
Content-Id:	Unique identifier
Content-Transfer-Encoding:	How the body is wrapped for transmission
Content-Type:	Type and format of the content

Figure 7-12. Message headers added by MIME.

The *Content-Description:* header is an ASCII string telling what is in the message. This header is needed so the recipient will know whether it is worth decoding and reading the message. If the string says "Photo of Barbara's hamster" and the person getting the message is not a big hamster fan, the message will probably be discarded rather than decoded into a high-resolution color photograph.

The *Content-Id:* header identifies the content. It uses the same format as the standard *Message-Id:* header.

The *Content-Transfer-Encoding:* tells how the body is wrapped for transmission through the network. A key problem at the time MIME was developed was that the mail transfer (SMTP) protocols expected ASCII messages in which no line exceeded 1000 characters. ASCII characters use 7 bits out of each 8-bit byte. Binary data such as executable programs and images use all 8 bits of each byte, as

do extended character sets. There was no guarantee this data would be transferred safely. Hence, some method of carrying binary data that made it look like a regular ASCII mail message was needed. Extensions to SMTP since the development of MIME do allow 8-bit binary data to be transferred, though even today binary data may not always go through the mail system correctly if unencoded.

MIME provides five transfer encoding schemes, plus an escape to new schemes—just in case. The simplest scheme is just ASCII text messages. ASCII characters use 7 bits and can be carried directly by the email protocol, provided that no line exceeds 1000 characters.

The next simplest scheme is the same thing, but using 8-bit characters, that is, all values from 0 up to and including 255 are allowed. Messages using the 8-bit encoding must still adhere to the standard maximum line length.

Then there are messages that use a true binary encoding. These are arbitrary binary files that not only use all 8 bits but also do not adhere to the 1000-character line limit. Executable programs fall into this category. Nowadays, mail servers can negotiate to send data in binary (or 8-bit) encoding, falling back to ASCII if both ends do not support the extension.

The ASCII encoding of binary data is called **base64 encoding**. In this scheme, groups of 24 bits are broken up into four 6-bit units, with each unit being sent as a legal ASCII character. The coding is "A" for 0, "B" for 1, and so on, followed by the 26 lowercase letters, the 10 digits, and finally + and / for 62 and 63, respectively. The == and = sequences indicate that the last group contained only 8 or 16 bits, respectively. Carriage returns and line feeds are ignored, so they can be inserted at will in the encoded character stream to keep the lines short enough. Arbitrary binary text can be sent safely using this scheme, albeit inefficiently. This encoding was very popular before binary-capable mail servers were widely deployed. It is still commonly seen.

For messages that are almost entirely ASCII but with a few non-ASCII characters, base64 encoding is somewhat inefficient. Instead, an encoding known as **quoted-printable encoding** is used. This is just 7-bit ASCII, with all the characters above 127 encoded as an equals sign followed by the character's value as two hexadecimal digits. Control characters, some punctuation marks and math symbols, as well as trailing spaces are also so encoded.

Finally, when there are valid reasons not to use one of these schemes, it is possible to specify a user-defined encoding in the *Content-Transfer-Encoding:* header.

The last header shown in Fig. 7-12 is really the most interesting one. It specifies the nature of the message body and has had an impact well beyond email. For instance, content downloaded from the Web is labeled with MIME types so that the browser knows how to present it. So is content sent over streaming media and real-time transports such as voice over IP.

Initially, seven MIME types were defined in RFC 1521. Each type has one or more available subtypes. The type and subtype are separated by a slash, as in

"Content-Type: video/mpeg''. Since then, hundreds of subtypes have been added, along with another type. Additional entries are being added all the time as new types of content are developed. The list of assigned types and subtypes is maintained online by IANA at *www.iana.org/assignments/media-types*.

The types, along with examples of commonly used subtypes, are given in Fig. 7-13. Let us briefly go through them, starting with *text*. The *text/plain* combination is for ordinary messages that can be displayed as received, with no encoding and no further processing. This option allows ordinary messages to be transported in MIME with only a few extra headers. The *text/html* subtype was added when the Web became popular (in RFC 2854) to allow Web pages to be sent in RFC 822 email. A subtype for the eXtensible Markup Language, *text/xml*, is defined in RFC 3023. XML documents have proliferated with the development of the Web. We will study HTML and XML in Sec. 7.3.

Type	Example subtypes	Description
text	plain, html, xml, css	Text in various formats
image	gif, jpeg, tiff	Pictures
audio	basic, mpeg, mp4	Sounds
video	mpeg, mp4, quicktime	Movies
model	vrml	3D model
application	octet-stream, pdf, javascript, zip	Data produced by applications
message	http, rfc822	Encapsulated message
multipart	mixed, alternative, parallel, digest	Combination of multiple types

Figure 7-13. MIME content types and example subtypes.

The next MIME type is *image*, which is used to transmit still pictures. Many formats are widely used for storing and transmitting images nowadays, both with and without compression. Several of these, including GIF, JPEG, and TIFF, are built into nearly all browsers. Many other formats and corresponding subtypes exist as well.

The *audio* and *video* types are for sound and moving pictures, respectively. Please note that *video* may include only the visual information, not the sound. If a movie with sound is to be transmitted, the video and audio portions may have to be transmitted separately, depending on the encoding system used. The first video format defined was the one devised by the modestly named Moving Picture Experts Group (MPEG), but others have been added since. In addition to *audio/basic*, a new audio type, *audio/mpeg*, was added in RFC 3003 to allow people to email MP3 audio files. The *video/mp4* and *audio/mp4* types signal video and audio data that are stored in the newer MPEG 4 format.

The *model* type was added after the other content types. It is intended for describing 3D model data. However, it has not been widely used to date.

The *application* type is a catchall for formats that are not covered by one of the other types and that require an application to interpret the data. We have listed the subtypes *pdf*, *javascript*, and *zip* as examples for PDF documents, JavaScript programs, and Zip archives, respectively. User agents that receive this content use a third-party library or external program to display the content; the display may or may not appear to be integrated with the user agent.

By using MIME types, user agents gain the extensibility to handle new types of application content as it is developed. This is a significant benefit. On the other hand, many of the new forms of content are executed or interpreted by applications, which presents some dangers. Obviously, running an arbitrary executable program that has arrived via the mail system from "friends" poses a security hazard. The program may do all sorts of nasty damage to the parts of the computer to which it has access, especially if it can read and write files and use the network. Less obviously, document formats can pose the same hazards. This is because formats such as PDF are full-blown programming languages in disguise. While they are interpreted and restricted in scope, bugs in the interpreter often allow devious documents to escape the restrictions.

Besides these examples, there are many more application subtypes because there are many more applications. As a fallback to be used when no other subtype is known to be more fitting, the *octet-stream* subtype denotes a sequence of uninterpreted bytes. Upon receiving such a stream, it is likely that a user agent will display it by suggesting to the user that it be copied to a file. Subsequent processing is then up to the user, who presumably knows what kind of content it is.

The last two types are useful for composing and manipulating messages themselves. The *message* type allows one message to be fully encapsulated inside another. This scheme is useful for forwarding email, for example. When a complete RFC 822 message is encapsulated inside an outer message, the *rfc822* subtype should be used. Similarly, it is common for HTML documents to be encapsulated. And the *partial* subtype makes it possible to break an encapsulated message into pieces and send them separately (for example, if the encapsulated message is too long). Parameters make it possible to reassemble all the parts at the destination in the correct order.

Finally, the *multipart* type allows a message to contain more than one part, with the beginning and end of each part being clearly delimited. The *mixed* subtype allows each part to be a different type, with no additional structure imposed. Many email programs allow the user to provide one or more attachments to a text message. These attachments are sent using the *multipart* type.

In contrast to *mixed*, the *alternative* subtype allows the same message to be included multiple times but expressed in two or more different media. For example, a message could be sent in plain ASCII, in HMTL, and in PDF. A properly designed user agent getting such a message would display it according to user preferences. Likely PDF would be the first choice, if that is possible. The second choice would be HTML. If neither of these were possible, then the flat ASCII

text would be displayed. The parts should be ordered from simplest to most complex to help recipients with pre-MIME user agents make some sense of the message (e.g., even a pre-MIME user can read flat ASCII text).

The *alternative* subtype can also be used for multiple languages. In this context, the Rosetta Stone can be thought of as an early *multipart/alternative* message.

Of the other two example subtypes, the *parallel* subtype is used when all parts must be "viewed" simultaneously. For example, movies often have an audio channel and a video channel. Movies are more effective if these two channels are played back in parallel, instead of consecutively. The *digest* subtype is used when multiple messages are packed together into a composite message. For example, some discussion groups on the Internet collect messages from subscribers and then send them out to the group periodically as a single *multipart/digest* message.

As an example of how MIME types may be used for email messages, a multimedia message is shown in Fig. 7-14. Here, a birthday greeting is transmitted in alternative forms as HTML and as an audio file. Assuming the receiver has audio capability, the user agent there will play the sound file. In this example, the sound is carried by reference as a *message/external-body* subtype, so first the user agent must fetch the sound file *birthday.snd* using FTP. If the user agent has no audio capability, the lyrics are displayed on the screen in stony silence. The two parts are delimited by two hyphens followed by a (software-generated) string specified in the *boundary* parameter.

Note that the *Content-Type* header occurs in three positions within this example. At the top level, it indicates that the message has multiple parts. Within each part, it gives the type and subtype of that part. Finally, within the body of the second part, it is required to tell the user agent what kind of external file it is to fetch. To indicate this slight difference in usage, we have used lowercase letters here, although all headers are case insensitive. The *Content-Transfer-Encoding* is similarly required for any external body that is not encoded as 7-bit ASCII.

7.2.4 Message Transfer

Now that we have described user agents and mail messages, we are ready to look at how the message transfer agents relay messages from the originator to the recipient. The mail transfer is done with the SMTP protocol.

The simplest way to move messages is to establish a transport connection from the source machine to the destination machine and then just transfer the message. This is how SMTP originally worked. Over the years, however, two different uses of SMTP have been differentiated. The first use is **mail submission**, step 1 in the email architecture of Fig. 7-7. This is the means by which user agents send messages into the mail system for delivery. The second use is to transfer messages between message transfer agents (step 2 in Fig. 7-7). This

```
From: alice@cs.washington.edu
To: bob@ee.uwa.edu.au
MIME-Version: 1.0
Message-Id: <0704760941.AA00747@cs.washington.edu>
Content-Type: multipart/alternative; boundary=qwertyuiopasdfghjklzxcvbnm
Subject: Earth orbits sun integral number of times

This is the preamble. The user agent ignores it. Have a nice day.

--qwertyuiopasdfghjklzxcvbnm
Content-Type: text/html

<p>Happy birthday to you<br>
Happy birthday to you<br>
Happy birthday dear <b> Bob </b><br>
Happy birthday to you</p>

--qwertyuiopasdfghjklzxcvbnm
Content-Type: message/external-body;
        access-type="anon-ftp";
        site="bicycle.cs.washington.edu";
        directory="pub";
        name="birthday.snd"

content-type: audio/basic
content-transfer-encoding: base64
--qwertyuiopasdfghjklzxcvbnm--
```

Figure 7-14. A multipart message containing HTML and audio alternatives.

sequence delivers mail all the way from the sending to the receiving message transfer agent in one hop. Final delivery is accomplished with different protocols that we will describe in the next section.

In this section, we will describe the basics of the SMTP protocol and its extension mechanism. Then we will discuss how it is used differently for mail submission and message transfer.

SMTP (Simple Mail Transfer Protocol) and Extensions

Within the Internet, email is delivered by having the sending computer establish a TCP connection to port 25 of the receiving computer. Listening to this port is a mail server that speaks **SMTP** (**Simple Mail Transfer Protocol**). This server accepts incoming connections, subject to some security checks, and accepts messages for delivery. If a message cannot be delivered, an error report containing the first part of the undeliverable message is returned to the sender.

SMTP is a simple ASCII protocol. This is not a weakness but a feature. Using ASCII text makes protocols easy to develop, test, and debug. They can be

tested by sending commands manually, and records of the messages are easy to read. Most application-level Internet protocols now work this way (e.g., HTTP).

We will walk through a simple message transfer between mail servers that delivers a message. After establishing the TCP connection to port 25, the sending machine, operating as the client, waits for the receiving machine, operating as the server, to talk first. The server starts by sending a line of text giving its identity and telling whether it is prepared to receive mail. If it is not, the client releases the connection and tries again later.

If the server is willing to accept email, the client announces whom the email is coming from and whom it is going to. If such a recipient exists at the destination, the server gives the client the go-ahead to send the message. Then the client sends the message and the server acknowledges it. No checksums are needed because TCP provides a reliable byte stream. If there is more email, that is now sent. When all the email has been exchanged in both directions, the connection is released. A sample dialog for sending the message of Fig. 7-14, including the numerical codes used by SMTP, is shown in Fig. 7-15. The lines sent by the client (i.e., the sender) are marked *C:*. Those sent by the server (i.e., the receiver) are marked *S:*.

The first command from the client is indeed meant to be *HELO*. Of the various four-character abbreviations for *HELLO*, this one has numerous advantages over its biggest competitor. Why all the commands had to be four characters has been lost in the mists of time.

In Fig. 7-15, the message is sent to only one recipient, so only one *RCPT* command is used. Such commands are allowed to send a single message to multiple receivers. Each one is individually acknowledged or rejected. Even if some recipients are rejected (because they do not exist at the destination), the message can be sent to the other ones.

Finally, although the syntax of the four-character commands from the client is rigidly specified, the syntax of the replies is less rigid. Only the numerical code really counts. Each implementation can put whatever string it wants after the code.

The basic SMTP works well, but it is limited in several respects. It does not include authentication. This means that the *FROM* command in the example could give any sender address that it pleases. This is quite useful for sending spam. Another limitation is that SMTP transfers ASCII messages, not binary data. This is why the base64 MIME content transfer encoding was needed. However, with that encoding the mail transmission uses bandwidth inefficiently, which is an issue for large messages. A third limitation is that SMTP sends messages in the clear. It has no encryption to provide a measure of privacy against prying eyes.

To allow these and many other problems related to message processing to be addressed, SMTP was revised to have an extension mechanism. This mechanism is a mandatory part of the RFC 5321 standard. The use of SMTP with extensions is called **ESMTP** (**Extended SMTP**).

```
                    S: 220 ee.uwa.edu.au SMTP service ready
C: HELO abcd.com
                    S: 250 cs.washington.edu says hello to ee.uwa.edu.au
C: MAIL FROM: <alice@cs.washington.edu>
                    S: 250 sender ok
C: RCPT TO: <bob@ee.uwa.edu.au>
                    S: 250 recipient ok
C: DATA
                    S: 354 Send mail; end with "." on a line by itself
C: From: alice@cs.washington.edu
C: To: bob@ee.uwa.edu.au
C: MIME-Version: 1.0
C: Message-Id: <0704760941.AA00747@ee.uwa.edu.au>
C: Content-Type: multipart/alternative; boundary=qwertyuiopasdfghjklzxcvbnm
C: Subject: Earth orbits sun integral number of times
C:
C: This is the preamble. The user agent ignores it. Have a nice day.
C:
C: --qwertyuiopasdfghjklzxcvbnm
C: Content-Type: text/html
C:
C: <p>Happy birthday to you
C: Happy birthday to you
C: Happy birthday dear <bold> Bob </bold>
C: Happy birthday to you
C:
C: --qwertyuiopasdfghjklzxcvbnm
C: Content-Type: message/external-body;
C:          access-type="anon-ftp";
C:          site="bicycle.cs.washington.edu";
C:          directory="pub";
C:          name="birthday.snd"
C:
C: content-type: audio/basic
C: content-transfer-encoding: base64
C: --qwertyuiopasdfghjklzxcvbnm
C: .
                    S: 250 message accepted
C: QUIT
                    S: 221 ee.uwa.edu.au closing connection
```

Figure 7-15. Sending a message from *alice@cs.washington.edu* to *bob@ee.uwa.edu.au.*

Clients wanting to use an extension send an *EHLO* message instead of *HELO* initially. If this is rejected, the server is a regular SMTP server, and the client should proceed in the usual way. If the *EHLO* is accepted, the server replies with the extensions that it supports. The client may then use any of these extensions. Several common extensions are shown in Fig. 7-16. The figure gives the keyword

as used in the extension mechanism, along with a description of the new functionality. We will not go into extensions in further detail.

Keyword	Description
AUTH	Client authentication
BINARYMIME	Server accepts binary messages
CHUNKING	Server accepts large messages in chunks
SIZE	Check message size before trying to send
STARTTLS	Switch to secure transport (TLS; see Chap. 8)
UTF8SMTP	Internationalized addresses

Figure 7-16. Some SMTP extensions.

To get a better feel for how SMTP and some of the other protocols described in this chapter work, try them out. In all cases, first go to a machine connected to the Internet. On a UNIX (or Linux) system, in a shell, type

```
telnet mail.isp.com 25
```

substituting the DNS name of your ISP's mail server for *mail.isp.com*. On a Windows XP system, click on Start, then Run, and type the command in the dialog box. On a Vista or Windows 7 machine, you may have to first install the telnet program (or equivalent) and then start it yourself. This command will establish a telnet (i.e., TCP) connection to port 25 on that machine. Port 25 is the SMTP port; see Fig. 6-34 for the ports for other common protocols. You will probably get a response something like this:

```
Trying 192.30.200.66...
Connected to mail.isp.com
Escape character is '^]'.
220 mail.isp.com Smail #74 ready at Thu, 25 Sept 2002 13:26 +0200
```

The first three lines are from telnet, telling you what it is doing. The last line is from the SMTP server on the remote machine, announcing its willingness to talk to you and accept email. To find out what commands it accepts, type

```
HELP
```

From this point on, a command sequence such as the one in Fig. 7-16 is possible if the server is willing to accept mail from you.

Mail Submission

Originally, user agents ran on the same computer as the sending message transfer agent. In this setting, all that is required to send a message is for the user agent to talk to the local mail server, using the dialog that we have just described. However, this setting is no longer the usual case.

User agents often run on laptops, home PCs, and mobile phones. They are not always connected to the Internet. Mail transfer agents run on ISP and company servers. They are always connected to the Internet. This difference means that a user agent in Boston may need to contact its regular mail server in Seattle to send a mail message because the user is traveling.

By itself, this remote communication poses no problem. It is exactly what the TCP/IP protocols are designed to support. However, an ISP or company usually does not want any remote user to be able to submit messages to its mail server to be delivered elsewhere. The ISP or company is not running the server as a public service. In addition, this kind of **open mail relay** attracts spammers. This is because it provides a way to launder the original sender and thus make the message more difficult to identify as spam.

Given these considerations, SMTP is normally used for mail submission with the *AUTH* extension. This extension lets the server check the credentials (username and password) of the client to confirm that the server should be providing mail service.

There are several other differences in the way SMTP is used for mail submission. For example, port 587 is used in preference to port 25 and the SMTP server can check and correct the format of the messages sent by the user agent. For more information about the restricted use of SMTP for mail submission, please see RFC 4409.

Message Transfer

Once the sending mail transfer agent receives a message from the user agent, it will deliver it to the receiving mail transfer agent using SMTP. To do this, the sender uses the destination address. Consider the message in Fig. 7-15, addressed to *bob@ee.uwa.edu.au*. To what mail server should the message be delivered?

To determine the correct mail server to contact, DNS is consulted. In the previous section, we described how DNS contains multiple types of records, including the *MX*, or mail exchanger, record. In this case, a DNS query is made for the *MX* records of the domain *ee.uwa.edu.au*. This query returns an ordered list of the names and IP addresses of one or more mail servers.

The sending mail transfer agent then makes a TCP connection on port 25 to the IP address of the mail server to reach the receiving mail transfer agent, and uses SMTP to relay the message. The receiving mail transfer agent will then place mail for the user *bob* in the correct mailbox for Bob to read it at a later time. This local delivery step may involve moving the message among computers if there is a large mail infrastructure.

With this delivery process, mail travels from the initial to the final mail transfer agent in a single hop. There are no intermediate servers in the message transfer stage. It is possible, however, for this delivery process to occur multiple times. One example that we have described already is when a message transfer agent

implements a mailing list. In this case, a message is received for the list. It is then expanded as a message to each member of the list that is sent to the individual member addresses.

As another example of relaying, Bob may have graduated from M.I.T. and also be reachable via the address *bob@alum.mit.edu*. Rather than reading mail on multiple accounts, Bob can arrange for mail sent to this address to be forwarded to *bob@ee.uwa.edu*. In this case, mail sent to *bob@alum.mit.edu* will undergo two deliveries. First, it will be sent to the mail server for *alum.mit.edu*. Then, it will be sent to the mail server for *ee.uwa.edu.au*. Each of these legs is a complete and separate delivery as far as the mail transfer agents are concerned.

Another consideration nowadays is spam. Nine out of ten messages sent today are spam (McAfee, 2010). Few people want more spam, but it is hard to avoid because it masquerades as regular mail. Before accepting a message, additional checks may be made to reduce the opportunities for spam. The message for Bob was sent from *alice@cs.washington.edu*. The receiving mail transfer agent can look up the sending mail transfer agent in DNS. This lets it check that the IP address of the other end of the TCP connection matches the DNS name. More generally, the receiving agent may look up the sending domain in DNS to see if it has a mail sending policy. This information is often given in the *TXT* and *SPF* records. It may indicate that other checks can be made. For example, mail sent from *cs.washington.edu* may always be sent from the host *june.cs.washington.edu*. If the sending mail transfer agent is not *june*, there is a problem.

If any of these checks fail, the mail is probably being forged with a fake sending address. In this case, it is discarded. However, passing these checks does not imply that mail is not spam. The checks merely ensure that the mail seems to be coming from the region of the network that it purports to come from. The idea is that spammers should be forced to use the correct sending address when they send mail. This makes spam easier to recognize and delete when it is unwanted.

7.2.5 Final Delivery

Our mail message is almost delivered. It has arrived at Bob's mailbox. All that remains is to transfer a copy of the message to Bob's user agent for display. This is step 3 in the architecture of Fig. 7-7. This task was straightforward in the early Internet, when the user agent and mail transfer agent ran on the same machine as different processes. The mail transfer agent simply wrote new messages to the end of the mailbox file, and the user agent simply checked the mailbox file for new mail.

Nowadays, the user agent on a PC, laptop, or mobile, is likely to be on a different machine than the ISP or company mail server. Users want to be able to access their mail remotely, from wherever they are. They want to access email from work, from their home PCs, from their laptops when on business trips, and from cybercafes when on so-called vacation. They also want to be able to work offline,

then reconnect to receive incoming mail and send outgoing mail. Moreover, each user may run several user agents depending on what computer it is convenient to use at the moment. Several user agents may even be running at the same time.

In this setting, the job of the user agent is to present a view of the contents of the mailbox, and to allow the mailbox to be remotely manipulated. Several different protocols can be used for this purpose, but SMTP is not one of them. SMTP is a push-based protocol. It takes a message and connects to a remote server to transfer the message. Final delivery cannot be achieved in this manner both because the mailbox must continue to be stored on the mail transfer agent and because the user agent may not be connected to the Internet at the moment that SMTP attempts to relay messages.

IMAP—The Internet Message Access Protocol

One of the main protocols that is used for final delivery is **IMAP** (**Internet Message Access Protocol**). Version 4 of the protocol is defined in RFC 3501. To use IMAP, the mail server runs an IMAP server that listens to port 143. The user agent runs an IMAP client. The client connects to the server and begins to issue commands from those listed in Fig. 7-17.

First, the client will start a secure transport if one is to be used (in order to keep the messages and commands confidential), and then log in or otherwise authenticate itself to the server. Once logged in, there are many commands to list folders and messages, fetch messages or even parts of messages, mark messages with flags for later deletion, and organize messages into folders. To avoid confusion, please note that we use the term "folder" here to be consistent with the rest of the material in this section, in which a user has a single mailbox made up of multiple folders. However, in the IMAP specification, the term *mailbox* is used instead. One user thus has many IMAP mailboxes, each of which is typically presented to the user as a folder.

IMAP has many other features, too. It has the ability to address mail not by message number, but by using attributes (e.g., give me the first message from Alice). Searches can be performed on the server to find the messages that satisfy certain criteria so that only those messages are fetched by the client.

IMAP is an improvement over an earlier final delivery protocol, **POP3** (**Post Office Protocol, version 3**), which is specified in RFC 1939. POP3 is a simpler protocol but supports fewer features and is less secure in typical usage. Mail is usually downloaded to the user agent computer, instead of remaining on the mail server. This makes life easier on the server, but harder on the user. It is not easy to read mail on multiple computers, plus if the user agent computer breaks, all email may be lost permanently. Nonetheless, you will still find POP3 in use.

Proprietary protocols can also be used because the protocol runs between a mail server and user agent that can be supplied by the same company. Microsoft Exchange is a mail system with a proprietary protocol.

Command	Description
CAPABILITY	List server capabilities
STARTTLS	Start secure transport (TLS; see Chap. 8)
LOGIN	Log on to server
AUTHENTICATE	Log on with other method
SELECT	Select a folder
EXAMINE	Select a read-only folder
CREATE	Create a folder
DELETE	Delete a folder
RENAME	Rename a folder
SUBSCRIBE	Add folder to active set
UNSUBSCRIBE	Remove folder from active set
LIST	List the available folders
LSUB	List the active folders
STATUS	Get the status of a folder
APPEND	Add a message to a folder
CHECK	Get a checkpoint of a folder
FETCH	Get messages from a folder
SEARCH	Find messages in a folder
STORE	Alter message flags
COPY	Make a copy of a message in a folder
EXPUNGE	Remove messages flagged for deletion
UID	Issue commands using unique identifiers
NOOP	Do nothing
CLOSE	Remove flagged messages and close folder
LOGOUT	Log out and close connection

Figure 7-17. IMAP (version 4) commands.

Webmail

An increasingly popular alternative to IMAP and SMTP for providing email service is to use the Web as an interface for sending and receiving mail. Widely used **Webmail** systems include Google Gmail, Microsoft Hotmail and Yahoo! Mail. Webmail is one example of software (in this case, a mail user agent) that is provided as a service using the Web.

In this architecture, the provider runs mail servers as usual to accept messages for users with SMTP on port 25. However, the user agent is different. Instead of

being a standalone program, it is a user interface that is provided via Web pages. This means that users can use any browser they like to access their mail and send new messages.

We have not yet studied the Web, but a brief description that you might come back to is as follows. When the user goes to the email Web page of the provider, a form is presented in which the user is asked for a login name and password. The login name and password are sent to the server, which then validates them. If the login is successful, the server finds the user's mailbox and builds a Web page listing the contents of the mailbox on the fly. The Web page is then sent to the browser for display.

Many of the items on the page showing the mailbox are clickable, so messages can be read, deleted, and so on. To make the interface responsive, the Web pages will often include JavaScript programs. These programs are run locally on the client in response to local events (e.g., mouse clicks) and can also download and upload messages in the background, to prepare the next message for display or a new message for submission. In this model, mail submission happens using the normal Web protocols by posting data to a URL. The Web server takes care of injecting messages into the traditional mail delivery system that we have described. For security, the standard Web protocols can be used as well. These protocols concern themselves with encrypting Web pages, not whether the content of the Web page is a mail message.

7.3 THE WORLD WIDE WEB

The Web, as the World Wide Web is popularly known, is an architectural framework for accessing linked content spread out over millions of machines all over the Internet. In 10 years it went from being a way to coordinate the design of high-energy physics experiments in Switzerland to the application that millions of people think of as being "The Internet." Its enormous popularity stems from the fact that it is easy for beginners to use and provides access with a rich graphical interface to an enormous wealth of information on almost every conceivable subject, from aardvarks to Zulus.

The Web began in 1989 at CERN, the European Center for Nuclear Research. The initial idea was to help large teams, often with members in half a dozen or more countries and time zones, collaborate using a constantly changing collection of reports, blueprints, drawings, photos, and other documents produced by experiments in particle physics. The proposal for a web of linked documents came from CERN physicist Tim Berners-Lee. The first (text-based) prototype was operational 18 months later. A public demonstration given at the Hypertext '91 conference caught the attention of other researchers, which led Marc Andreessen at the University of Illinois to develop the first graphical browser. It was called Mosaic and released in February 1993.

The rest, as they say, is now history. Mosaic was so popular that a year later Andreessen left to form a company, Netscape Communications Corp., whose goal was to develop Web software. For the next three years, Netscape Navigator and Microsoft's Internet Explorer engaged in a "browser war," each one trying to capture a larger share of the new market by frantically adding more features (and thus more bugs) than the other one.

Through the 1990s and 2000s, Web sites and Web pages, as Web content is called, grew exponentially until there were millions of sites and billions of pages. A small number of these sites became tremendously popular. Those sites and the companies behind them largely define the Web as people experience it today. Examples include: a bookstore (Amazon, started in 1994, market capitalization $50 billion), a flea market (eBay, 1995, $30B), search (Google, 1998, $150B), and social networking (Facebook, 2004, private company valued at more than $15B). The period through 2000, when many Web companies became worth hundreds of millions of dollars overnight, only to go bust practically the next day when they turned out to be hype, even has a name. It is called the **dot com era**. New ideas are still striking it rich on the Web. Many of them come from students. For example, Mark Zuckerberg was a Harvard student when he started Facebook, and Sergey Brin and Larry Page were students at Stanford when they started Google. Perhaps you will come up with the next big thing.

In 1994, CERN and M.I.T. signed an agreement setting up the **W3C** (**World Wide Web Consortium**), an organization devoted to further developing the Web, standardizing protocols, and encouraging interoperability between sites. Berners-Lee became the director. Since then, several hundred universities and companies have joined the consortium. Although there are now more books about the Web than you can shake a stick at, the best place to get up-to-date information about the Web is (naturally) on the Web itself. The consortium's home page is at *www.w3.org*. Interested readers are referred there for links to pages covering all of the consortium's numerous documents and activities.

7.3.1 Architectural Overview

From the users' point of view, the Web consists of a vast, worldwide collection of content in the form of **Web pages**, often just called **pages** for short. Each page may contain links to other pages anywhere in the world. Users can follow a link by clicking on it, which then takes them to the page pointed to. This process can be repeated indefinitely. The idea of having one page point to another, now called **hypertext**, was invented by a visionary M.I.T. professor of electrical engineering, Vannevar Bush, in 1945 (Bush, 1945). This was long before the Internet was invented. In fact, it was before commercial computers existed although several universities had produced crude prototypes that filled large rooms and had less power than a modern pocket calculator.

Pages are generally viewed with a program called a **browser**. Firefox, Internet Explorer, and Chrome are examples of popular browsers. The browser fetches the page requested, interprets the content, and displays the page, properly formatted, on the screen. The content itself may be a mix of text, images, and formatting commands, in the manner of a traditional document, or other forms of content such as video or programs that produce a graphical interface with which users can interact.

A picture of a page is shown on the top-left side of Fig. 7-18. It is the page for the Computer Science & Engineering department at the University of Washington. This page shows text and graphical elements (that are mostly too small to read). Some parts of the page are associated with links to other pages. A piece of text, icon, image, and so on associated with another page is called a **hyperlink**. To follow a link, the user places the mouse cursor on the linked portion of the page area (which causes the cursor to change shape) and clicks. Following a link is simply a way of telling the browser to fetch another page. In the early days of the Web, links were highlighted with underlining and colored text so that they would stand out. Nowadays, the creators of Web pages have ways to control the look of linked regions, so a link might appear as an icon or change its appearance when the mouse passes over it. It is up to the creators of the page to make the links visually distinct, to provide a usable interface.

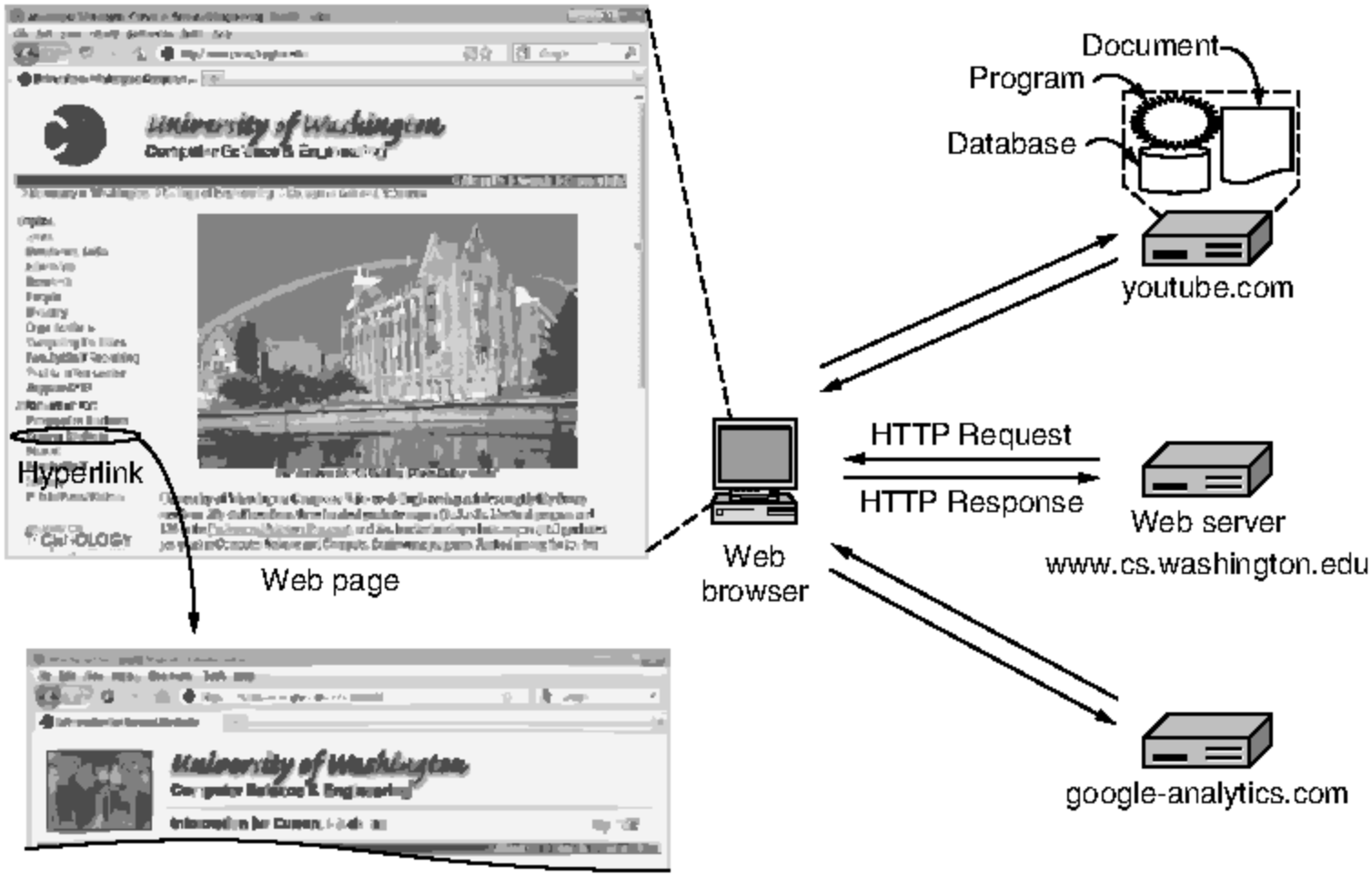


Figure 7-18. Architecture of the Web.

Students in the department can learn more by following a link to a page with information especially for them. This link is accessed by clicking in the circled area. The browser then fetches the new page and displays it, as partially shown in the bottom left of Fig. 7-18. Dozens of other pages are linked off the first page besides this example. Every other page can be comprised of content on the same machine(s) as the first page, or on machines halfway around the globe. The user cannot tell. Page fetching is done by the browser, without any help from the user. Thus, moving between machines while viewing content is seamless.

The basic model behind the display of pages is also shown in Fig. 7-18. The browser is displaying a Web page on the client machine. Each page is fetched by sending a request to one or more servers, which respond with the contents of the page. The request-response protocol for fetching pages is a simple text-based protocol that runs over TCP, just as was the case for SMTP. It is called **HTTP** (**HyperText Transfer Protocol**). The content may simply be a document that is read off a disk, or the result of a database query and program execution. The page is a **static page** if it is a document that is the same every time it is displayed. In contrast, if it was generated on demand by a program or contains a program it is a **dynamic page**.

A dynamic page may present itself differently each time it is displayed. For example, the front page for an electronic store may be different for each visitor. If a bookstore customer has bought mystery novels in the past, upon visiting the store's main page, the customer is likely to see new thrillers prominently displayed, whereas a more culinary-minded customer might be greeted with new cookbooks. How the Web site keeps track of who likes what is a story to be told shortly. But briefly, the answer involves cookies (even for culinarily challenged visitors).

In the figure, the browser contacts three servers to fetch the two pages, *cs.washington.edu*, *youtube.com*, and *google-analytics.com*. The content from these different servers is integrated for display by the browser. Display entails a range of processing that depends on the kind of content. Besides rendering text and graphics, it may involve playing a video or running a script that presents its own user interface as part of the page. In this case, the *cs.washington.edu* server supplies the main page, the *youtube.com* server supplies an embedded video, and the *google-analytics.com* server supplies nothing that the user can see but tracks visitors to the site. We will have more to say about trackers later.

The Client Side

Let us now examine the Web browser side in Fig. 7-18 in more detail. In essence, a browser is a program that can display a Web page and catch mouse clicks to items on the displayed page. When an item is selected, the browser follows the hyperlink and fetches the page selected.

When the Web was first created, it was immediately apparent that having one page point to another Web page required mechanisms for naming and locating pages. In particular, three questions had to be answered before a selected page could be displayed:

1. What is the page called?
2. Where is the page located?
3. How can the page be accessed?

If every page were somehow assigned a unique name, there would not be any ambiguity in identifying pages. Nevertheless, the problem would not be solved. Consider a parallel between people and pages. In the United States, almost everyone has a social security number, which is a unique identifier, as no two people are supposed to have the same one. Nevertheless, if you are armed only with a social security number, there is no way to find the owner's address, and certainly no way to tell whether you should write to the person in English, Spanish, or Chinese. The Web has basically the same problems.

The solution chosen identifies pages in a way that solves all three problems at once. Each page is assigned a **URL** (**Uniform Resource Locator**) that effectively serves as the page's worldwide name. URLs have three parts: the protocol (also known as the **scheme**), the DNS name of the machine on which the page is located, and the path uniquely indicating the specific page (a file to read or program to run on the machine). In the general case, the path has a hierarchical name that models a file directory structure. However, the interpretation of the path is up to the server; it may or may not reflect the actual directory structure.

As an example, the URL of the page shown in Fig. 7-18 is

http://www.cs.washington.edu/index.html

This URL consists of three parts: the protocol (*http*), the DNS name of the host (*www.cs.washington.edu*), and the path name (*index.html*).

When a user clicks on a hyperlink, the browser carries out a series of steps in order to fetch the page pointed to. Let us trace the steps that occur when our example link is selected:

1. The browser determines the URL (by seeing what was selected).
2. The browser asks DNS for the IP address of the server *www.cs.washington.edu*.
3. DNS replies with 128.208.3.88.
4. The browser makes a TCP connection to 128.208.3.88 on port 80, the well-known port for the HTTP protocol.
5. It sends over an HTTP request asking for the page */index.html*.

6. The *www.cs.washington.edu* server sends the page as an HTTP response, for example, by sending the file */index.html*.
7. If the page includes URLs that are needed for display, the browser fetches the other URLs using the same process. In this case, the URLs include multiple embedded images also fetched from *www.cs.washington.edu*, an embedded video from *youtube.com*, and a script from *google-analytics.com*.
8. The browser displays the page */index.html* as it appears in Fig. 7-18.
9. The TCP connections are released if there are no other requests to the same servers for a short period.

Many browsers display which step they are currently executing in a status line at the bottom of the screen. In this way, when the performance is poor, the user can see if it is due to DNS not responding, a server not responding, or simply page transmission over a slow or congested network.

The URL design is open-ended in the sense that it is straightforward to have browsers use multiple protocols to get at different kinds of resources. In fact, URLs for various other protocols have been defined. Slightly simplified forms of the common ones are listed in Fig. 7-19.

Name	Used for	Example
http	Hypertext (HTML)	http://www.ee.uwa.edu/~rob/
https	Hypertext with security	https://www.bank.com/accounts/
ftp	FTP	ftp://ftp.cs.vu.nl/pub/minix/README
file	Local file	file:///usr/suzanne/prog.c
mailto	Sending email	mailto:JohnUser@acm.org
rtsp	Streaming media	rtsp://youtube.com/montypython.mpg
sip	Multimedia calls	sip:eve@adversary.com
about	Browser information	about:plugins

Figure 7-19. Some common URL schemes.

Let us briefly go over the list. The *http* protocol is the Web's native language, the one spoken by Web servers. **HTTP** stands for **HyperText Transfer Protocol**. We will examine it in more detail later in this section.

The *ftp* protocol is used to access files by FTP, the Internet's file transfer protocol. FTP predates the Web and has been in use for more than three decades. The Web makes it easy to obtain files placed on numerous FTP servers throughout the world by providing a simple, clickable interface instead of a command-line interface. This improved access to information is one reason for the spectacular growth of the Web.

It is possible to access a local file as a Web page by using the *file* protocol, or more simply, by just naming it. This approach does not require having a server. Of course, it works only for local files, not remote ones.

The *mailto* protocol does not really have the flavor of fetching Web pages, but is useful anyway. It allows users to send email from a Web browser. Most browsers will respond when a *mailto* link is followed by starting the user's mail agent to compose a message with the address field already filled in.

The *rtsp* and *sip* protocols are for establishing streaming media sessions and audio and video calls.

Finally, the *about* protocol is a convention that provides information about the browser. For example, following the *about:plugins* link will cause most browsers to show a page that lists the MIME types that they handle with browser extensions called plug-ins.

In short, the URLs have been designed not only to allow users to navigate the Web, but to run older protocols such as FTP and email as well as newer protocols for audio and video, and to provide convenient access to local files and browser information. This approach makes all the specialized user interface programs for those other services unnecessary and integrates nearly all Internet access into a single program: the Web browser. If it were not for the fact that this idea was thought of by a British physicist working a research lab in Switzerland, it could easily pass for a plan dreamed up by some software company's advertising department.

Despite all these nice properties, the growing use of the Web has turned up an inherent weakness in the URL scheme. A URL points to one specific host, but sometimes it is useful to reference a page without simultaneously telling where it is. For example, for pages that are heavily referenced, it is desirable to have multiple copies far apart, to reduce the network traffic. There is no way to say: "I want page *xyz*, but I do not care where you get it."

To solve this kind of problem, URLs have been generalized into **URIs** (**Uniform Resource Identifiers**). Some URIs tell how to locate a resource. These are the URLs. Other URIs tell the name of a resource but not where to find it. These URIs are called **URNs** (**Uniform Resource Names**). The rules for writing URIs are given in RFC 3986, while the different URI schemes in use are tracked by IANA. There are many different kinds of URIs besides the schemes listed in Fig. 7-19, but those schemes dominate the Web as it is used today.

MIME Types

To be able to display the new page (or any page), the browser has to understand its format. To allow all browsers to understand all Web pages, Web pages are written in a standardized language called HTML. It is the lingua franca of the Web (for now). We will discuss it in detail later in this chapter.

Although a browser is basically an HTML interpreter, most browsers have numerous buttons and features to make it easier to navigate the Web. Most have a button for going back to the previous page, a button for going forward to the next page (only operative after the user has gone back from it), and a button for going straight to the user's preferred start page. Most browsers have a button or menu item to set a bookmark on a given page and another one to display the list of bookmarks, making it possible to revisit any of them with only a few mouse clicks.

As our example shows, HTML pages can contain rich content elements and not simply text and hypertext. For added generality, not all pages need contain HTML. A page may consist of a video in MPEG format, a document in PDF format, a photograph in JPEG format, a song in MP3 format, or any one of hundreds of other file types. Since standard HTML pages may link to any of these, the browser has a problem when it hits a page it does not know how to interpret.

Rather than making the browsers larger and larger by building in interpreters for a rapidly growing collection of file types, most browsers have chosen a more general solution. When a server returns a page, it also returns some additional information about the page. This information includes the MIME type of the page (see Fig. 7-13). Pages of type *text/html* are just displayed directly, as are pages in a few other built-in types. If the MIME type is not one of the built-in ones, the browser consults its table of MIME types to determine how to display the page. This table associates MIME types with viewers.

There are two possibilities: plug-ins and helper applications. A **plug-in** is a third-party code module that is installed as an extension to the browser, as illustrated in Fig. 7-20(a). Common examples are plug-ins for PDF, Flash, and Quicktime to render documents and play audio and video. Because plug-ins run inside the browser, they have access to the current page and can modify its appearance.

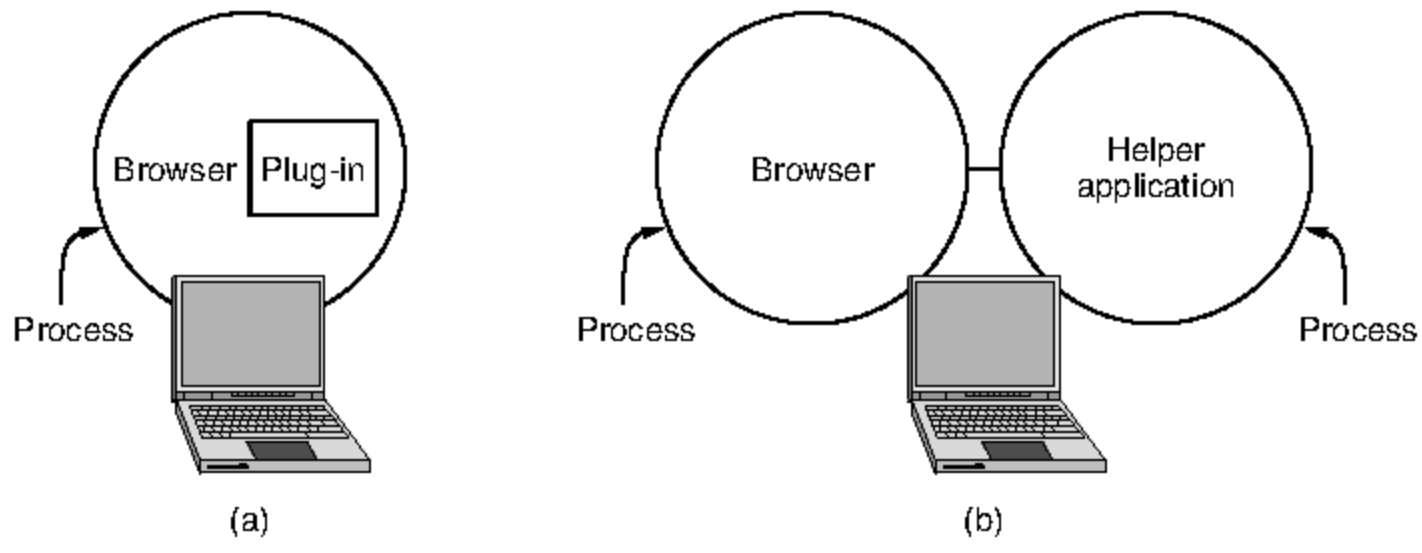


Figure 7-20. (a) A browser plug-in. (b) A helper application.

Each browser has a set of procedures that all plug-ins must implement so the browser can call the plug-ins. For example, there is typically a procedure the

browser's base code calls to supply the plug-in with data to display. This set of procedures is the plug-in's interface and is browser specific.

In addition, the browser makes a set of its own procedures available to the plug-in, to provide services to plug-ins. Typical procedures in the browser interface are for allocating and freeing memory, displaying a message on the browser's status line, and querying the browser about parameters.

Before a plug-in can be used, it must be installed. The usual installation procedure is for the user to go to the plug-in's Web site and download an installation file. Executing the installation file unpacks the plug-in and makes the appropriate calls to register the plug-in's MIME type with the browser and associate the plug-in with it. Browsers usually come preloaded with popular plug-ins.

The other way to extend a browser is make use of a **helper application**. This is a complete program, running as a separate process. It is illustrated in Fig. 7-20(b). Since the helper is a separate program, the interface is at arm's length from the browser. It usually just accepts the name of a scratch file where the content file has been stored, opens the file, and displays the contents. Typically, helpers are large programs that exist independently of the browser, for example, Microsoft Word or PowerPoint.

Many helper applications use the MIME type *application*. As a consequence, a considerable number of subtypes have been defined for them to use, for example, *application/vnd.ms-powerpoint* for PowerPoint files. *vnd* denotes vendor-specific formats. In this way, a URL can point directly to a PowerPoint file, and when the user clicks on it, PowerPoint is automatically started and handed the content to be displayed. Helper applications are not restricted to using the *application* MIME type.. Adobe Photoshop uses *image/x-photoshop*, for example.

Consequently, browsers can be configured to handle a virtually unlimited number of document types with no changes to themselves. Modern Web servers are often configured with hundreds of type/subtype combinations and new ones are often added every time a new program is installed.

A source of conflicts is that multiple plug-ins and helper applications are available for some subtypes, such as *video/mpeg*. What happens is that the last one to register overwrites the existing association with the MIME type, capturing the type for itself. As a consequence, installing a new program may change the way a browser handles existing types.

Browsers can also open local files, with no network in sight, rather than fetching them from remote Web servers. However, the browser needs some way to determine the MIME type of the file. The standard method is for the operating system to associate a file extension with a MIME type. In a typical configuration, opening *foo.pdf* will open it in the browser using an *application/pdf* plug-in and opening *bar.doc* will open it in Word as the *application/msword* helper.

Here, too, conflicts can arise, since many programs are willing—no, make that eager—to handle, say, mpg. During installation, programs intended for sophisticated users often display checkboxes for the MIME types and extensions

they are prepared to handle to allow the user to select the appropriate ones and thus not overwrite existing associations by accident. Programs aimed at the consumer market assume that the user does not have a clue what a MIME type is and simply grab everything they can without regard to what previously installed programs have done.

The ability to extend the browser with a large number of new types is convenient but can also lead to trouble. When a browser on a Windows PC fetches a file with the extension *exe*, it realizes that this file is an executable program and therefore has no helper. The obvious action is to run the program. However, this could be an enormous security hole. All a malicious Web site has to do is produce a Web page with pictures of, say, movie stars or sports heroes, all of which are linked to a virus. A single click on a picture then causes an unknown and potentially hostile executable program to be fetched and run on the user's machine. To prevent unwanted guests like this, Firefox and other browsers come configured to be cautious about running unknown programs automatically, but not all users understand what choices are safe rather than convenient.

The Server Side

So much for the client side. Now let us take a look at the server side. As we saw above, when the user types in a URL or clicks on a line of hypertext, the browser parses the URL and interprets the part between *http://* and the next slash as a DNS name to look up. Armed with the IP address of the server, the browser establishes a TCP connection to port 80 on that server. Then it sends over a command containing the rest of the URL, which is the path to the page on that server. The server then returns the page for the browser to display.

To a first approximation, a simple Web server is similar to the server of Fig. 6-6. That server is given the name of a file to look up and return via the network. In both cases, the steps that the server performs in its main loop are:

1. Accept a TCP connection from a client (a browser).
2. Get the path to the page, which is the name of the file requested.
3. Get the file (from disk).
4. Send the contents of the file to the client.
5. Release the TCP connection.

Modern Web servers have more features, but in essence, this is what a Web server does for the simple case of content that is contained in a file. For dynamic content, the third step may be replaced by the execution of a program (determined from the path) that returns the contents.

However, Web servers are implemented with a different design to serve many requests per second. One problem with the simple design is that accessing files is

often the bottleneck. Disk reads are very slow compared to program execution, and the same files may be read repeatedly from disk using operating system calls. Another problem is that only one request is processed at a time. The file may be large, and other requests will be blocked while it is transferred.

One obvious improvement (used by all Web servers) is to maintain a cache in memory of the n most recently read files or a certain number of gigabytes of content. Before going to disk to get a file, the server checks the cache. If the file is there, it can be served directly from memory, thus eliminating the disk access. Although effective caching requires a large amount of main memory and some extra processing time to check the cache and manage its contents, the savings in time are nearly always worth the overhead and expense.

To tackle the problem of serving a single request at a time, one strategy is to make the server **multithreaded**. In one design, the server consists of a front-end module that accepts all incoming requests and k processing modules, as shown in Fig. 7-21. The $k + 1$ threads all belong to the same process, so the processing modules all have access to the cache within the process' address space. When a request comes in, the front end accepts it and builds a short record describing it. It then hands the record to one of the processing modules.

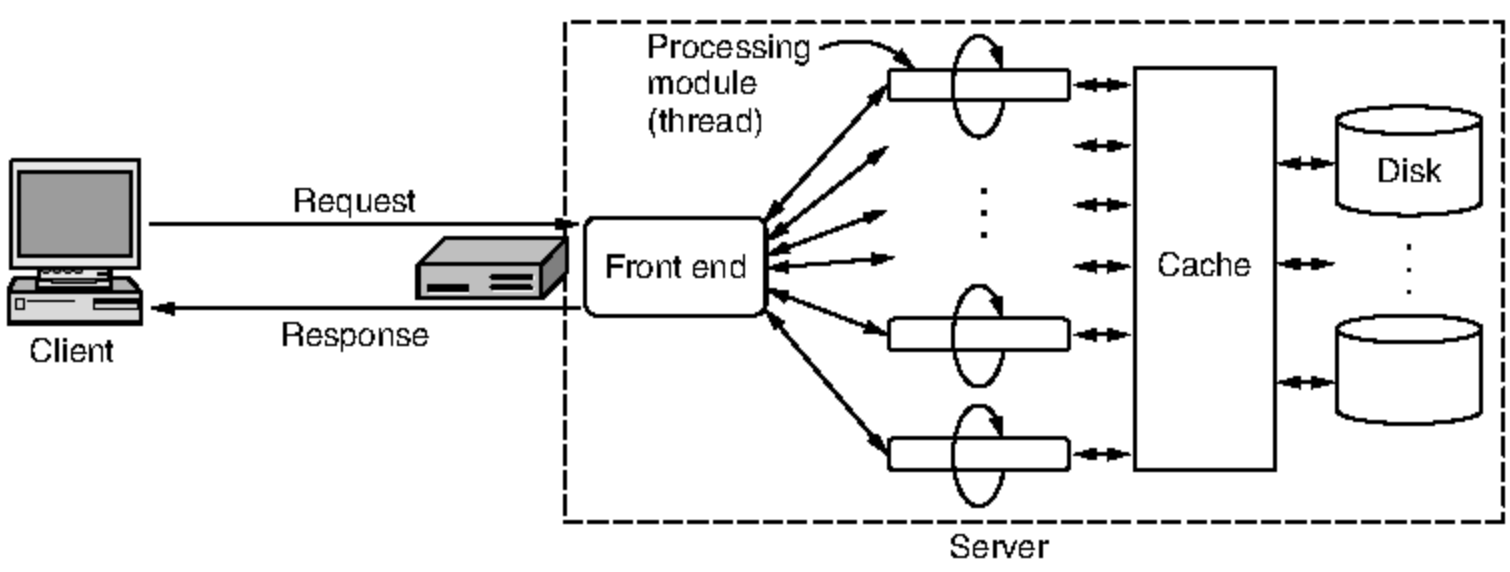


Figure 7-21. A multithreaded Web server with a front end and processing modules.

The processing module first checks the cache to see if the file needed is there. If so, it updates the record to include a pointer to the file in the record. If it is not there, the processing module starts a disk operation to read it into the cache (possibly discarding some other cached file(s) to make room for it). When the file comes in from the disk, it is put in the cache and also sent back to the client.

The advantage of this scheme is that while one or more processing modules are blocked waiting for a disk or network operation to complete (and thus consuming no CPU time), other modules can be actively working on other requests. With k processing modules, the throughput can be as much as k times higher than with a single-threaded server. Of course, when the disk or network is the limiting

factor, it is necessary to have multiple disks or a faster network to get any real improvement over the single-threaded model.

Modern Web servers do more than just accept path names and return files. In fact, the actual processing of each request can get quite complicated. For this reason, in many servers each processing module performs a series of steps. The front end passes each incoming request to the first available module, which then carries it out using some subset of the following steps, depending on which ones are needed for that particular request. These steps occur after the TCP connection and any secure transport mechanism (such as SSL/TLS, which will be described in Chap. 8) have been established.

1. Resolve the name of the Web page requested.
2. Perform access control on the Web page.
3. Check the cache.
4. Fetch the requested page from disk or run a program to build it.
5. Determine the rest of the response (e.g., the MIME type to send).
6. Return the response to the client.
7. Make an entry in the server log.

Step 1 is needed because the incoming request may not contain the actual name of a file or program as a literal string. It may contain built-in shortcuts that need to be translated. As a simple example, the URL *http://www.cs.vu.nl/* has an empty file name. It has to be expanded to some default file name that is usually *index.html*. Another common rule is to map *~user/* onto *user*'s Web directory. These rules can be used together. Thus, the home page of one of the authors (AST) can be reached at

http://www.cs.vu.nl/~ast/

even though the actual file name is *index.html* in a certain default directory.

Also, modern browsers can specify configuration information such as the browser software and the user's default language (e.g., Italian or English). This makes it possible for the server to select a Web page with small pictures for a mobile device and in the preferred language, if available. In general, name expansion is not quite so trivial as it might at first appear, due to a variety of conventions about how to map paths to the file directory and programs.

Step 2 checks to see if any access restrictions associated with the page are met. Not all pages are available to the general public. Determining whether a client can fetch a page may depend on the identity of the client (e.g., as given by usernames and passwords) or the location of the client in the DNS or IP space. For example, a page may be restricted to users inside a company. How this is

accomplished depends on the design of the server. For the popular Apache server, for instance, the convention is to place a file called *.htaccess* that lists the access restrictions in the directory where the restricted page is located.

Steps 3 and 4 involve getting the page. Whether it can be taken from the cache depends on processing rules. For example, pages that are created by running programs cannot always be cached because they might produce a different result each time they are run. Even files should occasionally be checked to see if their contents have changed so that the old contents can be removed from the cache. If the page requires a program to be run, there is also the issue of setting the program parameters or input. These data come from the path or other parts of the request.

Step 5 is about determining other parts of the response that accompany the contents of the page. The MIME type is one example. It may come from the file extension, the first few words of the file or program output, a configuration file, and possibly other sources.

Step 6 is returning the page across the network. To increase performance, a single TCP connection may be used by a client and server for multiple page fetches. This reuse means that some logic is needed to map a request to a shared connection and to return each response so that it is associated with the correct request.

Step 7 makes an entry in the system log for administrative purposes, along with keeping any other important statistics. Such logs can later be mined for valuable information about user behavior, for example, the order in which people access the pages.

Cookies

Navigating the Web as we have described it so far involves a series of independent page fetches. There is no concept of a login session. The browser sends a request to a server and gets back a file. Then the server forgets that it has ever seen that particular client.

This model is perfectly adequate for retrieving publicly available documents, and it worked well when the Web was first created. However, it is not suited for returning different pages to different users depending on what they have already done with the server. This behavior is needed for many ongoing interactions with Web sites. For example, some Web sites (e.g., newspapers) require clients to register (and possibly pay money) to use them. This raises the question of how servers can distinguish between requests from users who have previously registered and everyone else. A second example is from e-commerce. If a user wanders around an electronic store, tossing items into her virtual shopping cart from time to time, how does the server keep track of the contents of the cart? A third example is customized Web portals such as Yahoo!. Users can set up a personalized

detailed initial page with only the information they want (e.g., their stocks and their favorite sports teams), but how can the server display the correct page if it does not know who the user is?

At first glance, one might think that servers could track users by observing their IP addresses. However, this idea does not work. Many users share computers, especially at home, and the IP address merely identifies the computer, not the user. Even worse, many companies use NAT, so that outgoing packets bear the same IP address for all users. That is, all of the computers behind the NAT box look the same to the server. And many ISPs assign IP addresses to customers with DHCP. The IP addresses change over time, so to a server you might suddenly look like your neighbor. For all of these reasons, the server cannot use IP addresses to track users.

This problem is solved with an oft-critized mechanism called **cookies**. The name derives from ancient programmer slang in which a program calls a procedure and gets something back that it may need to present later to get some work done. In this sense, a UNIX file descriptor or a Windows object handle can be considered to be a cookie. Cookies were first implemented in the Netscape browser in 1994 and are now specified in RFC 2109.

When a client requests a Web page, the server can supply additional information in the form of a cookie along with the requested page. The cookie is a rather small, named string (of at most 4 KB) that the server can associate with a browser. This association is not the same thing as a user, but it is much closer and more useful than an IP address. Browsers store the offered cookies for an interval, usually in a cookie directory on the client's disk so that the cookies persist across browser invocations, unless the user has disabled cookies. Cookies are just strings, not executable programs. In principle, a cookie could contain a virus, but since cookies are treated as data, there is no official way for the virus to actually run and do damage. However, it is always possible for some hacker to exploit a browser bug to cause activation.

A cookie may contain up to five fields, as shown in Fig. 7-22. The *Domain* tells where the cookie came from. Browsers are supposed to check that servers are not lying about their domain. Each domain should store no more than 20 cookies per client. The *Path* is a path in the server's directory structure that identifies which parts of the server's file tree may use the cookie. It is often /, which means the whole tree.

The *Content* field takes the form *name* = *value*. Both *name* and *value* can be anything the server wants. This field is where the cookie's content is stored.

The *Expires* field specifies when the cookie expires. If this field is absent, the browser discards the cookie when it exits. Such a cookie is called a **nonpersistent cookie**. If a time and date are supplied, the cookie is said to be a **persistent cookie** and is kept until it expires. Expiration times are given in Greenwich Mean Time. To remove a cookie from a client's hard disk, a server just sends it again, but with an expiration time in the past.

Domain	Path	Content	Expires	Secure
toms-casino.com	/	CustomerID=297793521	15-10-10 17:00	Yes
jills-store.com	/	Cart=1-00501;1-07031;2-13721	11-1-11 14:22	No
aportal.com	/	Prefs=Stk:CSCO+ORCL;Spt:Jets	31-12-20 23:59	No
sneaky.com	/	UserID=4627239101	31-12-19 23:59	No

Figure 7-22. Some examples of cookies.

Finally, the *Secure* field can be set to indicate that the browser may only return the cookie to a server using a secure transport, namely SSL/TLS (which we will describe in Chap. 8). This feature is used for e-commerce, banking, and other secure applications.

We have now seen how cookies are acquired, but how are they used? Just before a browser sends a request for a page to some Web site, it checks its cookie directory to see if any cookies there were placed by the domain the request is going to. If so, all the cookies placed by that domain, and only that domain, are included in the request message. When the server gets them, it can interpret them any way it wants to.

Let us examine some possible uses for cookies. In Fig. 7-22, the first cookie was set by *toms-casino.com* and is used to identify the customer. When the client returns next week to throw away some more money, the browser sends over the cookie so the server knows who it is. Armed with the customer ID, the server can look up the customer's record in a database and use this information to build an appropriate Web page to display. Depending on the customer's known gambling habits, this page might consist of a poker hand, a listing of today's horse races, or a slot machine.

The second cookie came from *jills-store.com*. The scenario here is that the client is wandering around the store, looking for good things to buy. When she finds a bargain and clicks on it, the server adds it to her shopping cart (maintained on the server) and also builds a cookie containing the product code of the item and sends the cookie back to the client. As the client continues to wander around the store by clicking on new pages, the cookie is returned to the server on every new page request. As more purchases accumulate, the server adds them to the cookie. Finally, when the client clicks on PROCEED TO CHECKOUT, the cookie, now containing the full list of purchases, is sent along with the request. In this way, the server knows exactly what the customer wants to buy.

The third cookie is for a Web portal. When the customer clicks on a link to the portal, the browser sends over the cookie. This tells the portal to build a page containing the stock prices for Cisco and Oracle, and the New York Jets' football results. Since a cookie can be up to 4 KB, there is plenty of room for more detailed preferences concerning newspaper headlines, local weather, special offers, etc.

A more controversial use of cookies is to track the online behavior of users. This lets Web site operators understand how users navigate their sites, and advertisers build up profiles of the ads or sites a particular user has viewed. The controversy is that users are typically unaware that their activity is being tracked, even with detailed profiles and across seemingly unrelated Web sites. Nonetheless, **Web tracking** is big business. DoubleClick, which provides and tracks ads, is ranked among the 100 busiest Web sites in the world by the Web monitoring company Alexa. Google Analytics, which tracks site usage for operators, is used by more than half of the busiest 100,000 sites on the Web.

It is easy for a server to track user activity with cookies. Suppose a server wants to keep track of how many unique visitors it has had and how many pages each visitor looked at before leaving the site. When the first request comes in, there will be no accompanying cookie, so the server sends back a cookie containing *Counter = 1*. Subsequent page views on that site will send the cookie back to the server. Each time the counter is incremented and sent back to the client. By keeping track of the counters, the server can see how many people give up after seeing the first page, how many look at two pages, and so on.

Tracking the browsing behavior of users across sites is only slightly more complicated. It works like this. An advertising agency, say, Sneaky Ads, contacts major Web sites and places ads for its clients' products on their pages, for which it pays the site owners a fee. Instead, of giving the sites the ad as a GIF file to place on each page, it gives them a URL to add to each page. Each URL it hands out contains a unique number in the path, such as

http://www.sneaky.com/382674902342.gif

When a user first visits a page, *P*, containing such an ad, the browser fetches the HTML file. Then the browser inspects the HTML file and sees the link to the image file at *www.sneaky.com*, so it sends a request there for the image. A GIF file containing an ad is returned, along with a cookie containing a unique user ID, 4627239101 in Fig. 7-22. Sneaky records the fact that the user with this ID visited page *P*. This is easy to do since the path requested (*382674902342.gif*) is referenced only on page *P*. Of course, the actual ad may appear on thousands of pages, but each time with a different name. Sneaky probably collects a fraction of a penny from the product manufacturer each time it ships out the ad.

Later, when the user visits another Web page containing any of Sneaky's ads, the browser first fetches the HTML file from the server. Then it sees the link to, say, *http://www.sneaky.com/193654919923.gif* on the page and requests that file. Since it already has a cookie from the domain *sneaky.com*, the browser includes Sneaky's cookie containing the user's ID. Sneaky now knows a second page the user has visited.

In due course, Sneaky can build up a detailed profile of the user's browsing habits, even though the user has never clicked on any of the ads. Of course, it does not yet have the user's name (although it does have his IP address, which

may be enough to deduce the name from other databases). However, if the user ever supplies his name to any site cooperating with Sneaky, a complete profile along with a name will be available for sale to anyone who wants to buy it. The sale of this information may be profitable enough for Sneaky to place more ads on more Web sites and thus collect more information.

And if Sneaky wants to be supersneaky, the ad need not be a classical banner ad. An "ad" consisting of a single pixel in the background color (and thus invisible) has exactly the same effect as a banner ad: it requires the browser to go fetch the 1×1-pixel GIF image and send it all cookies originating at the pixel's domain.

Cookies have become a focal point for the debate over online privacy because of tracking behavior like the above. The most insidious part of the whole business is that many users are completely unaware of this information collection and may even think they are safe because they do not click on any of the ads. For this reason, cookies that track users across sites are considered by many to be **spyware**. Have a look at the cookies that are already stored by your browser. Most browsers will display this information along with the current privacy preferences. You might be surprised to find names, email addresses, or passwords as well as opaque identifiers. Hopefully, you will not find credit card numbers, but the potential for abuse is clear.

To maintain a semblance of privacy, some users configure their browsers to reject all cookies. However, this can cause problems because many Web sites will not work properly without cookies. Alternatively, most browsers let users block **third-party cookies**. A third-party cookie is one from a different site than the main page that is being fetched, for example, the *sneaky.com* cookie that is used when interacting with page P on a completely different Web site. Blocking these cookies helps to prevent tracking across Web sites. Browser extensions can also be installed to provide fine-grained control over how cookies are used (or, rather, not used). As the debate continues, many companies are developing privacy policies that limit how they will share information to prevent abuse. Of course, the policies are simply how the companies say they will handle information. For example: "We may use the information collected from you in the conduct of our business"—which might be selling the information.

7.3.2 Static Web Pages

The basis of the Web is transferring Web pages from server to client. In the simplest form, Web pages are static. That is, they are just files sitting on some server that present themselves in the same way each time they are fetched and viewed. Just because they are static does not mean that the pages are inert at the browser, however. A page containing a video can be a static Web page.

As mentioned earlier, the lingua franca of the Web, in which most pages are written, is HTML. The home pages of teachers are usually static HTML pages.

The home pages of companies are usually dynamic pages put together by a Web design company. In this section, we will take a brief look at static HTML pages as a foundation for later material. Readers already familiar with HTML can skip ahead to the next section, where we describe dynamic content and Web services.

HTML—The HyperText Markup Language

HTML (**HyperText Markup Language**) was introduced with the Web. It allows users to produce Web pages that include text, graphics, video, pointers to other Web pages, and more. HTML is a markup language, or language for describing how documents are to be formatted. The term "markup" comes from the old days when copyeditors actually marked up documents to tell the printer—in those days, a human being—which fonts to use, and so on. Markup languages thus contain explicit commands for formatting. For example, in HTML, **means start boldface mode, and** means leave boldface mode. LaTeX and TeX are other examples of markup languages that are well known to most academic authors.

The key advantage of a markup language over one with no explicit markup is that it separates content from how it should be presented. Writing a browser is then straightforward: the browser simply has to understand the markup commands and apply them to the content. Embedding all the markup commands within each HTML file and standardizing them makes it possible for any Web browser to read and reformat any Web page. That is crucial because a page may have been produced in a 1600×1200 window with 24-bit color on a high-end computer but may have to be displayed in a 640×320 window on a mobile phone.

While it is certainly possible to write documents like this with any plain text editor, and many people do, it is also possible to use word processors or special HTML editors that do most of the work (but correspondingly give the user less direct control over the details of the final result).

A simple Web page written in HTML and its presentation in a browser are given in Fig. 7-23. A Web page consists of a head and a body, each enclosed by <html> and </html> **tags** (formatting commands), although most browsers do not complain if these tags are missing. As can be seen in Fig. 7-23(a), the head is bracketed by the <head> and </head> tags and the body is bracketed by the <body> and </body> tags. The strings inside the tags are called **directives**. Most, but not all, HTML tags have this format. That is, they use <something> to mark the beginning of something and </something> to mark its end.

Tags can be in either lowercase or uppercase. Thus, <head> and <HEAD> mean the same thing, but lower case is best for compatibility. Actual layout of the HTML document is irrelevant. HTML parsers ignore extra spaces and carriage returns since they have to reformat the text to make it fit the current display area. Consequently, white space can be added at will to make HTML documents more

readable, something most of them are badly in need of. As another consequence, blank lines cannot be used to separate paragraphs, as they are simply ignored. An explicit tag is required.

Some tags have (named) parameters, called **attributes**. For example, the <img> tag in Fig. 7-23 is used for including an image inline with the text. It has two attributes, src and alt. The first attribute gives the URL for the image. The HTML standard does not specify which image formats are permitted. In practice, all browsers support GIF and JPEG files. Browsers are free to support other formats, but this extension is a two-edged sword. If a user is accustomed to a browser that supports, say, TIFF files, he may include these in his Web pages and later be surprised when other browsers just ignore all of his wonderful art.

The second attribute gives alternate text to use if the image cannot be displayed. For each tag, the HTML standard gives a list of what the permitted parameters, if any, are, and what they mean. Because each parameter is named, the order in which the parameters are given is not significant.

Technically, HTML documents are written in the ISO 8859-1 Latin-1 character set, but for users whose keyboards support only ASCII, escape sequences are present for the special characters, such as è. The list of special characters is given in the standard. All of them begin with an ampersand and end with a semicolon. For example,   produces a space, è produces è and é produces é. Since <, >, and & have special meanings, they can be expressed only with their escape sequences, <, >, and &, respectively.

The main item in the head is the title, delimited by <title> and </title>. Certain kinds of metainformation may also be present, though none are present in our example. The title itself is not displayed on the page. Some browsers use it to label the page's window.

Several headings are used in Fig. 7-23. Each heading is generated by an <h*n*> tag, where *n* is a digit in the range 1 to 6. Thus, <h1> is the most important heading; <h6> is the least important one. It is up to the browser to render these appropriately on the screen. Typically, the lower-numbered headings will be displayed in a larger and heavier font. The browser may also choose to use different colors for each level of heading. Usually, <h1> headings are large and boldface with at least one blank line above and below. In contrast, <h2> headings are in a smaller font with less space above and below.

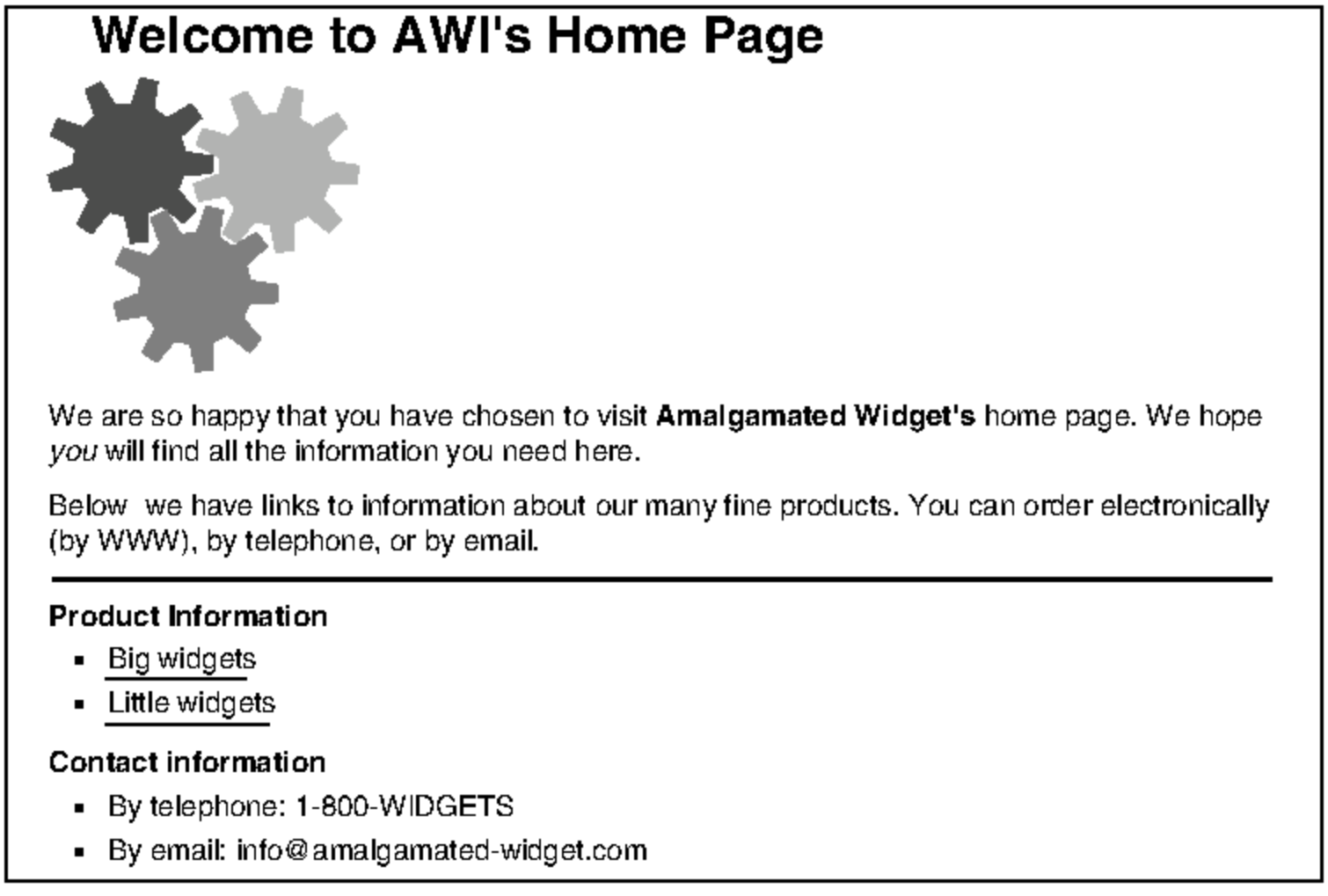
The tags <b> and <i> are used to enter boldface and italics mode, respectively. The <hr> tag forces a break and draws a horizontal line across the display.

The tag starts a paragraph. The browser might display this by inserting a blank line and some indentation, for example. Interestingly, the tag that exists to mark the end of a paragraph is often omitted by lazy HTML programmers.

HTML provides various mechanisms for making lists, including nested lists. Unordered lists, like the ones in Fig. 7-23 are started with <ul>, with <li> used to mark the start of items. There is also an <ol> tag to starts an ordered list. The

```
<html>
<head> <title> AMALGAMATED WIDGET, INC. </title> </head>
<body> <h1> Welcome to AWI's Home Page </h1>
<img src="http://www.widget.com/images/logo.gif" ALT="AWI Logo"> <br>
We are so happy that you have chosen to visit <b> Amalgamated Widget's</b>
home page. We hope <i> you </i> will find all the information you need here.
<p>Below we have links to information about our many fine products.
You can order electronically (by WWW), by telephone, or by email. </p>
<hr>
<h2> Product information </h2>
<ul>
   <li> <a href="http://widget.com/products/big"> Big widgets </a> </li>
   <li> <a href="http://widget.com/products/little"> Little widgets </a> </li>
</ul>
<h2> Contact information </h2>
<ul>
   <li> By telephone: 1-800-WIDGETS </li>
   <li> By email: info@amalgamated-widget.com </li>
</ul>
</body>
</html>
```

(a)



(b)

Figure 7-23. (a) The HTML for a sample Web page. (b) The formatted page.

individual items in unordered lists often appear with bullets (•) in front of them. Items in ordered lists are numbered by the browser.

Finally, we come to hyperlinks. Examples of these are seen in Fig. 7-23 using the <a> (anchor) and tags. The <a> tag has various parameters, the most important of which is *href* the linked URL. The text between the <a> and is displayed. If it is selected, the hyperlink is followed to a new page. It is also permitted to link other elements. For example, an image can be given between the <a> and tags using <img>. In this case, the image is displayed and clicking on it activates the hyperlink.

There are many other HTML tags and attributes that we have not seen in this simple example. For instance, the <a> tag can take a parameter *name* to plant a hyperlink, allowing a hyperlink to point to the middle of a page. This is useful, for example, for Web pages that start out with a clickable table of contents. By clicking on an item in the table of contents, the user jumps to the corresponding section of the same page. An example of a different tag is
. It forces the browser to break and start a new line.

Probably the best way to understand tags is to look at them in action. To do this, you can pick a Web page and look at the HTML in your browser to see how the page was put together. Most browsers have a VIEW SOURCE menu item (or something similar). Selecting this item displays the current page's HTML source, instead of its formatted output.

We have sketched the tags that have existed from the early Web. HTML keeps evolving. Fig. 7-24 shows some of the features that have been added with successive versions of HTML. HTML 1.0 refers to the version of HTML used with the introduction of the Web. HTML versions 2.0, 3.0, and 4.0 appeared in rapid succession in the space of only a few years as the Web exploded. After HTML 4.0, a period of almost ten years passed before the path to standarization of the next major version, HTML 5.0, became clear. Because it is a major upgrade that consolidates the ways that browsers handle rich content, the HTML 5.0 effort is ongoing and not expected to produce a standard before 2012 at the earliest. Standards notwithstanding, the major browsers already support HTML 5.0 functionality.

The progression through HTML versions is all about adding new features that people wanted but had to handle in nonstandard ways (e.g., plug-ins) until they became standard. For example, HTML 1.0 and HTML 2.0 did not have tables. They were added in HTML 3.0. An HTML table consists of one or more rows, each consisting of one or more table cells that can contain a wide range of material (e.g., text, images, other tables). Before HTML 3.0, authors needing a table had to resort to ad hoc methods, such as including an image showing the table.

In HTML 4.0, more new features were added. These included accessibility features for handicapped users, object embedding (a generalization of the <img> tag so other objects can also be embedded in pages), support for scripting languages (to allow dynamic content), and more.

Item	HTML 1.0	HTML 2.0	HTML 3.0	HTML 4.0	HTML 5.0
Hyperlinks	x	x	x	x	x
Images	x	x	x	x	x
Lists	x	x	x	x	x
Active maps & images		x	x	x	x
Forms		x	x	x	x
Equations			x	x	x
Toolbars			x	x	x
Tables			x	x	x
Accessibility features				x	x
Object embedding				x	x
Style sheets				x	x
Scripting				x	x
Video and audio					x
Inline vector graphics					x
XML representation					x
Background threads					x
Browser storage					x
Drawing canvas					x

Figure 7-24. Some differences between HTML versions.

HTML 5.0 includes many features to handle the rich media that are now routinely used on the Web. Video and audio can be included in pages and played by the browser without requiring the user to install plug-ins. Drawings can be built up in the browser as vector graphics, rather than using bitmap image formats (like JPEG and GIF) There is also more support for running scripts in browsers, such as background threads of computation and access to storage. All of these features help to support Web pages that are more like traditional applications with a user interface than documents. This is the direction the Web is heading.

Input and Forms

There is one important capability that we have not discussed yet: input. HTML 1.0 was basically one-way. Users could fetch pages from information providers, but it was difficult to send information back the other way. It quickly became apparent that there was a need for two-way traffic to allow orders for products to be placed via Web pages, registration cards to be filled out online, search terms to be entered, and much, much more.

Sending input from the user to the server (via the browser) requires two kinds of support. First, it requires that HTTP be able to carry data in that direction. We describe how this is done in a later section; it uses the *POST* method. The second requirement is to be able to present user interface elements that gather and package up the input. **Forms** were included with this functionality in HTML 2.0.

Forms contain boxes or buttons that allow users to fill in information or make choices and then send the information back to the page's owner. Forms are written just like other parts of HTML, as seen in the example of Fig. 7-25. Note that forms are still static content. They exhibit the same behavior regardless of who is using them. Dynamic content, which we will cover later, provides more sophisticated ways to gather input by sending a program whose behavior may depend on the browser environment.

Like all forms, this one is enclosed between the <form> and </form> tags. The attributes of this tag tell what to do with the data that are input, in this case using the *POST* method to send the data to the specified URL. Text not enclosed in a tag is just displayed. All the usual tags (e.g., <b>) are allowed in a form to let the author of the page control the look of the form on the screen.

Three kinds of input boxes are used in this form, each of which uses the <input> tag. It has a variety of parameters for determining the size, nature, and usage of the box displayed. The most common forms are blank fields for accepting user text, boxes that can be checked, and *submit* buttons that cause the data to be returned to the server.

The first kind of input box is a *text* box that follows the text "Name". The box is 46 characters wide and expects the user to type in a string, which is then stored in the variable *customer*.

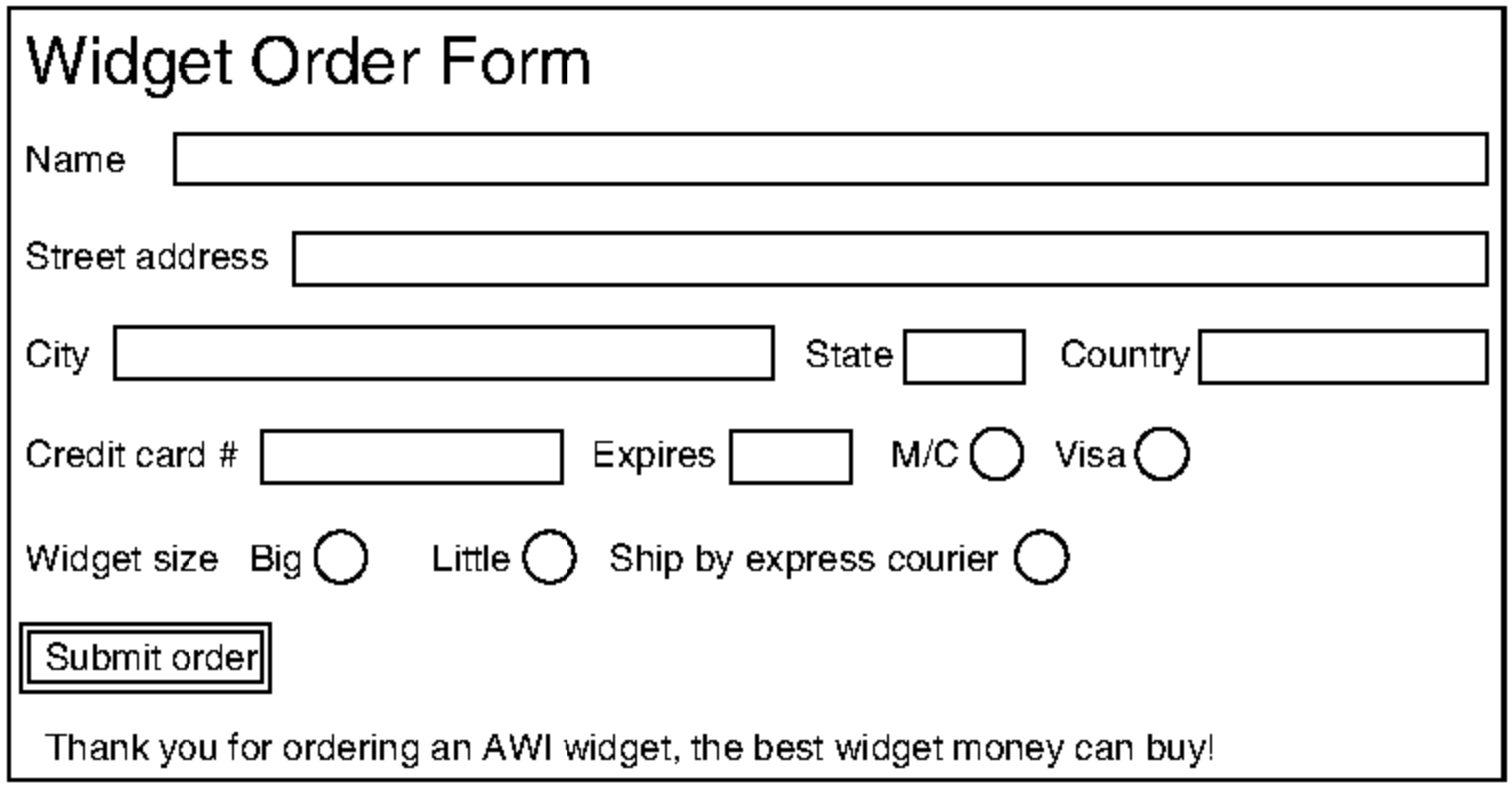
The next line of the form asks for the user's street address, 40 characters wide. Then comes a line asking for the city, state, and country. Since no <p> tags are used between these fields, the browser displays them all on one line (instead of as separate paragraphs) if they will fit. As far as the browser is concerned, the one paragraph contains just six items: three strings alternating with three boxes. The next line asks for the credit card number and expiration date. Transmitting credit card numbers over the Internet should only be done when adequate security measures have been taken. We will discuss some of these in Chap. 8.

Following the expiration date, we encounter a new feature: radio buttons. These are used when a choice must be made among two or more alternatives. The intellectual model here is a car radio with half a dozen buttons for choosing stations. Clicking on one button turns off all the other ones in the same group. The visual presentation is up to the browser. Widget size also uses two radio buttons. The two groups are distinguished by their *name* parameter, not by static scoping using something like <radiobutton> ... </radiobutton>.

The *value* parameters are used to indicate which radio button was pushed. For example, depending on which credit card options the user has chosen, the variable *cc* will be set to either the string "mastercard" or the string "visacard".

```
<html>
<head> <title> AWI CUSTOMER ORDERING FORM </title> </head>
<body>
<h1> Widget Order Form </h1>
<form ACTION="http://widget.com/cgi-bin/order.cgi" method=POST>
<p> Name <input name="customer" size=46> </p>
<p> Street address <input name="address" size=40> </p>
<p> City <input name="city" size=20> State <input name="state" size =4>
Country <input name="country" size=10> </p>
<p> Credit card # <input name="cardno" size=10>
Expires <input name="expires" size=4>
M/C <input name="cc" type=radio value="mastercard">
VISA <input name="cc" type=radio value="visacard"> </p>
<p> Widget size Big <input name="product" type=radio value="expensive">
Little <input name="product" type=radio value="cheap">
Ship by express courier <input name="express" type=checkbox> </p>
<p><input type=submit value="Submit order"> </p>
Thank you for ordering an AWI widget, the best widget money can buy!
</form>
</body>
</html>
```

(a)



(b)

Figure 7-25. (a) The HTML for an order form. (b) The formatted page.

After the two sets of radio buttons, we come to the shipping option, represented by a box of type *checkbox*. It can be either on or off. Unlike radio buttons, where exactly one out of the set must be chosen, each box of type *checkbox* can be on or off, independently of all the others.

Finally, we come to the *submit* button. The *value* string is the label on the button and is displayed. When the user clicks the *submit* button, the browser packages the collected information into a single long line and sends it back to the server to the URL provided as part of the <form> tag. A simple encoding is used. The & is used to separate fields and + is used to represent space. For our example form, the line might look like the contents of Fig. 7-26.

```
customer=John+Doe&address=100+Main+St.&city=White+Plains&
state=NY&country=USA&cardno=1234567890&expires=6/14&cc=mastercard&
product=cheap&express=on
```

Figure 7-26. A possible response from the browser to the server with information filled in by the user.

The string is sent back to the server as one line. (It is broken into three lines here because the page is not wide enough.) It is up to the server to make sense of this string, most likely by passing the information to a program that will process it. We will discuss how this can be done in the next section.

There are also other types of input that are not shown in this simple example. Two other types are *password* and *textarea*. A *password* box is the same as a *text* box (the default type that need not be named), except that the characters are not displayed as they are typed. A *textarea* box is also the same as a *text* box, except that it can contain multiple lines.

For long lists from which a choice must be made, the <select> and </select> tags are provided to bracket a list of alternatives. This list is often rendered as a drop-down menu. The semantics are those of radio buttons unless the *multiple* parameter is given, in which case the semantics are those of checkboxes.

Finally, there are ways to indicate default or initial values that the user can change. For example, if a *text* box is given a *value* field, the contents are displayed in the form for the user to edit or erase.

CSS—Cascading Style Sheets

The original goal of HTML was to specify the *structure* of the document, not its *appearance*. For example,

```
<h1> Deborah's Photos </h1>
```

instructs the browser to emphasize the heading, but does not say anything about the typeface, point size, or color. That is left up to the browser, which knows the properties of the display (e.g., how many pixels it has). However, many Web page designers wanted absolute control over how their pages appeared, so new tags were added to HTML to control appearance, such as

```
<font face="helvetica" size="24" color="red"> Deborah's Photos </font>
```

Also, ways were added to control positioning on the screen accurately. The trouble with this approach is that it is tedious and produces bloated HTML that is not portable. Although a page may render perfectly in the browser it is developed on, it may be a complete mess in another browser or another release of the same browser or at a different screen resolution.

A better alternative is the use of style sheets. Style sheets in text editors allow authors to associate text with a logical style instead of a physical style, for example, "initial paragraph" instead of "italic text." The appearance of each style is defined separately. In this way, if the author decides to change the initial paragraphs from 14-point italics in blue to 18-point boldface in shocking pink, all it requires is changing one definition to convert the entire document.

CSS (**Cascading Style Sheets**) introduced style sheets to the Web with HTML 4.0, though widespread use and browser support did not take off until 2000. CSS defines a simple language for describing rules that control the appearance of tagged content. Let us look at an example. Suppose that AWI wants snazzy Web pages with navy text in the Arial font on an off-white background, and level headings that are an extra 100% and 50% larger than the text for each level, respectively. The CSS definition in Fig. 7-27 gives these rules.

```
body {background-color:linen; color:navy; font-family:Arial;}
h1 {font-size:200%;}
h2 {font-size:150%;}
```

Figure 7-27. CSS example.

As can be seen, the style definitions can be compact. Each line selects an element to which it applies and gives the values of properties. The properties of an element apply as defaults to all other HTML elements that it contains. Thus, the style for body sets the style for paragraphs of text in the body. There are also convenient shorthands for color names (e.g., red). Any style parameters that are not defined are filled with defaults by the browser. This behavior makes style sheet definitions optional; some reasonable presentation will occur without them.

Style sheets can be placed in an HTML file (e.g., using the <style> tag), but it is more common to place them in a separate file and reference them. For example, the <head> tag of the AWI page can be modified to refer to a style sheet in the file *awistyle.css* as shown in Fig. 7-28. The example also shows the MIME type of CSS files to be *text/css*.

```
<head>
<title> AMALGAMATED WIDGET, INC. </title>
<link rel="stylesheet" type="text/css" href="awistyle.css" />
</head>
```

Figure 7-28. Including a CSS style sheet.

This strategy has two advantages. First, it lets one set of styles be applied to many pages on a Web site. This organization lends a consistent appearance to pages even if they were developed by different authors at different times, and allows the look of the entire site to be changed by editing one CSS file and not the HTML. This method can be compared to an #include file in a C program: changing one macro definition there changes it in all the program files that include the header. The second advantage is that the HTML files that are downloaded are kept small. This is because the browser can download one copy of the CSS file for all pages that reference it. It does not need to download a new copy of the definitions along with each Web page.

7.3.3 Dynamic Web Pages and Web Applications

The static page model we have used so far treats pages as multimedia documents that are conveniently linked together. It was a fitting model in the early days of the Web, as vast amounts of information were put online. Nowadays, much of the excitement around the Web is using it for applications and services. Examples include buying products on e-commerce sites, searching library catalogs, exploring maps, reading and sending email, and collaborating on documents.

These new uses are like traditional application software (e.g., mail readers and word processors). The twist is that these applications run inside the browser, with user data stored on servers in Internet data centers. They use Web protocols to access information via the Internet, and the browser to display a user interface. The advantage of this approach is that users do not need to install separate application programs, and user data can be accessed from different computers and backed up by the service operator. It is proving so successful that it is rivaling traditional application software. Of course, the fact that these applications are offered for free by large providers helps. This model is the prevalent form of **cloud computing**, in which computing moves off individual desktop computers and into shared clusters of servers in the Internet.

To act as applications, Web pages can no longer be static. Dynamic content is needed. For example, a page of the library catalog should reflect which books are currently available and which books are checked out and are thus not available. Similarly, a useful stock market page would allow the user to interact with the page to see stock prices over different periods of time and compute profits and losses. As these examples suggest, dynamic content can be generated by programs running on the server or in the browser (or in both places).

In this section, we will examine each of these two cases in turn. The general situation is as shown in Fig. 7-29. For example, consider a map service that lets the user enter a street address and presents a corresponding map of the location. Given a request for a location, the Web server must use a program to create a page that shows the map for the location from a database of streets and other geographic information. This action is shown as steps 1 through 3. The request (step

1) causes a program to run on the server. The program consults a database to generate the appropriate page (step 2) and returns it to the browser (step 3).

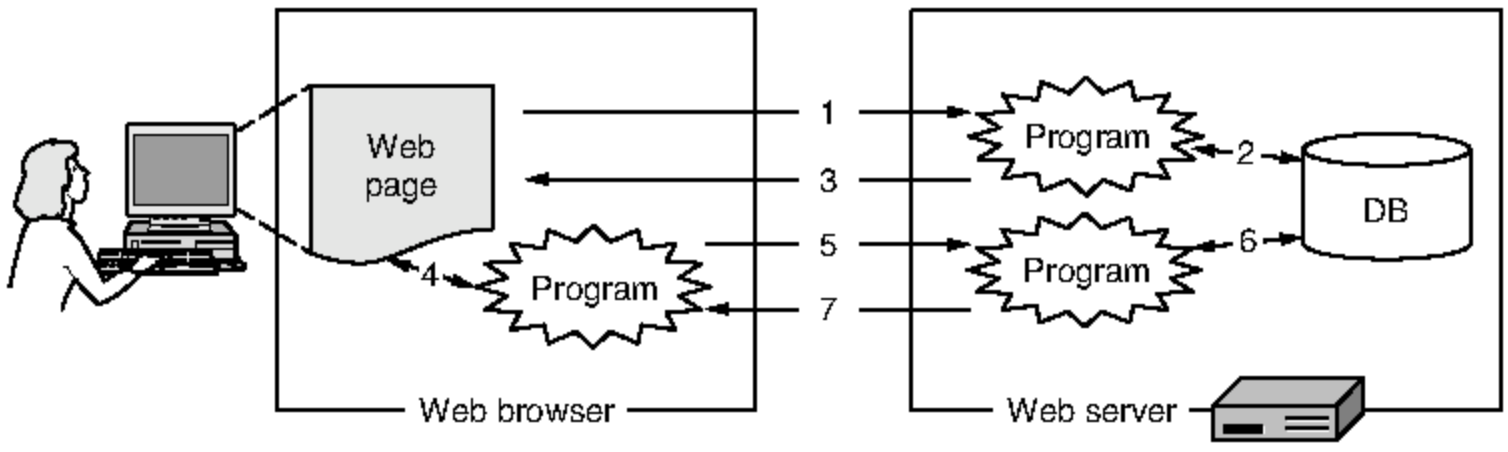


Figure 7-29. Dynamic pages.

There is more to dynamic content, however. The page that is returned may itself contain programs that run in the browser. In our map example, the program would let the user find routes and explore nearby areas at different levels of detail. It would update the page, zooming in or out as directed by the user (step 4). To handle some interactions, the program may need more data from the server. In this case, the program will send a request to the server (step 5) that will retrieve more information from the database (step 6) and return a response (step 7). The program will then continue updating the page (step 4). The requests and responses happen in the background; the user may not even be aware of them because the page URL and title typically do not change. By including client-side programs, the page can present a more responsive interface than with server-side programs alone.

Server-Side Dynamic Web Page Generation

Let us look at the case of server-side content generation in more detail. A simple situation in which server-side processing is necessary is the use of forms. Consider the user filling out the AWI order form of Fig. 7-25(b) and clicking the *Submit order* button. When the user clicks, a request is sent to the server at the URL specified with the form (a *POST* to *http://widget.com/cgi-bin/order.cgi* in this case) along with the contents of the form as filled in by the user. These data must be given to a program or script to process. Thus, the URL identifies the program to run; the data are provided to the program as input. In this case, processing would involve entering the order in AWI's internal system, updating customer records, and charging the credit card. The page returned by this request will depend on what happens during the processing. It is not fixed like a static page. If the order succeeds, the page returned might give the expected shipping date. If it is unsuccessful, the returned page might say that widgets requested are out of stock or the credit card was not valid for some reason.

Exactly how the server runs a program instead of retrieving a file depends on the design of the Web server. It is not specified by the Web protocols themselves. This is because the interface can be proprietary and the browser does not need to know the details. As far as the browser is concerned, it is simply making a request and fetching a page.

Nonetheless, standard APIs have been developed for Web servers to invoke programs. The existence of these interfaces makes it easier for developers to extend different servers with Web applications. We will briefly look at two APIs to give you a sense of what they entail.

The first API is a method for handling dynamic page requests that has been available since the beginning of the Web. It is called the **CGI** (**Common Gateway Interface**) and is defined in RFC 3875. CGI provides an interface to allow Web servers to talk to back-end programs and scripts that can accept input (e.g., from forms) and generate HTML pages in response. These programs may be written in whatever language is convenient for the developer, usually a scripting language for ease of development. Pick Python, Ruby, Perl or your favorite language.

By convention, programs invoked via CGI live in a directory called *cgi-bin*, which is visible in the URL. The server maps a request to this directory to a program name and executes that program as a separate process. It provides any data sent with the request as input to the program. The output of the program gives a Web page that is returned to the browser.

In our example, the program *order.cgi* is invoked with input from the form encoded as shown in Fig. 7-26. It will parse the parameters and process the order. A useful convention is that the program will return the HTML for the order form if no form input is provided. In this way, the program will be sure to know the representation of the form.

The second API we will look at is quite different. The approach here is to embed little scripts inside HTML pages and have them be executed by the server itself to generate the page. A popular language for writing these scripts is **PHP** (**PHP: Hypertext Preprocessor**). To use it, the server has to understand PHP, just as a browser has to understand CSS to interpret Web pages with style sheets. Usually, servers identify Web pages containing PHP from the file extension *php* rather than *html* or *htm*.

PHP is simpler to use than CGI. As an example of how it works with forms, see the example in Fig. 7-30(a). The top part of this figure contains a normal HTML page with a simple form in it. This time, the <form> tag specifies that *action.php* is to be invoked to handle the parameters when the user submits the form. The page displays two text boxes, one with a request for a name and one with a request for an age. After the two boxes have been filled in and the form submitted, the server parses the Fig. 7-26-type string sent back, putting the name in the *name* variable and the age in the *age* variable. It then starts to process the *action.php* file, shown in Fig. 7-30(b), as a reply. During the processing of this file,

the PHP commands are executed. If the user filled in "Barbara" and "24" in the boxes, the HTML file sent back will be the one given in Fig. 7-30(c). Thus, handling forms becomes extremely simple using PHP.

```
<html>
<body>
<form action="action.php" method="post">
<p> Please enter your name: <input type="text" name="name"> </p>
<p> Please enter your age: <input type="text" name="age"> </p>
<input type="submit">
</form>
</body>
</html>
```

(a)

```
<html>
<body>
<h1> Reply: </h1>
Hello <?php echo $name; ?>.
Prediction: next year you will be <?php echo $age + 1; ?>
</body>
</html>
```

(b)

```
<html>
<body>
<h1> Reply: </h1>
Hello Barbara.
Prediction: next year you will be 33
</body>
</html>
```

(c)

Figure 7-30. (a) A Web page containing a form. (b) A PHP script for handling the output of the form. (c) Output from the PHP script when the inputs are "Barbara" and "32", respectively.

Although PHP is easy to use, it is actually a powerful programming language for interfacing the Web and a server database. It has variables, strings, arrays, and most of the control structures found in C, but much more powerful I/O than just *printf*. PHP is open source code, freely available, and widely used. It was designed specifically to work well with Apache, which is also open source and is the world's most widely used Web server. For more information about PHP, see Valade (2009).

We have now seen two different ways to generate dynamic HTML pages: CGI scripts and embedded PHP. There are several others to choose from. **JSP** (**JavaServer Pages**) is similar to PHP, except that the dynamic part is written in

the Java programming language instead of in PHP. Pages using this technique have the file extension *.jsp*. **ASP.NET** (**Active Server Pages .NET**) is Microsoft's version of PHP and JavaServer Pages. It uses programs written in Microsoft's proprietary .NET networked application framework for generating the dynamic content. Pages using this technique have the extension *.aspx*. The choice among these three techniques usually has more to do with politics (open source vs. Microsoft) than with technology, since the three languages are roughly comparable.

Client-Side Dynamic Web Page Generation

PHP and CGI scripts solve the problem of handling input and interactions with databases on the server. They can all accept incoming information from forms, look up information in one or more databases, and generate HTML pages with the results. What none of them can do is respond to mouse movements or interact with users directly. For this purpose, it is necessary to have scripts embedded in HTML pages that are executed on the client machine rather than the server machine. Starting with HTML 4.0, such scripts are permitted using the tag <script>. The technologies used to produce these interactive Web pages are broadly referred to as **dynamic HTML**

The most popular scripting language for the client side is **JavaScript**, so we will now take a quick look at it. Despite the similarity in names, JavaScript has almost nothing to do with the Java programming language. Like other scripting languages, it is a very high-level language. For example, in a single line of JavaScript it is possible to pop up a dialog box, wait for text input, and store the resulting string in a variable. High-level features like this make JavaScript ideal for designing interactive Web pages. On the other hand, the fact that it is mutating faster than a fruit fly trapped in an X-ray machine makes it extremely difficult to write JavaScript programs that work on all platforms, but maybe some day it will stabilize.

As an example of a program in JavaScript, consider that of Fig. 7-31. Like that of Fig. 7-30, it displays a form asking for a name and age, and then predicts how old the person will be next year. The body is almost the same as the PHP example, the main difference being the declaration of the *Submit* button and the assignment statement in it. This assignment statement tells the browser to invoke the *response* script on a button click and pass it the form as a parameter.

What is completely new here is the declaration of the JavaScript function *response* in the head of the HTML file, an area normally reserved for titles, background colors, and so on. This function extracts the value of the *name* field from the form and stores it in the variable *person* as a string. It also extracts the value of the *age* field, converts it to an integer by using the *eval* function, adds 1 to it, and stores the result in *years*. Then it opens a document for output, does four

```
<html>
<head>
<script language="javascript" type="text/javascript">
function response(test_form) {
    var person = test_form.name.value;
    var years = eval(test_form.age.value) + 1;
    document.open();
    document.writeln("<html> <body>");
    document.writeln("Hello " + person + ".<br>");
    document.writeln("Prediction: next year you will be " + years + ".");
    document.writeln("</body> </html>");
    document.close();
}
</script>
</head>

<body>
<form>
Please enter your name: <input type="text" name="name">
<p>
Please enter your age: <input type="text" name="age">
<p>
<input type="button" value="submit" onclick="response(this.form)">
</form>
</body>
</html>
```

Figure 7-31. Use of JavaScript for processing a form.

writes to it using the *writeln* method, and closes the document. The document is an HTML file, as can be seen from the various HTML tags in it. The browser then displays the document on the screen.

It is very important to understand that while PHP and JavaScript look similar in that they both embed code in HTML files, they are processed totally differently. In the PHP example of Fig. 7-30, after the user has clicked on the *submit* button, the browser collects the information into a long string and sends it off to the server as a request for a PHP page. The server loads the PHP file and executes the PHP script that is embedded in to produce a new HTML page. That page is sent back to the browser for display. The browser cannot even be sure that it was produced by a program. This processing is shown as steps 1 to 4 in Fig. 7-32(a).

In the JavaScript example of Fig. 7-31, when the *submit* button is clicked the browser interprets a JavaScript function contained on the page. All the work is done locally, inside the browser. There is no contact with the server. This processing is shown as steps 1 and 2 in Fig. 7-32(b). As a consequence, the result is displayed virtually instantaneously, whereas with PHP there can be a delay of several seconds before the resulting HTML arrives at the client.

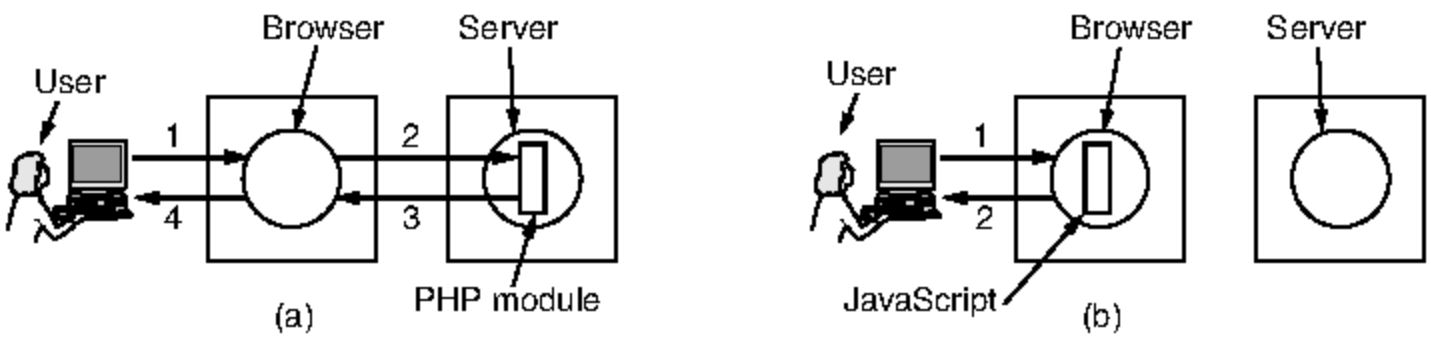


Figure 7-32. (a) Server-side scripting with PHP. (b) Client-side scripting with JavaScript.

This difference does not mean that JavaScript is better than PHP. Their uses are completely different. PHP (and, by implication, JSP and ASP) is used when interaction with a database on the server is needed. JavaScript (and other client-side languages we will mention, such as VBScript) is used when the interaction is with the user at the client computer. It is certainly possible to combine them, as we will see shortly.

JavaScript is not the only way to make Web pages highly interactive. An alternative on Windows platforms is **VBScript**, which is based on Visual Basic. Another popular method across platforms is the use of **applets**. These are small Java programs that have been compiled into machine instructions for a virtual computer called the **JVM** (**Java Virtual Machine**). Applets can be embedded in HTML pages (between <applet> and </applet>) and interpreted by JVM-capable browsers. Because Java applets are interpreted rather than directly executed, the Java interpreter can prevent them from doing Bad Things. At least in theory. In practice, applet writers have found a nearly endless stream of bugs in the Java I/O libraries to exploit.

Microsoft's answer to Sun's Java applets was allowing Web pages to hold **ActiveX controls**, which are programs compiled to x86 machine language and executed on the bare hardware. This feature makes them vastly faster and more flexible than interpreted Java applets because they can do anything a program can do. When Internet Explorer sees an ActiveX control in a Web page, it downloads it, verifies its identity, and executes it. However, downloading and running foreign programs raises enormous security issues, which we will discuss in Chap. 8.

Since nearly all browsers can interpret both Java programs and JavaScript, a designer who wants to make a highly interactive Web page has a choice of at least two techniques, and if portability to multiple platforms is not an issue, ActiveX in addition. As a general rule, JavaScript programs are easier to write, Java applets execute faster, and ActiveX controls run fastest of all. Also, since all browsers implement exactly the same JVM but no two browsers implement the same version of JavaScript, Java applets are more portable than JavaScript programs. For more information about JavaScript, there are many books, each with many (often with more than 1000) pages. See, for example, Flanagan (2010).

AJAX—Asynchronous JavaScript and XML

Compelling Web applications need responsive user interfaces and seamless access to data stored on remote Web servers. Scripting on the client (e.g., with JavaScript) and the server (e.g., with PHP) are basic technologies that provide pieces of the solution. These technologies are commonly used with several other key technologies in a combination called **AJAX** (**Asynchronous JAvascript and Xml**). Many full-featured Web applications, such as Google's Gmail, Maps, and Docs, are written with AJAX.

AJAX is somewhat confusing because it is not a language. It is a set of technologies that work together to enable Web applications that are every bit as responsive and powerful as traditional desktop applications. The technologies are:

1. HTML and CSS to present information as pages.
2. DOM (Document Object Model) to change parts of pages while they are viewed.
3. XML (eXtensible Markup Language) to let programs exchange application data with the server.
4. An asynchronous way for programs to send and retrieve XML data.
5. JavaScript as a language to bind all this functionality together.

As this is quite a collection, we will go through each piece to see what it contributes. We have already seen HTML and CSS. They are standards for describing content and how it should be displayed. Any program that can produce HTML and CSS can use a Web browser as a display engine.

DOM (**Document Object Model**) is a representation of an HTML page that is accessible to programs. This representation is structured as a tree that reflects the structure of the HTML elements. For instance, the DOM tree of the HTML in Fig. 7-30(a) is given in Fig. 7-33. At the root is an *html* element that represents the entire HTML block. This element is the parent of the *body* element, which is in turn parent to a *form* element. The form has two attributes that are drawn to the right-hand side, one for the form method (a *POST*) and one for the form action (the URL to request). This element has three children, reflecting the two paragraph tags and one input tag that are contained within the form. At the bottom of the tree are leaves that contain either elements or literals, such as text strings.

The significance of the DOM model is that it provides programs with a straightforward way to change parts of the page. There is no need to rewrite the entire page. Only the node that contains the change needs to be replaced. When this change is made, the browser will correspondingly update the display. For example, if an image on part of the page is changed in DOM, the browser will update that image without changing the other parts of the page. We have already seen DOM in action when the JavaScript example of Fig. 7-31 added lines to the

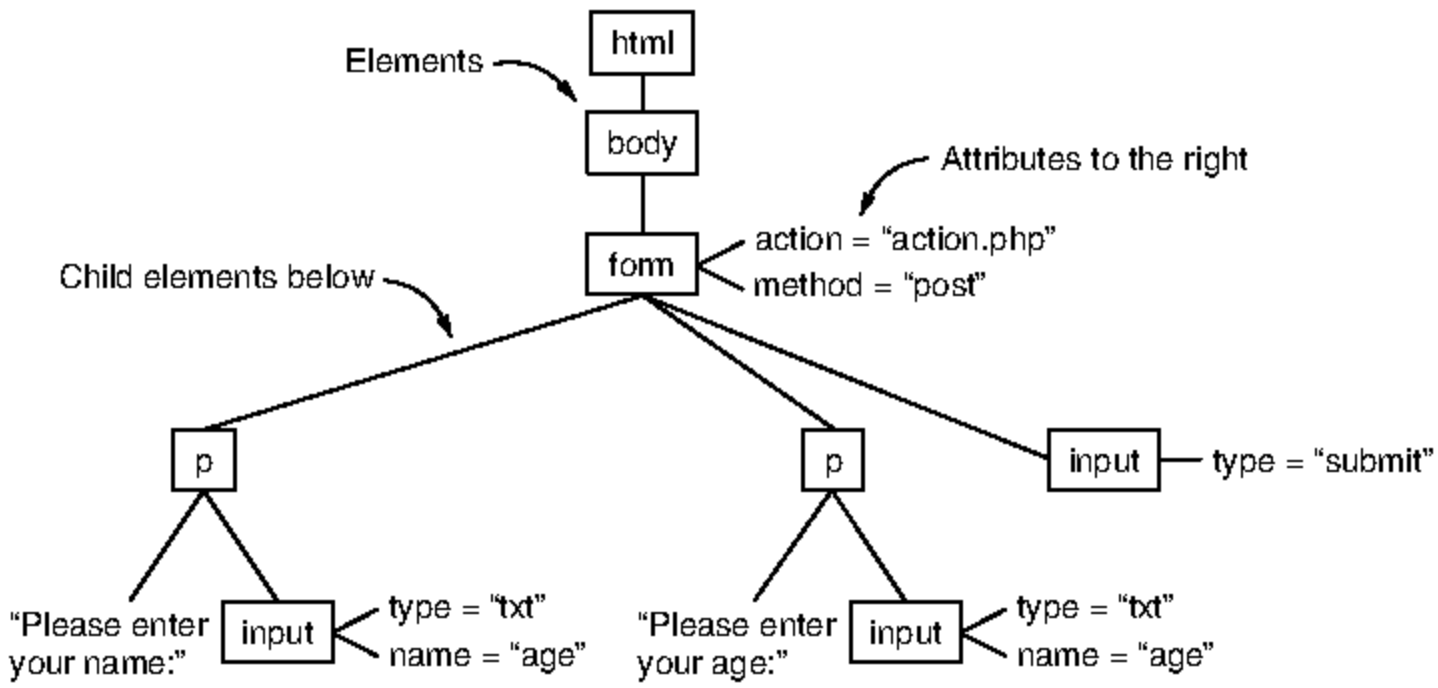


Figure 7-33. The DOM tree for the HTML in Fig. 7-30(a).

document element to cause new lines of text to appear at the bottom of the browser window. The DOM is a powerful method for producing pages that can evolve.

The third technology, **XML** (**eXtensible Markup Language**), is a language for specifying structured content. HTML mixes content with formatting because it is concerned with the presentation of information. However, as Web applications become more common, there is an increasing need to separate structured content from its presentation. For example, consider a program that searches the Web for the best price for some book. It needs to analyze many Web pages looking for the item's title and price. With Web pages in HTML, it is very difficult for a program to figure out where the title is and where the price is.

For this reason, the W3C developed XML (Bray et al., 2006) to allow Web content to be structured for automated processing. Unlike HTML, there are no defined tags for XML. Each user can define her own tags. A simple example of an XML document is given in Fig. 7-34. It defines a structure called book_list, which is a list of books. Each book has three fields, the title, author, and year of publication. These structures are extremely simple. It is permitted to have structures with repeated fields (e.g., multiple authors), optional fields (e.g., URL of the audio book), and alternative fields (e.g., URL of a bookstore if it is in print or URL of an auction site if it is out of print).

In this example, each of the three fields is an indivisible entity, but it is also permitted to further subdivide the fields. For example, the author field could have been done as follows to give finer-grained control over searching and formatting:

```
<author>
  <first_name> George </first_name>
  <last_name> Zipf </last_name>
</author>
```

Each field can be subdivided into subfields and subsubfields, arbitrarily deeply.

```
<?xml version="1.0" ?>

<book_list>

<book>
   <title> Human Behavior and the Principle of Least Effort </title>
   <author> George Zipf </author>
   <year> 1949 </year>
</book>

<book>
   <title> The Mathematical Theory of Communication </title>
   <author> Claude E. Shannon </author>
   <author> Warren Weaver </author>
   <year> 1949 </year>
</book>

<book>
   <title> Nineteen Eighty-Four </title>
   <author> George Orwell </author>
   <year> 1949 </year>
</book>

</book_list>
```

Figure 7-34. A simple XML document.

All the file of Fig. 7-34 does is define a book list containing three books. It is well suited for transporting information between programs running in browsers and servers, but it says nothing about how to display the document as a Web page. To do that, a program that consumes the information and judges 1949 to be a fine year for books might output HTML in which the titles are marked up as italic text. Alternatively, a language called **XSLT** (**eXtensible Stylesheet Language Transformations**), can be used to define how XML should be transformed into HTML. XSLT is like CSS, but much more powerful. We will spare you the details.

The other advantage of expressing data in XML, instead of HTML, is that it is easier for programs to analyze. HTML was originally written manually (and often is still) so a lot of it is a bit sloppy. Sometimes the closing tags, like </p>, are left out. Other tags do not have a matching closing tag, like
. Still other tags may be nested improperly, and the case of tag and attribute names can vary. Most browsers do their best to work out what was probably intended. XML is stricter and cleaner in its definition. Tag names and attributes are always lowercase, tags must always be closed in the reverse of the order that they were opened (or indicate clearly if they are an empty tag with no corresponding close), and attribute values must be enclosed in quotation marks. This precision makes parsing easier and unambiguous.

HTML is even being defined in terms of XML. This approach is called **XHTML** (**eXtended HyperText Markup Language**). Basically, it is a Very

Picky version of HTML. XHTML pages must strictly conform to the XML rules, otherwise they are not accepted by the browser. No more shoddy Web pages and inconsistencies across browsers. As with XML, the intent is to produce pages that are better for programs (in this case Web applications) to process. While XHTML has been around since 1998, it has been slow to catch on. People who produce HTML do not see why they need XHTML, and browser support has lagged. Now HTML 5.0 is being defined so that a page can be represented as either HTML or XHTML to aid the transition. Eventually, XHTML should replace HTML, but it will be a long time before this transition is complete.

XML has also proved popular as a language for communication between programs. When this communication is carried by the HTTP protocol (described in the next section) it is called a Web service. In particular, **SOAP** (**Simple Object Access Protocol**) is a way of implementing Web services that performs RPC between programs in a language- and system-independent way. The client just constructs the request as an XML message and sends it to the server, using the HTTP protocol. The server sends back a reply as an XML-formatted message. In this way, applications on heterogeneous platforms can communicate.

Getting back to AJAX, our point is simply that XML is a useful format to exchange data between programs running in the browser and the server. However, to provide a responsive interface in the browser while sending or receiving data, it must be possible for scripts to perform **asynchronous I/O** that does not block the display while awaiting the response to a request. For example, consider a map that can be scrolled in the browser. When it is notified of the scroll action, the script on the map page may request more map data from the server if the view of the map is near the edge of the data. The interface should not freeze while those data are fetched. Such an interface would win no user awards. Instead, the scrolling should continue smoothly. When the data arrive, the script is notified so that it can use the data. If all goes well, new map data will be fetched before it is needed. Modern browsers have support for this model of communication.

The final piece of the puzzle is a scripting language that holds AJAX together by providing access to the above list of technologies. In most cases, this language is JavaScript, but there are alternatives such as VBScript. We presented a simple example of JavaScript earlier. Do not be fooled by this simplicity. JavaScript has many quirks, but it is a full-blown programming language, with all the power of C or Java. It has variables, strings, arrays, objects, functions, and all the usual control structures. It also has interfaces specific to the browser and Web pages. JavaScript can track mouse motion over objects on the screen, which makes it easy to make a menu suddenly appear and leads to lively Web pages. It can use DOM to access pages, manipulate HTML and XML, and perform asynchronous HTTP communication.

Before leaving the subject of dynamic pages, let us briefly summarize the technologies we have covered so far by relating them on a single figure. Complete Web pages can be generated on the fly by various scripts on the server

machine. The scripts can be written in server extension languages like PHP, JSP, or ASP.NET, or run as separate CGI processes and thus be written in any language. These options are shown in Fig. 7-35.

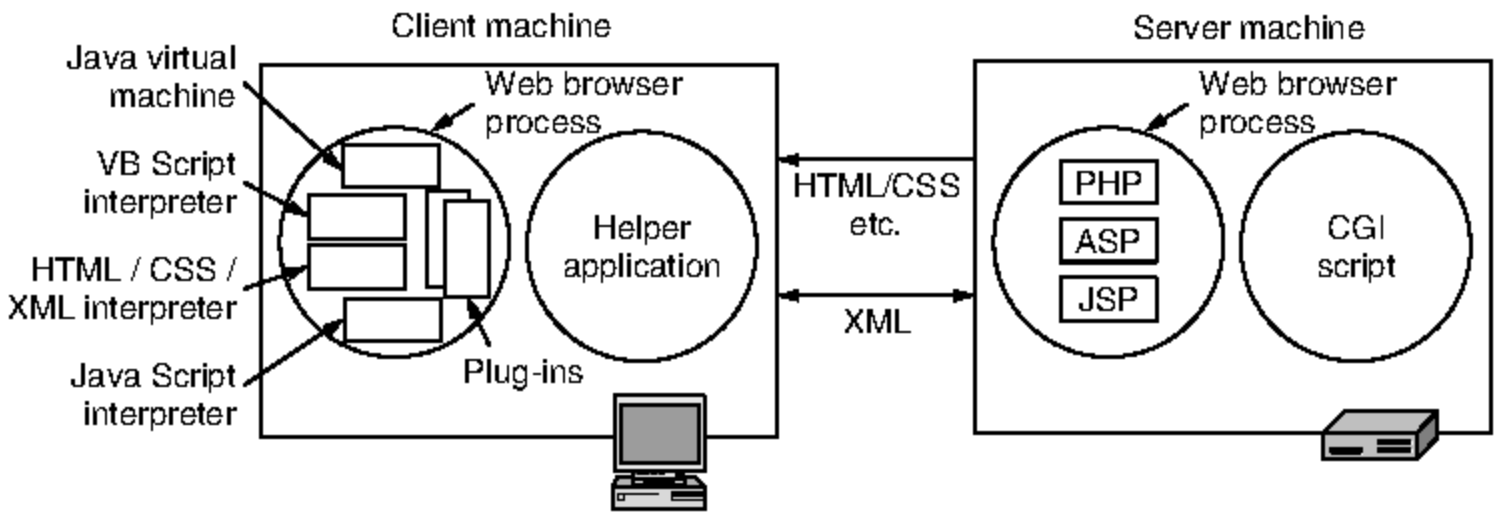


Figure 7-35. Various technologies used to generate dynamic pages.

Once these Web pages are received by the browser, they are treated as normal pages in HTML, CSS and other MIME types and just displayed. Plug-ins that run in the browser and helper applications that run outside of the browser can be installed to extend the MIME types that are supported by the browser.

Dynamic content generation is also possible on the client side. The programs that are embedded in Web pages can be written in JavaScript, VBScript, Java, and other languages. These programs can perform arbitrary computations and update the display. With AJAX, programs in Web pages can asynchronously exchange XML and other kinds of data with the server. This model supports rich Web applications that look just like traditional applications, except that they run inside the browser and access information that is stored at servers on the Internet.

7.3.4 HTTP—The HyperText Transfer Protocol

Now that we have an understanding of Web content and applications, it is time to look at the protocol that is used to transport all this information between Web servers and clients. It is **HTTP** (**HyperText Transfer Protocol**), as specified in RFC 2616.

HTTP is a simple request-response protocol that normally runs over TCP. It specifies what messages clients may send to servers and what responses they get back in return. The request and response headers are given in ASCII, just like in SMTP. The contents are given in a MIME-like format, also like in SMTP. This simple model was partly responsible for the early success of the Web because it made development and deployment straightforward.

In this section, we will look at the more important properties of HTTP as it is used nowadays. However, before getting into the details we will note that the way

it is used in the Internet is evolving. HTTP is an application layer protocol because it runs on top of TCP and is closely associated with the Web. That is why we are covering it in this chapter. However, in another sense HTTP is becoming more like a transport protocol that provides a way for processes to communicate content across the boundaries of different networks. These processes do not have to be a Web browser and Web server. A media player could use HTTP to talk to a server and request album information. Antivirus software could use HTTP to download the latest updates. Developers could use HTTP to fetch project files. Consumer electronics products like digital photo frames often use an embedded HTTP server as an interface to the outside world. Machine-to-machine communication increasingly runs over HTTP. For example, an airline server might use SOAP (an XML RPC over HTTP) to contact a car rental server and make a car reservation, all as part of a vacation package. These trends are likely to continue, along with the expanding use of HTTP.

Connections

The usual way for a browser to contact a server is to establish a TCP connection to port 80 on the server's machine, although this procedure is not formally required. The value of using TCP is that neither browsers nor servers have to worry about how to handle long messages, reliability, or congestion control. All of these matters are handled by the TCP implementation.

Early in the Web, with HTTP 1.0, after the connection was established a single request was sent over and a single response was sent back. Then the TCP connection was released. In a world in which the typical Web page consisted entirely of HTML text, this method was adequate. Quickly, the average Web page grew to contain large numbers of embedded links for content such as icons and other eye candy. Establishing a separate TCP connection to transport each single icon became a very expensive way to operate.

This observation led to HTTP 1.1, which supports **persistent connections**. With them, it is possible to establish a TCP connection, send a request and get a response, and then send additional requests and get additional responses. This strategy is also called **connection reuse**. By amortizing the TCP setup, startup, and release costs over multiple requests, the relative overhead due to TCP is reduced per request. It is also possible to pipeline requests, that is, send request 2 before the response to request 1 has arrived.

The performance difference between these three cases is shown in Fig. 7-36. Part (a) shows three requests, one after the other and each in a separate connection. Let us suppose that this represents a Web page with two embedded images on the same server. The URLs of the images are determined as the main page is fetched, so they are fetched after the main page. Nowadays, a typical page has around 40 other objects that must be fetched to present it, but that would make our figure far too big so we will use only two embedded objects.

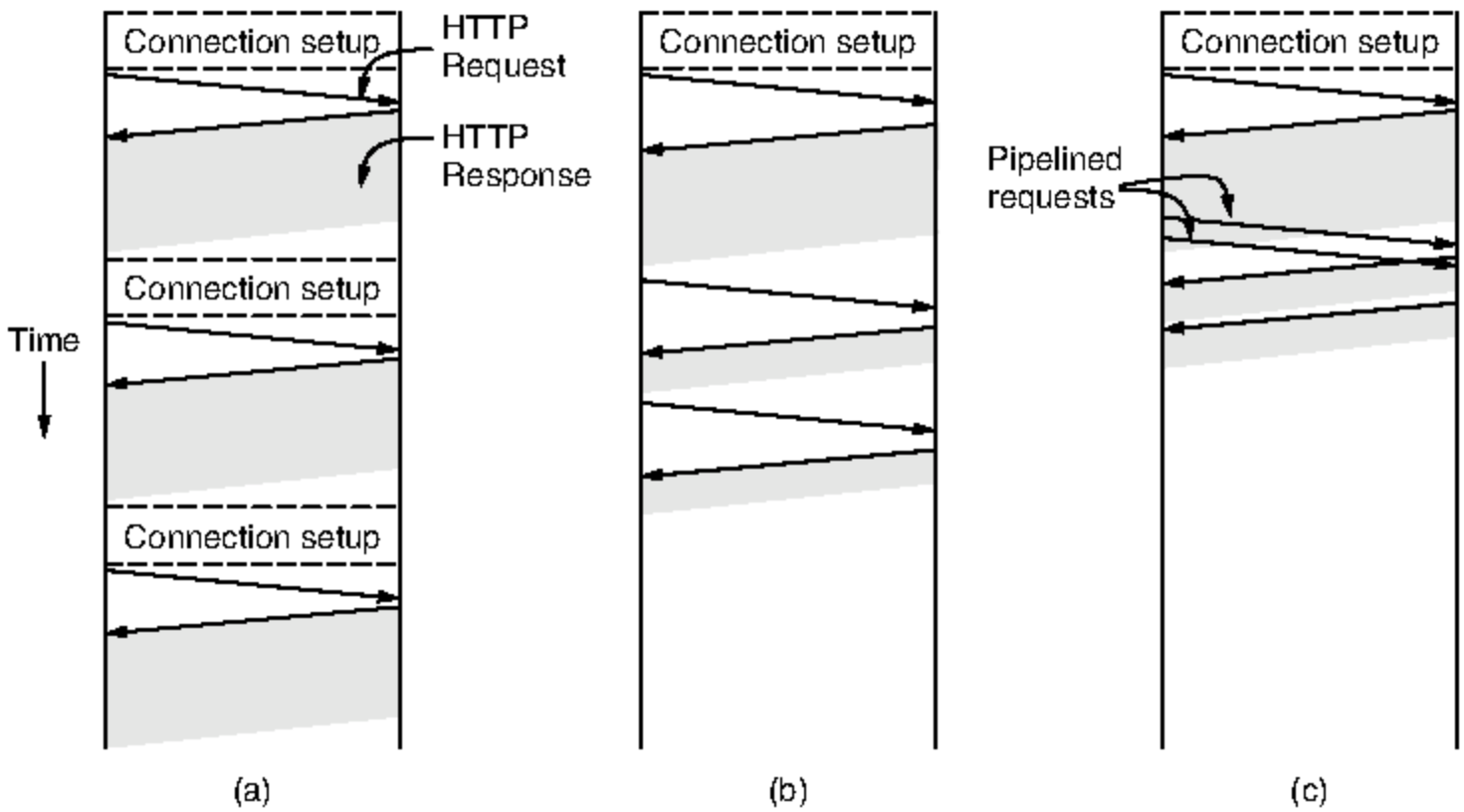


Figure 7-36. HTTP with (a) multiple connections and sequential requests. (b) A persistent connection and sequential requests. (c) A persistent connection and pipelined requests.

In Fig. 7-36(b), the page is fetched with a persistent connection. That is, the TCP connection is opened at the beginning, then the same three requests are sent, one after the other as before, and only then is the connection closed. Observe that the fetch completes more quickly. There are two reasons for the speedup. First, time is not wasted setting up additional connections. Each TCP connection requires at least one round-trip time to establish. Second, the transfer of the same images proceeds more quickly. Why is this? It is because of TCP congestion control. At the start of a connection, TCP uses the slow-start procedure to increase the throughput until it learns the behavior of the network path. The consequence of this warmup period is that multiple short TCP connections take disproportionately longer to transfer information than one longer TCP connection.

Finally, in Fig. 7-36(c), there is one persistent connection and the requests are pipelined. Specifically, the second and third requests are sent in rapid succession as soon as enough of the main page has been retrieved to identify that the images must be fetched. The responses for these requests follow eventually. This method cuts down the time that the server is idle, so it further improves performance.

Persistent connections do not come for free, however. A new issue that they raise is when to close the connection. A connection to a server should stay open while the page loads. What then? There is a good chance that the user will click on a link that requests another page from the server. If the connection remains open, the next request can be sent immediately. However, there is no guarantee that the client will make another request of the server any time soon. In practice,

clients and servers usually keep persistent connections open until they have been idle for a short time (e.g., 60 seconds) or they have a large number of open connections and need to close some.

The observant reader may have noticed that there is one combination that we have left out so far. It is also possible to send one request per TCP connection, but run multiple TCP connections in parallel. This **parallel connection** method was widely used by browsers before persistent connections. It has the same disadvantage as sequential connections—extra overhead—but much better performance. This is because setting up and ramping up the connections in parallel hides some of the latency. In our example, connections for both of the embedded images could be set up at the same time. However, running many TCP connections to the same server is discouraged. The reason is that TCP performs congestion control for each connection independently. As a consequence, the connections compete against each other, causing added packet loss, and in aggregate are more aggressive users of the network than an individual connection. Persistent connections are superior and used in preference to parallel connections because they avoid overhead and do not suffer from congestion problems.

Methods

Although HTTP was designed for use in the Web, it was intentionally made more general than necessary with an eye to future object-oriented uses. For this reason, operations, called **methods**, other than just requesting a Web page are supported. This generality is what permitted SOAP to come into existence.

Each request consists of one or more lines of ASCII text, with the first word on the first line being the name of the method requested. The built-in methods are listed in Fig. 7-37. The names are case sensitive, so *GET* is allowed but not *get*.

Method	Description
GET	Read a Web page
HEAD	Read a Web page's header
POST	Append to a Web page
PUT	Store a Web page
DELETE	Remove the Web page
TRACE	Echo the incoming request
CONNECT	Connect through a proxy
OPTIONS	Query options for a page

Figure 7-37. The built-in HTTP request methods.

The *GET* method requests the server to send the page. (When we say "page" we mean "object" in the most general case, but thinking of a page as the contents

of a file is sufficient to understand the concepts.) The page is suitably encoded in MIME. The vast majority of requests to Web servers are *GET*s. The usual form of *GET* is

```
GET filename HTTP/1.1
```

where *filename* names the page to be fetched and 1.1 is the protocol version.

The *HEAD* method just asks for the message header, without the actual page. This method can be used to collect information for indexing purposes, or just to test a URL for validity.

The *POST* method is used when forms are submitted. Both it and *GET* are also used for SOAP Web services. Like *GET*, it bears a URL, but instead of simply retrieving a page it uploads data to the server (i.e., the contents of the form or RPC parameters). The server then does something with the data that depends on the URL, conceptually appending the data to the object. The effect might be to purchase an item, for example, or to call a procedure. Finally, the method returns a page indicating the result.

The remaining methods are not used much for browsing the Web. The *PUT* method is the reverse of *GET*: instead of reading the page, it writes the page. This method makes it possible to build a collection of Web pages on a remote server. The body of the request contains the page. It may be encoded using MIME, in which case the lines following the *PUT* might include authentication headers, to prove that the caller indeed has permission to perform the requested operation.

DELETE does what you might expect: it removes the page, or at least it indicates that the Web server has agreed to remove the page. As with *PUT*, authentication and permission play a major role here.

The *TRACE* method is for debugging. It instructs the server to send back the request. This method is useful when requests are not being processed correctly and the client wants to know what request the server actually got.

The *CONNECT* method lets a user make a connection to a Web server through an intermediate device, such as a Web cache.

The *OPTIONS* method provides a way for the client to query the server for a page and obtain the methods and headers that can be used with that page.

Every request gets a response consisting of a status line, and possibly additional information (e.g., all or part of a Web page). The status line contains a three-digit status code telling whether the request was satisfied and, if not, why not. The first digit is used to divide the responses into five major groups, as shown in Fig. 7-38. The 1xx codes are rarely used in practice. The 2xx codes mean that the request was handled successfully and the content (if any) is being returned. The 3xx codes tell the client to look elsewhere, either using a different URL or in its own cache (discussed later). The 4xx codes mean the request failed due to a client error such an invalid request or a nonexistent page. Finally, the 5xx errors mean the server itself has an internal problem, either due to an error in its code or to a temporary overload.

Code	Meaning	Examples
1xx	Information	100 = server agrees to handle client's request
2xx	Success	200 = request succeeded; 204 = no content present
3xx	Redirection	301 = page moved; 304 = cached page still valid
4xx	Client error	403 = forbidden page; 404 = page not found
5xx	Server error	500 = internal server error; 503 = try again later

Figure 7-38. The status code response groups.

Message Headers

The request line (e.g., the line with the *GET* method) may be followed by additional lines with more information. They are called **request headers**. This information can be compared to the parameters of a procedure call. Responses may also have **response headers**. Some headers can be used in either direction. A selection of the more important ones is given in Fig. 7-39. This list is not short, so as you might imagine there is often a variety of headers on each request and response.

The *User-Agent* header allows the client to inform the server about its browser implementation (e.g., *Mozilla/5.0* and *Chrome/5.0.375.125*). This information is useful to let servers tailor their responses to the browser, since different browsers can have widely varying capabilities and behaviors.

The four *Accept* headers tell the server what the client is willing to accept in the event that it has a limited repertoire of what is acceptable. The first header specifies the MIME types that are welcome (e.g., *text/html*). The second gives the character set (e.g., *ISO-8859-5* or *Unicode-1-1*). The third deals with compression methods (e.g., *gzip*). The fourth indicates a natural language (e.g., Spanish). If the server has a choice of pages, it can use this information to supply the one the client is looking for. If it is unable to satisfy the request, an error code is returned and the request fails.

The *If-Modified-Since* and *If-None-Match* headers are used with caching. They let the client ask for a page to be sent only if the cached copy is no longer valid. We will describe caching shortly.

The *Host* header names the server. It is taken from the URL. This header is mandatory. It is used because some IP addresses may serve multiple DNS names and the server needs some way to tell which host to hand the request to.

The *Authorization* header is needed for pages that are protected. In this case, the client may have to prove it has a right to see the page requested. This header is used for that case.

The client uses the misspelled *Referer* header to give the URL that referred to the URL that is now requested. Most often this is the URL of the previous page.

Header	Type	Contents
User-Agent	Request	Information about the browser and its platform
Accept	Request	The type of pages the client can handle
Accept-Charset	Request	The character sets that are acceptable to the client
Accept-Encoding	Request	The page encodings the client can handle
Accept-Language	Request	The natural languages the client can handle
If-Modified-Since	Request	Time and date to check freshness
If-None-Match	Request	Previously sent tags to check freshness
Host	Request	The server's DNS name
Authorization	Request	A list of the client's credentials
Referer	Request	The previous URL from which the request came
Cookie	Request	Previously set cookie sent back to the server
Set-Cookie	Response	Cookie for the client to store
Server	Response	Information about the server
Content-Encoding	Response	How the content is encoded (e.g., *gzip*)
Content-Language	Response	The natural language used in the page
Content-Length	Response	The page's length in bytes
Content-Type	Response	The page's MIME type
Content-Range	Response	Identifies a portion of the page's content
Last-Modified	Response	Time and date the page was last changed
Expires	Response	Time and date when the page stops being valid
Location	Response	Tells the client where to send its request
Accept-Ranges	Response	Indicates the server will accept byte range requests
Date	Both	Date and time the message was sent
Range	Both	Identifies a portion of a page
Cache-Control	Both	Directives for how to treat caches
ETag	Both	Tag for the contents of the page
Upgrade	Both	The protocol the sender wants to switch to

Figure 7-39. Some HTTP message headers.

This header is particularly useful for tracking Web browsing, as it tells servers how a client arrived at the page.

Although cookies are dealt with in RFC 2109 rather than RFC 2616, they also have headers. The *Set-Cookie* header is how servers send cookies to clients. The client is expected to save the cookie and return it on subsequent requests to the server by using the *Cookie* header. (Note that there is a more recent specification for cookies with newer headers, RFC 2965, but this has largely been rejected by industry and is not widely implemented.)

Many other headers are used in responses. The *Server* header allows the server to identify its software build if it wishes. The next five headers, all starting with *Content-*, allow the server to describe properties of the page it is sending.

The *Last-Modified* header tells when the page was last modified, and the *Expires* header tells for how long the page will remain valid. Both of these headers play an important role in page caching.

The *Location* header is used by the server to inform the client that it should try a different URL. This can be used if the page has moved or to allow multiple URLs to refer to the same page (possibly on different servers). It is also used for companies that have a main Web page in the *com* domain but redirect clients to a national or regional page based on their IP addresses or preferred language.

If a page is very large, a small client may not want it all at once. Some servers will accept requests for byte ranges, so the page can be fetched in multiple small units. The *Accept-Ranges* header announces the server's willingness to handle this type of partial page request.

Now we come to headers that can be used in both directions. The *Date* header can be used in both directions and contains the time and date the message was sent, while the *Range* header tells the byte range of the page that is provided by the response.

The *ETag* header gives a short tag that serves as a name for the content of the page. It is used for caching. The *Cache-Control* header gives other explicit instructions about how to cache (or, more usually, how not to cache) pages.

Finally, the *Upgrade* header is used for switching to a new communication protocol, such as a future HTTP protocol or a secure transport. It allows the client to announce what it can support and the server to assert what it is using.

Caching

People often return to Web pages that they have viewed before, and related Web pages often have the same embedded resources. Some examples are the images that are used for navigation across the site, as well as common style sheets and scripts. It would be very wasteful to fetch all of these resources for these pages each time they are displayed because the browser already has a copy.

Squirreling away pages that are fetched for subsequent use is called **caching**. The advantage is that when a cached page can be reused, it is not necessary to repeat the transfer. HTTP has built-in support to help clients identify when they can safely reuse pages. This support improves performance by reducing both network traffic and latency. The trade-off is that the browser must now store pages, but this is nearly always a worthwhile trade-off because local storage is inexpensive. The pages are usually kept on disk so that they can be used when the browser is run at a later date.

The difficult issue with HTTP caching is how to determine that a previously cached copy of a page is the same as the page would be if it was fetched again.

This determination cannot be made solely from the URL. For example, the URL may give a page that displays the latest news item. The contents of this page will be updated frequently even though the URL stays the same. Alternatively, the contents of the page may be a list of the gods from Greek and Roman mythology. This page should change somewhat less rapidly.

HTTP uses two strategies to tackle this problem. They are shown in Fig. 7-40 as forms of processing between the request (step 1) and the response (step 5). The first strategy is page validation (step 2). The cache is consulted, and if it has a copy of a page for the requested URL that is known to be fresh (i.e., still valid), there is no need to fetch it anew from the server. Instead, the cached page can be returned directly. The *Expires* header returned when the cached page was originally fetched and the current date and time can be used to make this determination.

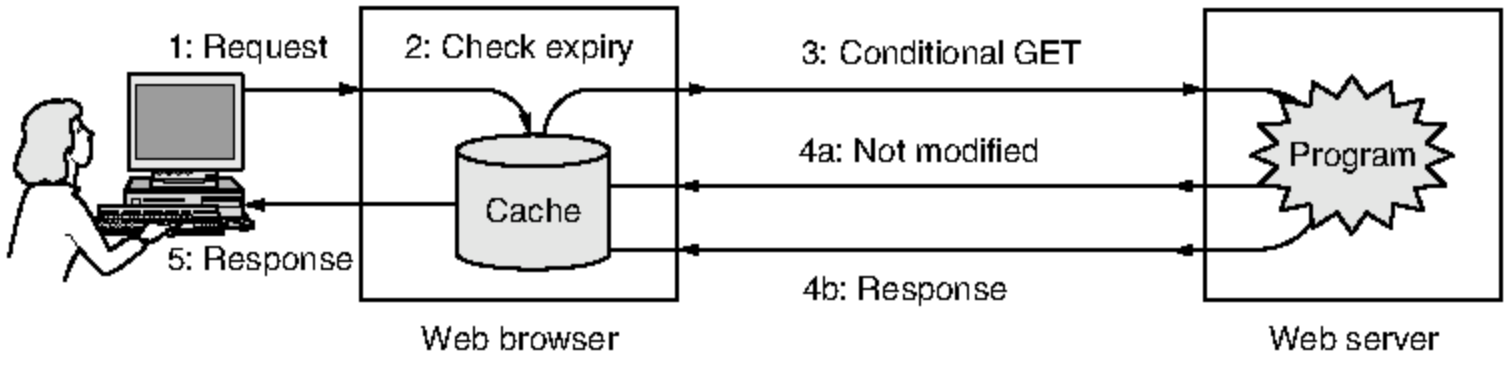


Figure 7-40. HTTP caching.

However, not all pages come with a convenient *Expires* header that tells when the page must be fetched again. After all, making predictions is hard—especially about the future. In this case, the browser may use heuristics. For example, if the page has not been modified in the past year (as told by the *Last-Modified* header) it is a fairly safe bet that it will not change in the next hour. There is no guarantee, however, and this may be a bad bet. For example, the stock market might have closed for the day so that the page will not change for hours, but it will change rapidly once the next trading session starts. Thus, the cacheability of a page may vary wildly over time. For this reason, heuristics should be used with care, though they often work well in practice.

Finding pages that have not expired is the most beneficial use of caching because it means that the server does not need to be contacted at all. Unfortunately, it does not always work. Servers must use the *Expires* header conservatively, since they may be unsure when a page will be updated. Thus, the cached copies may still be fresh, but the client does not know.

The second strategy is used in this case. It is to ask the server if the cached copy is still valid. This request is a **conditional GET**, and it is shown in Fig. 7-40 as step 3. If the server knows that the cached copy is still valid, it can send a short reply to say so (step 4a). Otherwise, it must send the full response (step 4b).

More header fields are used to let the server check whether a cached copy is still valid. The client has the time a cached page was last updated from the *Last-Modified* header. It can send this time to the server using the *If-Modified-Since* header to ask for the page only if it has been changed in the meantime.

Alternatively, the server may return an *ETag* header with a page. This header gives a tag that is a short name for the content of the page, like a checksum but better. (It can be a cryptographic hash, which we will describe in Chap. 8.) The client can validate cached copies by sending the server an *If-None-Match* header listing the tags of the cached copies. If any of the tags match the content that the server would respond with, the corresponding cached copy may be used. This method can be used when it is not convenient or useful to determine freshness. For example, a server may return different content for the same URL depending on what languages and MIME types are preferred. In this case, the modification date alone will not help the server to determine if the cached page is fresh.

Finally, note that both of these caching strategies are overridden by the directives carried in the *Cache-Control* header. These directives can be used to restrict caching (e.g., *no-cache*) when it is not appropriate. An example is a dynamic page that will be different the next time it is fetched. Pages that require authorization are also not cached.

There is much more to caching, but we only have the space to make two important points. First, caching can be performed at other places besides in the browser. In the general case, HTTP requests can be routed through a series of caches. The use of a cache external to the browser is called **proxy caching**. Each added level of caching can help to reduce requests further up the chain. It is common for organizations such as ISPs and companies to run proxy caches to gain the benefits of caching pages across different users. We will discuss proxy caching with the broader topic of content distribution in Sec. 7.5 at the end of this chapter.

Second, caches provide an important boost to performance, but not as much as one might hope. The reason is that, while there are certainly popular documents on the Web, there are also a great many unpopular documents that people fetch, many of which are also very long (e.g., videos). The "long tail" of unpopular documents take up space in caches, and the number of requests that can be handled from the cache grows only slowly with the size of the cache. Web caches are always likely to be able to handle less than half of the requests. See Breslau et al. (1999) for more information.

Experimenting with HTTP

Because HTTP is an ASCII protocol, it is quite easy for a person at a terminal (as opposed to a browser) to directly talk to Web servers. All that is needed is a TCP connection to port 80 on the server. Readers are encouraged to experiment with the following command sequence. It will work in most UNIX shells and the command window on Windows (once the telnet program is enabled).

```
telnet www.ietf.org 80
GET /rfc.html HTTP/1.1
Host: www.ietf.org
```

This sequence of commands starts up a telnet (i.e., TCP) connection to port 80 on IETF's Web server, *www.ietf.org*. Then comes the *GET* command naming the path of the URL and the protocol. Try servers and URLs of your choosing. The next line is the mandatory *Host* header. A blank line following the last header is mandatory. It tells the server that there are no more request headers. The server will then send the response. Depending on the server and the URL, many different kinds of headers and pages can be observed.

7.3.5 The Mobile Web

The Web is used from most every type of computer, and that includes mobile phones. Browsing the Web over a wireless network while mobile can be very useful. It also presents technical problems because much Web content was designed for flashy presentations on desktop computers with broadband connectivity. In this section we will describe how Web access from mobile devices, or the **mobile Web**, is being developed.

Compared to desktop computers at work or at home, mobile phones present several difficulties for Web browsing:

1. Relatively small screens preclude large pages and large images.
2. Limited input capabilities make it tedious to enter URLs or other lengthy input.
3. Network bandwidth is limited over wireless links, particularly on cellular (3G) networks, where it is often expensive too.
4. Connectivity may be intermittent.
5. Computing power is limited, for reasons of battery life, size, heat dissipation, and cost.

These difficulties mean that simply using desktop content for the mobile Web is likely to deliver a frustrating user experience.

Early approaches to the mobile Web devised a new protocol stack tailored to wireless devices with limited capabilities. **WAP (Wireless Application Protocol)** is the most well-known example of this strategy. The WAP effort was started in 1997 by major mobile phone vendors that included Nokia, Ericsson, and Motorola. However, something unexpected happened along the way. Over the next decade, network bandwidth and device capabilities grew tremendously with the deployment of 3G data services and mobile phones with larger color displays,

faster processors, and 802.11 wireless capabilities. All of a sudden, it was possible for mobiles to run simple Web browsers. There is still a gap between these mobiles and desktops that will never close, but many of the technology problems that gave impetus to a separate protocol stack have faded.

The approach that is increasingly used is to run the same Web protocols for mobiles and desktops, and to have Web sites deliver mobile-friendly content when the user happens to be on a mobile device. Web servers are able to detect whether to return desktop or mobile versions of Web pages by looking at the request headers. The *User-Agent* header is especially useful in this regard because it identifies the browser software. Thus, when a Web server receives a request, it may look at the headers and return a page with small images, less text, and simpler navigation to an iPhone and a full-featured page to a user on a laptop.

W3C is encouraging this approach in several ways. One way is to standardize best practices for mobile Web content. A list of 60 such best practices is provided in the first specification (Rabin and McCathieNevile, 2008). Most of these practices take sensible steps to reduce the size of pages, including by the use of compression, since the costs of communication are higher than those of computation, and by maximizing the effectiveness of caching. This approach encourages sites, especially large sites, to create mobile Web versions of their content because that is all that is required to capture mobile Web users. To help those users along, there is also a logo to indicate pages that can be viewed (well) on the mobile Web.

Another useful tool is a stripped-down version of HTML called **XHTML Basic**. This language is a subset of XHTML that is intended for use by mobile phones, televisions, PDAs, vending machines, pagers, cars, game machines, and even watches. For this reason, it does not support style sheets, scripts, or frames, but most of the standard tags are there. They are grouped into 11 modules. Some are required; some are optional. All are defined in XML. The modules and some example tags are listed in Fig. 7-41.

However, not all pages will be designed to work well on the mobile Web. Thus, a complementary approach is the use of **content transformation** or **transcoding**. In this approach, a computer that sits between the mobile and the server takes requests from the mobile, fetches content from the server, and transforms it to mobile Web content. A simple transformation is to reduce the size of large images by reformatting them at a lower resolution. Many other small but useful transformations are possible. Transcoding has been used with some success since the early days of the mobile Web. See, for example, Fox et al. (1996). However, when both approaches are used there is a tension between the mobile content decisions that are made by the server and by the transcoder. For instance, a Web site may select a particular combination of image and text for a mobile Web user, only to have a transcoder change the format of the image.

Our discussion so far has been about content, not protocols, as it is the content that is the biggest problem in realizing the mobile Web. However, we will briefly mention the issue of protocols. The HTTP, TCP, and IP protocols used by the

Module	Req.?	Function	Example tags
Structure	Yes	Doc. structure	body, head, html, title
Text	Yes	Information	br, code, dfn, em, h*n*, kbd, p, strong
Hypertext	Yes	Hyperlinks	a
List	Yes	Itemized lists	dl, dt, dd, ol, ul, li
Forms	No	Fill-in forms	form, input, label, option, textarea
Tables	No	Rectangular tables	caption, table, td, th, tr
Image	No	Pictures	img
Object	No	Applets, maps, etc.	object, param
Meta-information	No	Extra info	meta
Link	No	Similar to <a>	link
Base	No	URL starting point	base

Figure 7-41. The XHTML Basic modules and tags.

Web may consume a significant amount of bandwidth on protocol overheads such as headers. To tackle this problem, WAP and other solutions defined special-purpose protocols. This turns out to be largely unecessary. Header compression technologies, such as ROHC (RObust Header Compression) described in Chap. 6, can reduce the overheads of these protocols. In this way, it is possible to have one set of protocols (HTTP, TCP, IP) and use them over either high- or low- bandwidth links. Use over the low-bandwidth links simply requires that header compression be turned on.

7.3.6 Web Search

To finish our description of the Web, we will discuss what is arguably the most successful Web application: search. In 1998, Sergey Brin and Larry Page, then graduate students at Stanford, formed a startup called Google to build a better Web search engine. They were armed with the then-radical idea that a search algorithm that counted how many times each page was pointed to by other pages was a better measure of its importance than how many times it contained the key words being sought. For instance, many pages link to the main Cisco page, which makes this page more important to a user searching for "Cisco" than a page outside of the company that happens to use the word "Cisco" many times.

They were right. It did prove possible to build a better search engine, and people flocked to it. Backed by venture capital, Google grew tremendously. It became a public company in 2004, with a market capitalization of $23 billion. By 2010, it was estimated to run more than one million servers in data centers throughout the world.

In one sense, search is simply another Web application, albeit one of the most mature Web applications because it has been under development since the early days of the Web. However, Web search has proved indispensible in everyday usage. Over one billion Web searches are estimated to be done each day. People looking for all manner of information use search as a starting point. For example, to find out where to buy Vegemite in Seattle, there is no obvious Web site to use as a starting point. But chances are that a search engine knows of a page with the desired information and can quickly direct you to the answer.

To perform a Web search in the traditional manner, the user directs her browser to the URL of a Web search site. The major search sites include Google, Yahoo!, and Bing. Next, the user submits search terms using a form. This act causes the search engine to perform a query on its database for relevant pages or images, or whatever kind of resource is being searched for, and return the result as a dynamic page. The user can then follow links to the pages that have been found.

Web search is an interesting topic for discussion because it has implications for the design and use of networks. First, there is the question of how Web search finds pages. The Web search engine must have a database of pages to run a query. Each HTML page may contain links to other pages, and everything interesting (or at least searchable) is linked somewhere. This means that it is theoretically possible to start with a handful of pages and find all other pages on the Web by doing a traversal of all pages and links. This process is called **Web crawling**. All Web search engines use Web crawlers.

One issue with crawling is the kind of pages that it can find. Fetching static documents and following links is easy. However, many Web pages contain programs that display different pages depending on user interaction. An example is an online catalog for a store. The catalog may contain dynamic pages created from a product database and queries for different products. This kind of content is different from static pages that are easy to traverse. How do Web crawlers find these dynamic pages? The answer is that, for the most part, they do not. This kind of hidden content is called the **deep Web**. How to search the deep Web is an open problem that researchers are now tackling. See, for example, madhavan et al. (2008). There are also conventions by which sites make a page (known as *robots.txt*) to tell crawlers what parts of the sites should or should not be visited.

A second consideration is how to process all of the crawled data. To let indexing algorithms be run over the mass of data, the pages must be stored. Estimates vary, but the main search engines are thought to have an index of tens of billions of pages taken from the visible part of the Web. The average page size is estimated at 320 KB. These figures mean that a crawled copy of the Web takes on the order of 20 petabytes or 2×10^{16} bytes to store. While this is a truly huge number, it is also an amount of data that can comfortably be stored and processed in Internet data centers (Chang et al., 2006). For example, if disk storage costs \$20/TB, then 2×10^4 TB costs \$400,000, which is not exactly a huge amount for companies the size of Google, Microsoft, and Yahoo!. And while the Web is