normal delivery of data. Furthermore, datagrams sent to these addresses only reach machines on the same local network as the sender; there are no IP multicast addresses that refer to all systems in the internet or all routers in the internet.

17.8 Multicast Address Semantics

IP treats multicast addresses differently than unicast addresses. For example, a multicast address can only be used as a destination address. Thus, a multicast address can never appear in the source address field of a datagram, nor can it appear in a source route or record route option. Furthermore, no ICMP error messages can be generated about multicast datagrams (e.g., destination unreachable, source quench, echo reply, or time exceeded). Thus, a ping sent to a multicast address will go unanswered.

The rule prohibiting ICMP errors is somewhat surprising because IP routers do honor the time-to-live field in the header of a multicast datagram. As usual, each router decrements the count, and discards the datagram (without sending an ICMP message) if the count reaches zero. We will see that some protocols use the time-to-live count as a way to limit datagram propagation.

17.9 Mapping IP Multicast To Ethernet Multicast

Although the IP multicast standard does not cover all types of network hardware, it does specify how to map an IP multicast address to an Ethernet multicast address. The mapping is efficient and easy to understand:

> *To map an IP multicast address to the corresponding Ethernet multicast address, place the low-order 23 bits of the IP multicast address into the low-order 23 bits of the special Ethernet multicast address* $01.00.5E.00.00.00_{16}$.

For example, IP multicast address 224.0.0.2 becomes Ethernet multicast address $01.00.5E.00.00.02_{16}$.

Interestingly, the mapping is not unique. Because IP multicast addresses have 28 significant bits that identify the multicast group, more than one multicast group may map onto the same Ethernet multicast address at the same time. The designers chose this scheme as a compromise. On one hand, using 23 of the 28 bits for a hardware address means most of the multicast address is included. The set of addresses is large enough so the chances of two groups choosing addresses with all low-order 23 bits identical is small. On the other hand, arranging for IP to use a fixed part of the Ethernet multicast address space makes debugging much easier and eliminates interference between IP and other protocols that share an Ethernet. The consequence of this design is that some multicast datagrams may be received at a host that are not destined for that host. Thus, the IP software must carefully check addresses on all incoming datagrams and discard any unwanted multicast datagrams.

17.10 Hosts And Multicast Delivery

We said that IP multicasting can be used on a single physical network or throughout an internet. In the former case, a host can send directly to a destination host merely by placing the datagram in a frame and using a hardware multicast address to which the receiver is listening. In the latter case, special *multicast routers* forward multicast datagrams among networks, so a host must send the datagram to a multicast router. Surprisingly, a host does not need to install a route to a multicast router, nor does the host's default route need to specify one. Instead, the technique a host uses to forward a multicast datagram to a router is unlike the routing lookup used for unicast and broadcast datagrams — the host merely uses the local network hardware's multicast capability to transmit the datagram. Multicast routers listen for all IP multicast transmissions; if a multicast router is present on the network, it will receive the datagram and forward it on to another network if necessary. Thus, the primary difference between local and nonlocal multicast lies in multicast routers, not in hosts.

17.11 Multicast Scope

The *scope* of a multicast group refers to the range of group members. If all members are on the same physical network, we say that the group's scope is restricted to one network. Similarly, if all members of a group lie within a single organization, we say that the group has a scope limited to one organization.

In addition to the group's scope, each multicast datagram has a scope which is defined to be the set of networks over which a given multicast datagram will be propagated. Informally, a datagram's scope is referred to as its *range*.

IP uses two techniques to control multicast scope. The first technique relies on the datagram's *time-to-live (TTL)* field to control its range. By setting the TTL to a small value, a host can limit the distance the datagram will be routed. For example, the standard specifies that control messages, which are used for communication between a host and a router on the same network, must have a TTL of 1. As a consequence, a router never forwards any datagram carrying control information because the TTL expires causing the router to discard the datagram. Similarly, if two applications running on a single host want to use IP multicast for interprocessor communication (e.g., for testing software), they can choose a TTL value of 0 to prevent the datagram from leaving the host. It is possible to use successively larger values of the TTL field to further extend the notion of scope. For example, some router vendors suggest configuring routers at a site to restrict multicast datagrams from leaving the site unless the datagram has a TTL greater than 15. We conclude that it is possible to use the TTL field in a datagram header to provide coarse-grain control over the datagram's scope.

Known as *administrative scoping*, the second technique used to control scoping consists of reserving parts of the address space for groups that are local to a given site or local to a given organization. According to the standard, routers in the Internet are forbidden from forwarding any datagram that has an address chosen from the restricted space. Thus, to prevent multicast communication among group members from accidentally reaching outsiders, an organization can assign the group an address that has local scope. Figure 17.2 shows examples of address ranges that correspond to administrative scoping.

17.12 Extending Host Software To Handle Multicasting

A host participates in IP multicast at one of three levels as Figure 17.3 shows:

Level	Meaning
0	**Host can neither send nor receive IP multicast**
1	**Host can send but not receive IP multicast**
2	**Host can both send and receive IP multicast**

Figure 17.3 The three levels of participation in IP multicast.

Modifications that allow a host to send IP multicast are not difficult. The IP software must allow an application program to specify a multicast address as a destination IP address, and the network interface software must be able to map an IP multicast address into the corresponding hardware multicast address (or use broadcast if the hardware does not support multicasting).

Extending host software to receive IP multicast datagrams is more complex. IP software on the host must have an API that allows an application program to declare that it wants to join or leave a particular multicast group. If multiple application programs join the same group, the IP software must remember to pass each of them a copy of datagrams that arrive destined for that group. If all application programs leave a group, the host must remember that it no longer participates in the group. Furthermore, as we will see in the next section, the host must run a protocol that informs the local multicast routers of its group membership status. Much of the complexity comes from a basic idea:

Hosts join specific IP multicast groups on specific networks.

That is, a host with multiple network connections may join a particular multicast group on one network and not on another. To understand the reason for keeping group membership associated with networks, remember that it is possible to use IP multicasting among local sets of machines. The host may want to use a multicast application to interact with machines on one physical net, but not with machines on another.

Because group membership is associated with particular networks, the software must keep separate lists of multicast addresses for each network to which the machine attaches. Furthermore, an application program must specify a particular network when it asks to join or leave a multicast group.

17.13 Internet Group Management Protocol

To participate in IP multicast on a local network, a host must have software that allows it to send and receive multicast datagrams. To participate in a multicast that spans multiple networks, the host must inform local multicast routers. The local routers contact other multicast routers, passing on the membership information and establishing routes. We will see later that the concept is similar to conventional route propagation among internet routers.

Before a multicast router can propagate multicast membership information, it must determine that one or more hosts on the local network have decided to join a multicast group. To do so, multicast routers and hosts that implement multicast must use the *Internet Group Management Protocol* (*IGMP*) to communicate group membership information. Because the current version is *2*, the protocol described here is officially known as *IGMPv2*.

IGMP is analogous to ICMP†. Like ICMP, it uses IP datagrams to carry messages. Also like ICMP, it provides a service used by IP. Therefore,

> *Although IGMP uses IP datagrams to carry messages, we think of it as an integral part of IP, not a separate protocol.*

Furthermore, IGMP is a standard for TCP/IP; it is required on all machines that receive IP multicast (i.e., all hosts and routers that participate at level *2*).

Conceptually, IGMP has two phases. Phase 1: When a host joins a new multicast group, it sends an IGMP message to the group's multicast address declaring its membership. Local multicast routers receive the message, and establish necessary routing by propagating the group membership information to other multicast routers throughout the internet. Phase 2: Because membership is dynamic, local multicast routers periodically poll hosts on the local network to determine whether any hosts still remain members of each group. If any host responds for a given group, the router keeps the group active. If no host reports membership in a group after several polls, the multicast router assumes that none of the hosts on the network remain in the group, and stops advertising group membership to other multicast routers.

17.14 IGMP Implementation

IGMP is carefully designed to avoid adding overhead that can congest networks. In particular, because a given network can include multiple multicast routers as well as hosts that all participate in multicasting, IGMP must avoid having all participants generate control traffic. There are several ways IGMP minimizes its effect on the network:

> First, all communication between hosts and multicast routers uses IP multicast. That is, when IGMP messages are encapsulated in an IP datagram for transmission, the IP destination address is a multicast address — routers

†Chapter 9 discusses ICMP, the Internet Control Message Protocol.

send general IGMP queries to the all hosts address, hosts send some IGMP messages to the all routers address, and both hosts and routers send IGMP messages that are specific to a group to the group's address. Thus, datagrams carrying IGMP messages are transmitted using hardware multicast if it is available. As a result, on networks that support hardware multicast, hosts not participating in IP multicast never receive IGMP messages.

Second, when polling to determine group membership, a multicast router sends a single query to request information about all groups instead of sending a separate message to each†. The default polling rate is 125 seconds, which means that IGMP does not generate much traffic.

Third, if multiple multicast routers attach to the same network, they quickly and efficiently choose a single router to poll host membership. Thus, the amount of IGMP traffic on a network does not increase as additional multicast routers are attached to the net.

Fourth, hosts do not respond to a router's IGMP query at the same time. Instead, each query contains a value, *N*, that specifies a maximum response time (the default is 10 seconds). When a query arrives, a host chooses a random delay between *0* and *N* which it waits before sending a response. In fact, if a given host is a member of multiple groups, the host chooses a different random number for each. Thus, a host's response to a router's query will be spaced randomly over *10* seconds.

Fifth, each host listens for responses from other hosts in the group, and suppresses unnecessary response traffic.

To understand why extra responses from group members can be suppressed, recall that a multicast router does not need to keep an exact record of group membership. Transmissions to the group are sent using hardware multicast. Thus, a router only needs to know whether at least one host on the network remains a member of the group. Because a query sent to the all systems address reaches every member of a group, each host computes a random delay and begins to wait. The host with smallest delay sends its response first. Because the response is sent to the group's multicast address, all other members receive a copy as does the multicast router. Other members cancel their timers and suppress transmission. Thus, in practice, only one host from each group responds to a request message.

17.15 Group Membership State Transitions

On a host, IGMP must remember the status of each multicast group to which the host belongs (i.e., a group from which the host accepts datagrams).‡. We think of a host as keeping a table in which it records group membership information. Initially, all entries in the table are unused. Whenever an application program on the host joins a

†The protocol does include a message type that allows a router to query a specific group, if necessary.

‡The *all systems group*, 224.0.0.1, is an exception — a host never reports membership in that group.

new group, IGMP software allocates an entry and fills in information about the group. Among the information, IGMP keeps a group reference counter which it initializes to *1*. Each time another application program joins the group, IGMP increments the reference counter in the entry. If one of the application programs terminates execution (or explicitly drops out of the group), IGMP decrements the group's reference counter. When the reference count reaches zero, the host informs multicast routers that it is leaving the multicast group.

The actions IGMP software takes in response to various events can best be explained by the state transition diagram in Figure 17.4.

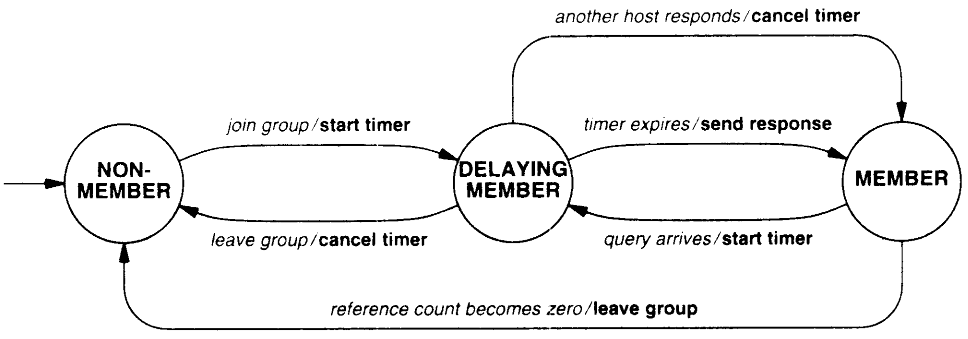

another host responds/cancel timer
join group/start timer
timer expires/send response
NON-MEMBER
DELAYING MEMBER
MEMBER
leave group/cancel timer
query arrives/start timer
reference count becomes zero/leave group

Figure 17.4 The three possible states of an entry in a host's multicast group table and transitions among them where each transition is labeled with an event and an action. The state transitions do not show messages sent when joining and leaving a group.

A host maintains an independent table entry for each group of which it is currently a member. As the figure shows, when a host first joins the group or when a query arrives from a multicast router, the host moves the entry to the *DELAYING MEMBER* state and chooses a random delay. If another host in the group responds to the router's query before the timer expires, the host cancels its timer and moves to the *MEMBER* state. If the timer expires, the host sends a response message before moving to the *MEMBER* state. Because a router only generates a query every 125 seconds, one expects the host to remain in the *MEMBER* state most of the time.

The diagram in Figure 17.4 omits a few details. For example, if a query arrives while the host is in the *DELAYING MEMBER* state, the protocol requires the host to reset its timer. More important, to maintain backward compatibility with IGMPv1, version *2* also handles version *1* messages, making it possible to use both IGMPv1 and IGMPv2 on the same network concurrently.

17.16 IGMP Message Format

As Figure 17.5 shows, IGMP messages used by hosts have a simple format.

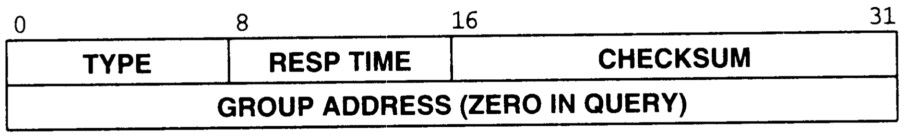


Figure 17.5 The format of the 8-octet IGMP message used for communication between hosts and routers.

Each IGMP message contains exactly eight octets. Field *TYPE* identifies the type of message, with the possible types listed in Figure 17.6. When a router polls for group membership, field labeled *RESP TIME* carries a maximum interval for the random delay that group members compute, measured in tenths of seconds. Each host in the group delays a random time between zero and the specified value before responding. As we said, the default is 10 seconds, which means all hosts in a group choose a random value between 0 and 10. IGMP allows routers set a maximum value in each query message to give managers control over IGMP traffic. If a network contains many hosts, a higher delay value further spreads out response times and, therefore, lowers the probability of having more than one host respond to the query. The *CHECKSUM* field contains a checksum for the message (IGMP checksums are computed over the IGMP message only, and use the same algorithm as TCP and IP). Finally, the *GROUP ADDRESS* field is either used to specify a particular group or contains zero to refer to all groups. When it sends a query to a specific group, a router fills in the *GROUP ADDRESS* field; hosts fill in the field when sending membership reports.

Type	Group Address	Meaning
0x11	unused (zero)	General membership query
0x11	used	Specific group membership query
0x16	used	Membership report
0x17	used	Leave group
0x12	used	Membership report (version 1)

Figure 17.6 IGMP message types used in version 2. The version 1 membership report message provides backward compatibility.

Note that IGMP does not provide a mechanism that allows a host to discover the IP address of a group — application software must know the group address before it can use IGMP to join the group. Some applications use permanently assigned addresses, some allow a manager to configure the address when the software is installed, and others obtain the address dynamically (e.g., from a server). In any case, IGMP provides no support for address lookup.

17.17 Multicast Forwarding And Routing Information

Although IGMP and the multicast addressing scheme described above specify how hosts interact with a local router and how multicast datagrams are transferred across a single network, they do not specify how routers exchange group membership information or how routers ensure that a copy of each datagram reaches all group members. More important, although multiple protocols have been proposed, no single standard has emerged for the propagation of multicast routing information. In fact, although much effort has been expended, there is no agreement on an overall plan — existing protocols differ in their goals and basic approach.

Why is multicast routing so difficult? Why not extend conventional routing schemes to handle multicast? The answer is that multicast routing differs from conventional routing in fundamental ways because multicast forwarding differs from conventional forwarding. To appreciate some of the differences, consider multicast forwarding over the architecture that Figure 17.7 depicts.

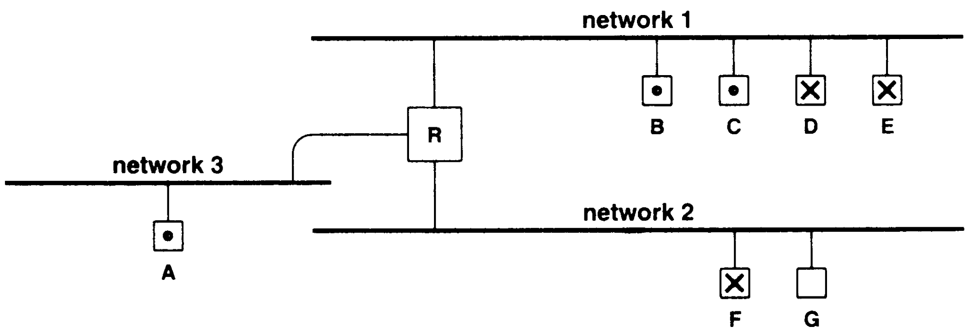


Figure 17.7 A simple internet with three networks connected by a router that illustrates multicast forwarding. Hosts marked with a dot participate in one multicast group while those marked with an "x" participate in another.

17.17.1 Need For Dynamic Routing

Even for the simple topology shown in the figure, multicast forwarding differs from unicast forwarding. For example, the figure shows two multicast groups: the group denoted by a dot has members *A*, *B*, and *C*, and the group denoted by a cross has members *D*, *E*, and *F*. The dotted group has no members on network *2*. To avoid wasting bandwidth unnecessarily, the router should never send packets intended for the dotted group across network *2*. However, a host can join any group at any time — if the host is the first on its network to join the group, multicast routing must be changed to include the network. Thus, we come to an important difference between conventional routing and mulficast routing:

> *Unlike unicast routing in which routes change only when the topology changes or equipment fails, multicast routes can change simply because an application program joins or leaves a multicast group.*

17.17.2 Insufficiency Of Destination Routing

The example in Figure 17.7 illustrates another aspect of multicast routing. If host *F* and host *E* each send a datagram to the cross group, router *R* will receive and forward them. Because both datagrams are directed at the same group, they have the same destination address. However, the correct forwarding actions differ: *R* sends the datagram from *E* to net *2*, and sends the datagram from *F* to net *1*. Interestingly, when it receives a datagram destinated for the cross group sent by host *A*, the router uses a third action: it forwards two copies, one to net *1* and the other to net *2*. Thus, we see the second major difference between conventional forwarding and multicast forwarding:

> *Multicast forwarding requires a router to examine more than the destination address.*

17.17.3 Arbitrary Senders

The final feature of multicast routing illustrated by Figure 17.7 arises because IP allows an arbitrary host, one that is not necessarily a member of the group, to send a datagram to the group. In the figure, for example, host *G* can send a datagram to the dotted group even though *G* is not a member of any group and there are no members of the dotted group on *G's* network. More important, as it travels through the internet, the datagram may pass across other networks that have no group members attached. Thus, we can summarize:

> *A multicast datagram may originate on a computer that is not part of the multicast group, and may be routed across networks that do not have any group members attached.*

17.18 Basic Multicast Routing Paradigms

We know from the example above that multicast routers use more than the destination address to forward datagrams, so the question arises: "exactly what information does a multicast router use when deciding how to forward a datagram?" The answer lies in understanding that because a multicast destination represents a set of computers, an optimal forwarding system will reach all members of the set without sending a datagram across a given network twice. Although a single multicast router such as the one in Figure 17.7 can simply avoid sending a datagram back over the interface on which it arrives, using the interface alone will not prevent a datagram from being forwarded among a set of routers that are arranged in a cycle. To avoid such routing loops, multicast routers rely on the datagram's source address.

One of the first ideas to emerge for multicast forwarding was a form of broadcasting described earlier. Known as *Reverse Path Forwarding* (*RPF*),† the scheme uses a datagram's source address to prevent the datagram from traveling around a loop repeatedly. To use RPF, a multicast router must have a conventional routing table with shortest paths to all destinations. When a datagram arrives, the router extracts the source address, looks it up in the local routing table, and finds *I*, the interface that leads to the source. If the datagram arrived over interface *I*, the router forwards a copy to each of the other interfaces; otherwise, the router discards the copy.

Because it ensures that a copy of each multicast datagram is sent across every network in the internet, the basic RPF scheme guarantees that every host in a multicast group will receive a copy of each datagram sent to the group. However, RPF alone is not used for multicast routing because it wastes bandwidth by transmitting multicast datagrams over networks that neither have group members nor lead to group members.

To avoid propagating multicast datagrams where they are not needed, a modified form of RPF was invented. Known as *Truncated Reverse Path Forwarding* (*TRPF*) or *Truncated Reverse Path Broadcasting* (*TRPB*), the scheme follows the RPF algorithm, but further restricts propagation by avoiding paths that do not lead to group members. To use TRPF, a multicast router needs two pieces of information: a conventional routing table and a list of multicast groups reachable through each network interface. When a multicast datagram arrives, the router first applies the RPF rule. If RPF specifies discarding the copy, the router does so. However, if RPF specifies transmitting the datagram over a particular interface, the router first makes an additional check to verify that one or more members of the group designated in the datagram's destination address are reachable over the interface. If no group members are reachable over the interface, the router skips that interface, and continues examining the next one. In fact, we can now understand the origin of the term *truncated* — a router truncates forwarding when no more group members lie along the path.

We can summarize:

> *When making a forwarding decision, a multicast router uses both the datagram's source and destination addresses. The basic forwarding mechanism is known as Truncated Reverse Path Forwarding.*

†Reverse path forwarding is sometimes called *Reverse Path Broadcasting* (*RPB*).

17.19 Consequences Of TRPF

Although TRPF guarantees that each member of a multicast group receives a copy of each datagram sent to the group, it has two surprising consequences. First, because it relies on RPF to prevent loops, TRPF delivers an extra copy of datagrams to some networks just like conventional RPF. Figure 17.8 illustrates how duplicates arise.

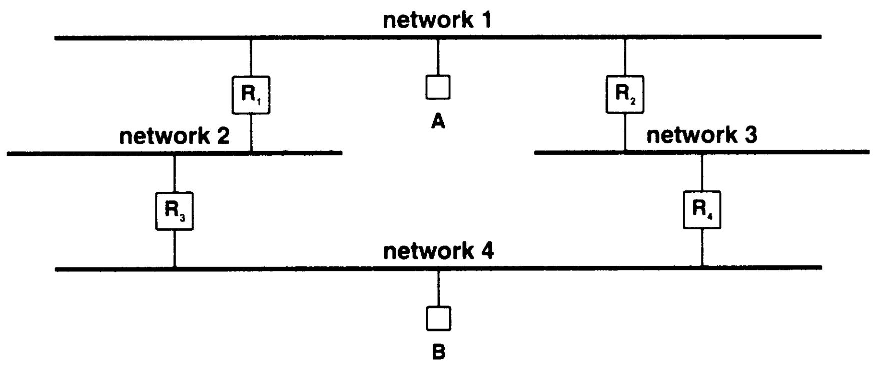
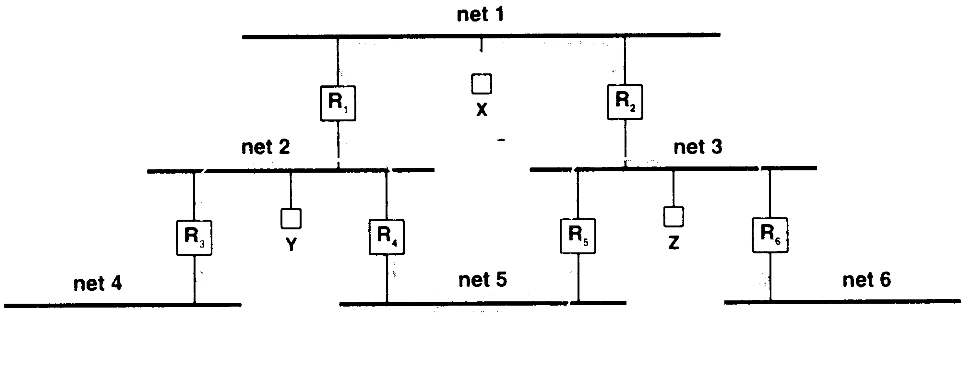


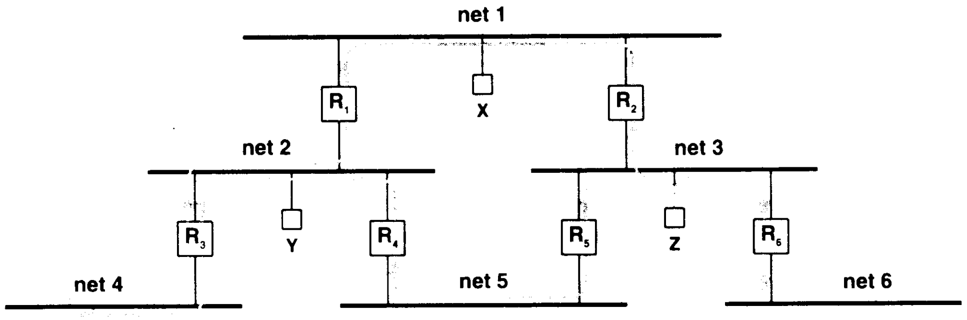
Figure 17.8 A topology that causes an RPF scheme to deliver multiple copies of a datagram to some destinations.

In the figure, when host *A* sends a datagram, routers R_1 and R_2 each receive a copy. Because the datagram arrives over the interface that lies along the shortest path to *A*, R_1 forwards a copy to network *2*, and R_2 forwards a copy to network *3*. When it receives a copy from network *2* (the shortest path to *A*), R_3 forwards the copy to network *4*. Unfortunately, R_4 also forwards a copy to network *4*. Thus, although RPF allows R_3 and R_4 to prevent a loop by discarding the copy that arrives over network *4*, host *B* receives two copies of the datagram.

A second surprising consequence arises because TRPF uses both source and destination addresses when forwarding datagrams: delivery depends on a datagram's source. For example, Figure 17.9 shows how multicast routers forward datagrams from two different sources across a fixed topology.



(a)



(b)

Figure 17.9 Examples of paths a multicast datagram follows under TRPF assuming the source is (a) host *X*, and (b) host *Z*, and the group has a member on each of the networks. The number of copies received depends on the source.

As the figure shows, the source affects both the path a datagram follows to reach a given network as well as the delivery details. For example, in part (a) of the figure, a transmission by host *X* causes TRPF to deliver two copies of the datagram to network *5*. In part (b), only one copy of a transmission by host *Z* reaches network *5*, but two copies reach networks *2* and *4*.

17.20 Multicast Trees

Researchers use graph theory terminology to describe the set of paths from a given source to all members of a multicast group: they say that the paths define a graph-theoretic *tree*†, which is sometimes called a *forwarding tree* or a *delivery tree*. Each multicast router corresponds to a *node* in the tree, and a network that connects two routers corresponds to an *edge* in the tree. The source of a datagram is the *root* or *root node* of the tree. Finally, the last router along each of the paths from the source is called a *leaf* router. The terminology is sometimes applied to networks as well — researchers call a network hanging off a leaf router a *leaf network*.

As an example of the terminology, consider Figure 17.9. Part *a* shows a tree with root *X*, and leaves R_3, R_4, R_5, and R_6. Technically, part *b* does not show a tree because router R_3 lies along two paths. Informally, researchers often overlook the details and refer to such graphs as trees.

The graph terminology allows us to express an important principle:

> *A multicast forwarding tree is defined as a set of paths through multicast routers from a source to all members of a multicast group. For a given multicast group, each possible source of datagrams can determine a different forwarding tree.*

One of the immediate consequences of the principle concerns the size of tables used to forward multicast. Unlike conventional routing tables, each entry in a multicast table is identified by a pair:

(multicast group, source)

Conceptually, *source* identifies a single host that can send datagrams to the group (i.e., any host in the internet). In practice, keeping a separate entry for each host is unwise because the forwarding trees defined by all hosts on a single network are identical. Thus, to save space, routing protocol use a network prefix as a *source*. That is, each router defines one forwarding entry that is used for all hosts on the same physical network.

Aggregating entries by network prefix instead of by host address reduces the table size dramatically. However, multicast routing tables can grow much larger than conventional routing tables. Unlike a conventional table in which the size is proportional to the number of networks in the internet, a multicast table has size proportional to the product of the number of networks in the internet and the number of multicast groups.

†A graph is a tree if it does not contain any cycles (i.e., a router does not appear on more than one path).

17.21 The Essence Of Multicast Routing

Observant readers may have noticed an inconsistency between the features of IP multicasting and TRPF. We said that TRPF is used instead of conventional RPF to avoid unnecessary traffic: TRPF does not forward a datagram to a network unless that network leads to at least one member of the group. Consequently, a multicast router must have knowledge of group membership. We also said that IP allows any host to join or leave a multicast group at any time, which results in rapid membership changes. More important, membership does not follow local scope — a host that joins may be far from some router that is forwarding datagrams to the group. So, group membership information must be propagated across the internet.

The issue of membership is central to routing; all multicast routing schemes provide a mechanism for propagating membership information as well as a way to use the information when forwarding datagrams. In general, because membership can change rapidly, the information available at a given router is imperfect, so routing may lag changes. Therefore, a multicast design represents a tradeoff between routing traffic overhead and inefficient data transmission. On one hand, if group membership information is not propagated rapidly, multicast routers will not make optimal decisions (i.e., they either forward datagrams across some networks unnecessarily or fail to send datagrams to all group members). On the other hand, a multicast routing scheme that communicates every membership change to every router is doomed because the resulting traffic can overwhelm an internet. Each design chooses a compromise between the two extremes.

17.22 Reverse Path Multicasting

One of the earliest forms of multicast routing was derived from TRPF. Known as *Reverse Path Multicast* (*RPM*), the scheme extends TRPF to make it more dynamic. Three assumptions underlie the design. First, it is more important to ensure that a multicast datagram reaches each member of the group to which it is sent than to eliminate unnecessary transmission. Second, multicast routers each contain a conventional routing table that has correct information. Third, multicast routing should improve efficiency when possible (i.e. eliminate needless transmission).

RPM uses a two step process. When it begins, RPM uses the RPF broadcast scheme to send a copy of each datagram across all networks in the internet. Doing so ensures that all group members receive a copy. Simultaneously, RPM proceeds to have multicast routers inform one another about paths that do not lead to group members. Once it learns that no group members lie along a given path, a router stops forwarding along that path.

How do routers learn about the location of group members? As in most multicast routing schemes, RPM propagates membership information bottom-up. The information starts with hosts that choose to join or leave groups. Hosts communicate membership information with their local router by using IGMP. Thus, although a multicast router does not know about distant group members, it does know about local members (i.e. members on each of its directly-attached networks). As a consequence, routers attached to leaf networks can decide whether to forward over the leaf network — if a leaf network contains no members for a given group, the router connecting that network to the rest of the internet does not forward on the network. In addition to taking local action, the leaf router informs the next router along the path back to the source. Once it learns that no group members lie beyond a given network interface, the next router stops forwarding datagrams for the group across the network. When a router finds that no group members lie beyond it, the router informs the next router along the path to the root.

Using graph-theoretic terminology, we say that when a router learns that a group has no members along a path and stops forwarding, it has *pruned* (i.e., removed) the path from the forwarding tree. In fact, RPM is called a *broadcast and prune* strategy because a router broadcasts (using RPF) until it receives information that allows it to prune a path. Researchers also use another term for the RPM algorithm: they say that the system is *data-driven* because a router does not send group membership information to any other routers until datagrams arrive for that group.

In the data-driven model, a router must also handle the case where a host decides to join a particular group after the router has pruned the path for that group. RPM handles joins bottom-up: when a host informs a local router that it has joined a group, the router consults its record of the group and obtains the address of the router to which it had previously sent a prune request. The router sends a new message that undoes the effect of the previous prune and causes datagrams to flow again. Such messages are known as *graft requests*, and the algorithm is said to graft the previously pruned branch back onto the tree.

17.23 Distance Vector Multicast Routing Protocol

One of the first multicast routing protocols is still in use in the global Internet. Known as the *Distance Vector Multicast Routing Protocol* (*DVMRP*), the protocol allows multicast routers to pass group membership and routing information among themselves. DVMRP resembles the RIP protocol described in Chapter 16, but has been extended for multicast. In essence, the protocol passes information about current multicast group membership and the cost to transfer datagrams between routers. For each possible (group, source) pair, the routers impose a forwarding tree on top of the physical interconnections. When a router receives a datagram destined for an IP multicast group, it sends a copy of the datagram out over the network links that correspond to branches in the forwarding tree†.

Interestingly, DVMRP defines an extended form of IGMP used for communication between a pair of multicast routers. It specifies additional IGMP message types that allow routers to declare membership in a multicast group, leave a multicast group, and interrogate other routers. The extensions also provide messages that carry routing information, including cost metrics.

†DVMRP changed substantially between version *2* and *3* when it incorporated the RPM algorithm described above.

17.24 The Mrouted Program

Mrouted is a well-known program that implements DVMRP for UNIX systems. Like *routed*†, *mrouted* cooperates closely with the operating system kernel to install multicast routing information. Unlike *routed*, however, *mrouted* does not use the standard routing table. Instead, it can be used only with a special version of UNIX known as a *multicast kernel.* A UNIX multicast kernel contains a special multicast routing table as well as the code needed to forward multicast datagrams. *Mrouted* handles:

- *Route propagation. Mrouted* uses DVMRP to propagate multicast routing information from one router to another. A computer running *mrouted* interprets multicast routing information, and constructs a multicast routing table. As expected, each entry in the table specifies a (group, source) pair and a corresponding set of interfaces over which to forward datagrams that match the entry. *Mrouted* does not replace conventional route propagation protocols; a computer usually runs *mrouted* in addition to standard routing protocol software.
- *Multicast tunneling.* One of the chief problems with internet multicast arises because not all internet routers can forward multicast datagrams. *Mrouted* can arrange to *tunnel* a multicast datagram from one router to another through intermediate routers that do not participate in multicast routing.

Although a single *mrouted* program can perform both tasks, a given computer may not need both functions. To allow a manager to specify exactly how it should operate, *mrouted* uses a configuration file. The configuration file contains entries that specify which multicast groups *mrouted* is permitted to advertise on each interface, and how it should forward datagrams. Furthermore, the configuration file associates a metric and threshold with each route. The metric allows a manager to assign a cost to each path (e.g., to ensure that the cost assigned to a path over a local area network will be lower than the cost of a path across a slow serial link). The threshold gives the minimum IP *time to live* (*TTL*) that a datagram needs to complete the path. If a datagram does not have a sufficient TTL to reach its destination, a multicast kernel does not forward the datagram. Instead, it discards the datagram, which avoids wasting bandwidth.

Multicast tunneling is perhaps the most interesting capability of *mrouted.* A tunnel is needed when two or more hosts wish to participate in multicast applications, and one or more routers along the path between the participating hosts do not run multicast routing software. Figure 17.10 illustrates the concept.

†Recall that *routed* is the UNIX program that implements RIP.

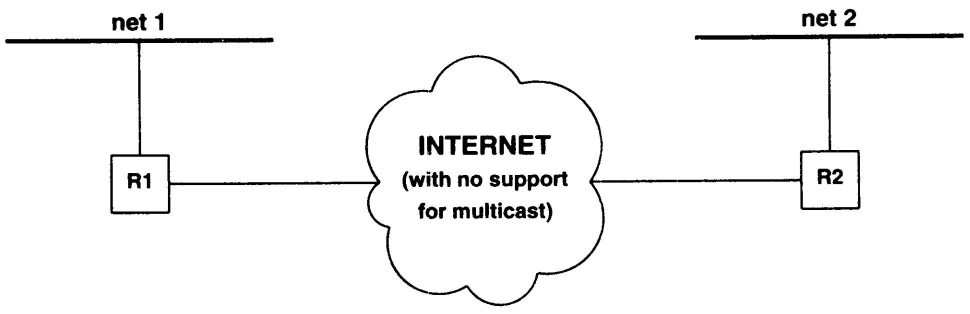


Figure 17.10 An example internet configuration that requires multicast tunneling for computers attached to networks *1* and *2* to participate in multicast communication. Routers in the internet that separates the two networks do not propagate multicast routes, and cannot forward datagrams sent to a multicast address.

To allow hosts on networks *1* and *2* to exchange multicast, managers of the two routers configure an *mrouted tunnel*. The tunnel merely consists of an agreement between the *mrouted* programs running on the two routers to exchange datagrams. Each router listens on its local net for datagrams sent to the specified multicast destination for which the tunnel has been configured. When a multicast datagram arrives that has a destination address equal to one of the configured tunnels, *mrouted* encapsulates the datagram in a conventional unicast datagram and sends it across the internet to the other router. When it receives a unicast datagram through one of its tunnels, *mrouted* extracts the multicast datagram, and then forwards according to its multicast routing table.

The encapsulation technique that *mrouted* uses to tunnel datagrams is known as *IP-in-IP*. Figure 17.11 illustrates the concept.

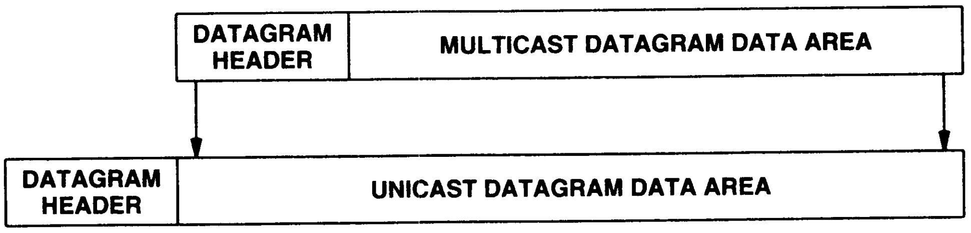


Figure 17.11 An illustration of IP-in-IP encapsulation in which one datagram is placed in the data area of another. A pair of multicast routers use the encapsulation to communicate when intermediate routers do not understand multicasting.

As the figure shows, IP-in-IP encapsulation preserves the original multicast datagram, including the header, by placing it in the data area of a conventional unicast datagram. On the receiving machine, the multicast kernel extracts and processes the multicast datagram as if it arrived over a local interface. In particular, once it extracts the multicast datagram, the receiving machine must decrement the time to live field in the header by one before forwarding. Thus, when it creates a tunnel, *mrouted* treats the internet connecting two multicast routers like a single, physical network. Note that the outer, unicast datagram has its own time to live counter, which operates independently from the time to live counter in the multicast datagram header. Thus, it is possible to limit the number of physical hops across a given tunnel independent of the number of logical hops a multicast datagram must visit on its journey from the original source to the ultimate destination.

Multicast tunnels form the basis of the Internet's *Multicast Backbone* (*MBONE*). Many Internet sites participate in the MBONE; the MBONE allows hosts at participating sites to send and receive multicast datagrams, which are then propagated to all other participating sites. The MBONE is often used to propagate audio and video (e.g., for teleconferences).

To participate in the MBONE, a site must have at least one multicast router connected to at least one local network. Another site must agree to tunnel traffic, and a tunnel is configured between routers at the two sites. When a host at the site sends a multicast datagram, the local router at the host's site receives a copy, consults its multicast routing table, and forwards the datagram over the tunnel using IP-in-IP. When it receives a multicast datagram over a tunnel, a multicast router removes the outer encapsulation, and then forwards the datagram according to the local multicast routing table.

The easiest way to understand the MBONE is to think of it as a virtual network built on top of the Internet (which is a virtual network). Conceptually, the MBONE consists of multicast routers that are interconnected by a set of point-to-point networks. Some of the conceptual point-to-point connections coincide with physical networks; others are achieved by tunneling. The details are hidden from the multicast routing software. Thus, when *mrouted* computes a multicast forwarding tree for a given (group, source), it thinks of a tunnel as a single link connecting two routers.

Tunneling has two consequences. First, because some tunnels are much more expensive than others, they cannot all be treated equally. *Mrouted* handles the problem by allowing a manager to assign a cost to each tunnel, and uses the costs when choosing routes. Typically, a manager assigns a cost that reflects the number of hops in the underlying internet. It is also possible to assign costs that reflect administrative boundaries (e.g., the cost assigned to a tunnel between two sites in the same company is assigned a much lower cost than a tunnel to another company). Second, because DVMRP forwarding depends on knowing the shortest path to each source, and because multicast tunnels are completely unknown to conventional routing protocols, DVMRP must compute its own version of unicast forwarding that includes the tunnels.

17.25 Alternative Protocols

Although DVMRP has been used in the MBONE for many years, as the Internet grew, the IETF became aware of its limitations. Like RIP, DVMRP uses a small value for *infinity*. More important, the amount of information DVMRP keeps is overwhelming — in addition to entries for each active (group, source), it must also store entries for previously active groups so it knows where to send a graft message when a host joins a group that was pruned. Finally, DVMRP uses a broadcast-and-prune paradigm that generates traffic on all networks until membership information can be propagated. Ironically, DVMRP also uses a distance-vector algorithm to propagate membership information, which makes propagation slow.

Taken together, the limitations of DVMRP mean that it cannot scale to handle a large number of routers, larger numbers of multicast groups, or rapid changes in membership. Thus, DVMRP is inappropriate as a general-purpose multicast routing protocol for the global Internet.

To overcome the limitations of DVMRP, the IETF has investigated other multicast protocols. Efforts have resulted in several designs, including *Core Based Trees* (*CBT*), *Protocol Independent Multicast* (*PIM*), and *Multicast extensions to OSPF* (*MOSPF*). Each is intended to handle the problems of scale, but does so in a slightly different way. Although all these protocols have been implemented and both PIM and MOSPF have been used in parts of the MBONE, none of them is a required standard.

17.26 Core Based Trees (CBT)

CBT avoids broadcasting and allows all sources to share the same forwarding tree whenever possible. To avoid broadcasting, CBT does not forward multicasts along a path until one or more hosts along that path join the multicast group. Thus, CBT reverses the fundamental scheme used by DVMRP — instead of forwarding datagrams until negative information has been propagated, CBT does not forward along a path until positive information has been received. We say that instead of using the data-driven paradigm, CBT uses a *demand-driven* paradigm.

The demand-driven paradigm in CBT means that when a host uses IGMP to join a particular group, the local router must then inform other routers before datagrams will be forwarded. Which router or routers should be informed? The question is critical in all demand-driven multicast routing schemes. Recall that in a data-driven scheme, a router uses the arrival of data traffic to know where to send routing messages (it propagates routing messages back over networks from which the traffic arrives). However, in a positive-information scheme, no traffic will arrive for a group until the membership information has been propagated.

CBT uses a combination of static and dynamic algorithms to build a multicast forwarding tree. To make the scheme scalable, CBT divides the internet into *regions*, where the size of a region is determined by network administrators. Within each region, one of the routers is designated as a *core router*; other routers in the region must either be configured to know the core for their region, or use a dynamic *discovery mechanism* to find it. In any case, core discovery only occurs when a router boots.

Knowledge of a core is important because it allows multicast routers in a region to form a *shared tree* for the region. As soon as a host joins a multicast group, the local router that receives the host request, *L*, generates a CBT *join request* which it sends to the core using conventional unicast routing. Each intermediate router along the path to the core examines the request. As soon as the request reaches a router *R* that is already part of the CBT shared tree, *R* returns an acknowledgement, passes the group membership information on to its parent, and begins forwarding traffic for the group. As the acknowledgement passes back to the leaf router, intermediate routers examine the message, and configure their multicast routing table to forward datagrams for the group. Thus, router *L* is linked into the forwarding tree at router *R*.

We can summarize:

> *Because CBT uses a demand-driven paradigm, it divides the internet into regions and designates a* core router *for each region; other routers in the region dynamically build a forwarding tree by sending* join requests *to the core.*

CBT includes a facility for tree maintenance that detects when a link between a pair of routers fails. To detect failure, each router periodically sends a CBT *echo request* to its parent in the tree (i.e., the next router along the path to the core). If the request is unacknowledged, CBT informs any routers that depend on it, and proceeds to rejoin the tree at another point.

17.27 Protocol Independent Multicast (PIM)

In reality, PIM consists of two independent protocols that share little beyond the name and basic message header formats: *PIM - Dense Mode* (*PIM-DM*) and *PIM - Sparse Mode* (*PIM-SM*). The distinction arises because no single protocol works well in all possible situations. In particular, PIM's dense mode is designed for a LAN environment in which all, or nearly all, networks have hosts listening to each multicast group; whereas, PIM's sparse mode is deigned to accommodate a wide area environment in which the members of a given multicast group occupy a small subset of all possible networks.

17.27.1 PIM Dense Mode (PIM-DM)

Because PIM's dense mode assumes low-delay networks that have plenty of bandwidth, the protocol has been optimized to guarantee delivery rather than to reduce overhead. Thus, PIM-DM uses a broadcast-and-prune approach similar to DVMRP — it begins by using RPF to broadcast each datagram to every group, and only stops sending when it receives explicit prune requests.

17.27.2 Protocol Independence

The greatest difference between DVMRP and PIM dense mode arises from the information PIM assumes is available. In particular, in order to use RPF, PIM-DM dense mode requires traditional unicast routing information — the shortest path to each destination must be known. Unlike DVMRP, however, PIM-DM does not contain facilities to propagate conventional routes. Instead, it assumes the router also uses a conventional routing protocol that computes the shortest path to each destination, installs the route in the routing table, and maintains the route over time. In fact, part of PIM-DM's *protocol independence* refers to its ability to co-exist with standard routing protocols. Thus, a router can use any of the routing protocols discussed (e.g., RIP, or OSPF) to maintain correct unicast routes, and PIM's dense mode can use routes produced by any of them. To summarize:

Although it assumes a correct unicast routing table exists, PIM dense mode does not propagate unicast routes. Instead, it assumes each router also runs a conventional routing protocol which maintains the unicast routes.

17.27.3 PIM Sparse Mode (PIM-SM)

PIM's sparse mode can be viewed as an extension of basic concepts from CBT. Like CBT, PIM-SM is demand-driven. Also like CBT, PIM-SM needs a point to which join messages can be sent. Therefore, sparse mode designates a router called a *Rendezvous Point* (*RP*) that is the functional equivalent of a CBT core. When a host joins a multicast group, the local router unicasts a *join* request to the RP; routers along the path examine the message, and if any router is already part of the tree, the router intercepts the message and replies. Thus, PIM-SM builds a shared forwarding tree for each group like CBT, and the trees are rooted at the rendezvous point†.

The main conceptual difference between CBT and PIM-SM arises from sparse mode's ability to optimize connectivity through reconfiguration. For example, instead of a single RP, each sparse mode router maintains a set of potential RP routers, with one selected at any time. If the current RP becomes unreachable (e.g., because a network failure causes disconnection), PIM-SM selects another RP from the set and starts rebuilding the forwarding tree for each multicast group. The next section considers a more significant reconfiguration.

17.27.4 Switching From Shared To Shortest Path Trees

In addition to selecting an alternative RP, PIM-SM can switch from the shared tree to a *Shortest Path tree* (*SP tree*). To understand the motivation, consider the network interconnection that Figure 17.12 illustrates.

†When an arbitrary host sends a datagram to a multicast group, the datagram is tunneled to the RP for the group, which then multicasts the datagram down the shared tree.

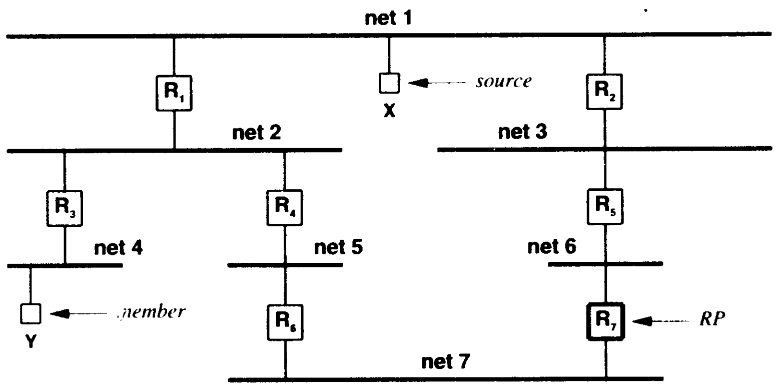


Figure 17.12 A set of networks with a rendezvous point and a multicast group that contains two members. The demand-driven strategy of building a shared tree to the rendezvous results in nonoptimal routing.

In the figure, router R_7 has been selected as the RP. Thus, routers join the shared tree by sending along a path to R_7. For example, assume hosts X and Y have joined a particular multicast group. The path to the shared tree from host X consists of routers R_2, R_5, and R_7, and the path from host Y to the shared tree consists of routers R_3, R_4, R_6, and R_7.

Although the shared tree approach forms shortest paths from each host to the RP, it may not optimize routing. In particular, if group members are not close to the RP, the inefficiency can be significant. For example, the figure shows that when host X sends a datagram to the group, the datagram is routed from X to the RP and from the RP to Y. Thus, the datagram must pass through six routers. However, the optimal (i.e., shortest) path from X to Y only contains two routers (R_1 and R_3).

PIM sparse mode includes a facility to allow a router to choose between the shared tree or a shorest path tree to the source (sometimes called a *source tree*). Although switching trees is conceptually straightforward, many details complicate the protocol. For example, most implementations use the receipt of traffic to trigger the change — if the traffic from a particular source exceeds a preset threshold, the router begins to establish a shortest path†. Unfortunately, traffic can change rapidly, so routers must apply hysteresis to prevent oscillations. Furthermore, the change requires routers along the shortest path to cooperate; all routers must agree to forward datagrams for the group. Interestingly, because the change affects only a single source, a router must continue its connection to the shared tree so it can continue to receive from other sources. More important, it must keep sufficient routing information to avoid forwarding multiple copies of each datagram from a (group, source) pair for which a shortest path tree has been established.

†The implementation from at least one vendor starts building a shortest path immediately (i.e., the traffic threshold is zero).

17.28 Multicast Extensions To OSPF (MOSPF)

So far, we have seen that multicast routing protocols like PIM can use information from a unicast routing table to form delivery trees. Researchers have also investigated a broader question: "how can multicast routing benefit from additional information that is gathered by conventional routing protocols?" In particular, a link state protocol such as OSPF provides each router with a copy of the internet topology. More specifically, OSPF provides the router with the topology of its OSPF *area.*

When such information is available, multicast protocols can indeed use it to compute a forwarding tree. The idea has been demonstrated in a protocol known as *Multicast extensions to OSPF* (*MOSPF*), which uses OSPF's topology database to form a forwarding tree for each source. MOSPF has the advantage of being *demand-driven,* meaning that the traffic for a particular group is not propagated until it is needed (i.e., because a host joins or leaves the group). The disadvantage of a demand-driven scheme arises from the cost of propagating routing information — all routers in an area must maintain membership about every group. Furthermore, the information must be synchronized to ensure that every router has exactly the same database. As a consequence, MOSPF sends less data traffic, but sends more routing information than data-driven protocols.

Although MOSPF's paradigm of sending all group information to all routers works within an area, it cannot scale to an arbitrary internet. Thus, MOSPF defines inter-area multicast routing in a slightly different way. OSPF designates one or more routers in an area to be an *Area Border Router* (*ABR*) which then propagates routing information to other areas. MOSPF further designates one or more of the area's ABRs to be a *Multicast Area Border Router MABR* which propagates group membership information to other areas. MABRs do not implement a symmetric transfer. Instead, MABRs use a core approach — they propagate membership information from their area to the backbone area, but do not propagate information from the backbone down.

An MABR can propagate multicast information to another area without acting as an active receiver for traffic. Instead, each area designates a router to receive multicast on behalf of the area. When an outside area sends in multicast traffic, traffic for all groups in the area is sent to the designated receiver, which is sometimes called a *multicast wildcard receiver.*

17.29 Reliable Multicast And ACK Implosions

The term *reliable multicast* refers to any system that uses multicast delivery, but also guarantees that all group members receive data in order without any loss, duplication, or corruption. In theory, reliable multicast combines the advantage of a forwarding scheme that is more efficient than broadcast with the advantage of having all data arrive intact. Thus, reliable multicast has great potential benefit and applicability (e.g., a stock exchange could use reliable multicast to deliver stock prices to many destinations).

In practice, reliable multicast is not as general or straightforward as it sounds. First, if a multicast group has multiple senders, the notion of delivering datagrams "in sequence" becomes meaningless. Second, we have seen that widely used multicast forwarding schemes such as RPF can produce duplication even on small internets. Third, in addition to guarantees that all data will eventually arrive, applications like audio or video expect reliable systems to bound the delay and jitter. Fourth, because reliability requires acknowledgements and a multicast group can have an arbitrary number of members, traditional reliable protocols require a sender to handle an arbitrary number of acknowledgements. Unfortunately, no computer has enough processing power to do so. We refer to the problem as an *ACK implosion*; it has become the main focus of much research.

To overcome the ACK implosion problem, reliable multicast protocols take a hierarchical approach in which multicasting is restricted to a single source†. Before data is sent, a forwarding tree is established from the source to all group members, and *acknowledgement points* must be identified.

An acknowledgement point, which is also known as an *acknowledgement aggregator* or *designated router* (*DR*), consists of a router in the forwarding tree that agrees to cache copies of the data and process acknowledgements from routers or hosts further down the tree. If a retransmission is required, the acknowledgement point obtains a copy from its cache.

Most reliable multicast schemes use negative rather than positive acknowledgements — the host does not respond unless a datagram is lost. To allow a host to detect loss, each datagram must be assigned a unique sequence number. When it detects loss, a host sends a *NACK* to request retransmission. The NACK propagates along the forwarding tree toward the source until it reaches an acknowledgement point. The acknowledgement point processes the NACK, and retransmits a copy of the lost datagram along the forwarding tree.

How does an acknowledgement point ensure that it has a copy of all datagrams in the sequence? It uses the same scheme as a host. When a datagram arrives, the acknowledgement point checks the sequence number, places a copy in its memory, and then proceeds to propagate the datagram down the forwarding tree. If it finds that a datagram is missing, the acknowledgement point sends a NACK up the tree toward the source. The NACK either reaches another acknowledgement point that has a copy of the datagram (in which case that acknowledgement point transmits a second copy), or the NACK reaches the source (which retransmits the missing datagram).

The choice of branching topology and acknowledgement points is crucial to the success of a reliable multicast scheme. Without sufficient acknowledgement points, a missing datagram can cause an ACK implosion. In particular, if a given router has many descendants, a lost datagram can cause that router to be overrun with retransmission requests. Unfortunately, automating selection of acknowledgement points has not turned out to be simple. Consequently, many reliable multicast protocols require manual configuration. Thus, multicast is best suited to: services that tend to persist over long periods of time, topologies that do not change rapidly, and situations where intermediate routers agree to serve as acknowledgement points.

†Note that a single source does not limit functionality because the source can agree to forward any message it receives via unicast. Thus, an arbitrary host can send a packet to the source, which then multicasts the packet to the group.

Is there an alternative approach to reliability? Some researchers are experimenting with protocols that incorporate redundant information to reduce or eliminate retransmission. One scheme sends redundant datagrams. Instead of sending a single copy of each datagram, the source sends *N* copies (typically *2* or *3*). Redundant datagrams work especially well when routers implement a *Random Early Discard* (*RED*) strategy because the probability of more than one copy being discarded is extremely small.

Another approach to redundancy involves *forward error-correcting codes.* Analogous to the error-correcting codes used with audio CDs, the scheme requires a sender to incorporate error-correction information into each datagram in a data stream. If one datagram is lost, the error correcting code contains sufficient redundant information to allow a receiver to reconstruct the missing datagram without requesting a retransmission.

17.30 Summary

IP multicasting is an abstraction of hardware multicasting. It allows delivery of a datagram to multiple destinations. IP uses class *D* addresses to specify multicast delivery; actual transmission uses hardware multicast, it it is available.

IP multicast groups are dynamic: a host can join or leave a group at any time. For local multicast, hosts only need the ability to send and receive multicast datagrams. However, IP multicasting is not limited to a single physical network – multicast routers propagate group membership information and arrange routing so that each member of a multicast group receives a copy of every datagram sent to that group.

Hosts communicate their group membership to multicast routers using IGMP. IGMP has been designed to be efficient and to avoid using network resources. In most cases, the only traffic IGMP introduces is a periodic message from a multicast router and a single reply for each multicast group to which hosts on that network belong.

A variety of protocols have been designed to propagate multicast routing information across an internet. The two basic approaches are data-driven and demand-driven. In either case, the amount of information in a multicast forwarding table is much larger than in a unicast routing table because multicasting requires entries for each (group, source) pair.

Not all routers in the global Internet propagate multicast routes or forward multicast traffic. Groups at two or more sites, separated by an internet that does not support multicast routing, can use an IP tunnel to transfer multicast datagrams. When using a tunnel, a program encapsulates a multicast datagram in a conventional unicast datagram. The receiver must extract and handle the multicast datagram.

Reliable multicast refers to a scheme that uses multicast forwarding but offers reliable delivery semantics. To avoid the ACK implosion problem, reliable multicast schemes either use a hierarchy of acknowledgement points or send redundant information.

FOR FURTHER STUDY

Deering [RFC 2236] specifies the standard for IP multicasting described in this chapter, which includes version 2 of IGMP. Waitzman, Partridge, and Deering [RFC 1075] describes DVMRP, Estrin et. al. [RFC 2362] describes PIM sparse mode, Ballardie [RFCs 2189 2201] describes CBT, and Moy [RFC 1585] describes MOSPF.

Eriksson [1994] explains the multicast backbone. Casner and Deering [July 1992] reports on the first multicast of an IETF meeting.

EXERCISES

17.1 The standard suggests using 23 bits of an IP multicast address to form a hardware multicast address. In such a scheme, how many IP multicast addresses map to a single hardware multicast address?

17.2 Argue that IP multicast addresses should use only 23 of the 28 possible bits. Hint: what are the practical limits on the number of groups to which a host can belong and the number of hosts on a single network?

17.3 IP must always check the destination addresses on incoming multicast datagrams and discard datagrams if the host is not in the specified multicast group. Explain how the host might receive a multicast destined for a group to which that host is not a member.

17.4 Multicast routers need to know whether a group has members on a given network. Is there any advantage to them knowing the exact set of hosts on a network that belong to a given multicast group?

17.5 Find three applications in your environment that can benefit from IP multicast.

17.6 The standard says that IP software must arrange to deliver a copy of any outgoing multicast datagram to application programs on the host that belong to the specified multicast group. Does this design make programming easier or more difficult? Explain.

17.7 When the underlying hardware does not support multicast, IP multicast uses hardware broadcast for delivery. How can doing so cause problems? Is there any advantage to using IP multicast over such networks?

17.8 DVMRP was derived from RIP. Read RFC 1075 on DVMRP and compare the two protocols. How much more complex is DVMRP than RIP?

17.9 IGMP does not include a strategy for acknowledgement or retransmission, even when used on networks that use best-effort delivery. What can happen if a query is lost? What can happen if a response is lost?

17.10 Explain why a multi-homed host may need to join a multicast group on one network, but not on another. (Hint: consider an audio teleconference.)

17.11 Estimate the size of the multicast forwarding table needed to handle multicast of audio from 100 radio stations, if each station has a total of ten million listeners at random locations around the world.

17.12 Argue that only two types of multicast are practical in the Internet: statically configured commercial services that multicast to large numbers of subscribers and dynamically configured services that include a few participants (e.g., family members in three households participating in a conference phone call).

17.13 Consider reliable multicast achieved through redundant transmission. If a given link has high probability of corruption, is it better to send redundant copies of a datagram or to send one copy that uses forward error-correcting codes? Explain.

17.14 The data-driven multicast routing paradigm works best on local networks that have low delay and excess capacity, while the demand-driven paradigm works best in a wide area environment that has limited capacity and higher delay. Does it make sense to devise a single protocol that combines the two schemes? Why or why not. (Hint: investigate MOSPF.)

17.15 Devise a quantitative measure that can be used to decide when PIM-SM should switch from a shared tree to a shortest path tree.

17.16 Read the protocol specification to find out the notion of "sparse" used in PIM-SM. Find an example of an internet in which the population of group members is sparse, but for which DVMRP is a better multicast routing protocol.

18

TCP/IP Over ATM Networks

18.1 Introduction

Previous chapters explain the fundamental parts of TCP/IP and show how the components operate over conventional LAN and WAN technologies. This chapter explores how TCP/IP, which was designed for connectionless networks, can be used over a connection-oriented technology†. We will see that TCP/IP is extremely flexible — a few of the address binding details must be modified for a connection-oriented environment, but most protocols remain unchanged.

The challenge arises when using TCP/IP over *Non-Broadcast Multiple-Access* (*NBMA*) networks (i.e., connection-oriented networks which allow multiple computers to attach, but do not support broadcast from one computer to all others). We will see that an NBMA environment requires modifications to IP protocols such as ARP that rely on broadcast.

To make the discussion concrete and relate it to available hardware, we will use *Asynchronous Transfer Mode* (*ATM*) in all examples. This chapter expands the brief description of ATM in Chapter 2, and covers additional details. The next sections describe the physical topology of an ATM network, the logical connectivity provided, ATM's connection paradigm, and the ATM adaptation protocol used to transfer data. Later sections discuss the relationship between ATM and TCP/IP. They explain ATM addressing, and show the relationship between a host's ATM address and its IP address. They also describe a modified form of the Address Resolution Protocol (ARP) used to resolve an IP address across a connection-oriented network, and a modified form of Inverse ARP that a server can use to obtain and manage addresses. Most important, we will see how IP datagrams travel across an ATM network without IP fragmentation.

†Some documents use the abbreviation *CL* for *connectionless* and *CO* for *connection-oriented*.

18.2 ATM Hardware

Like most connection-oriented technologies, ATM uses special-purpose electronic switches as the basic network building block. The switches in an ATM LAN usually provide connections for between 16 and 32 computers.† Although it is possible to use copper wiring between a host and an ATM switch, most installations use optical fiber to provide higher data rates. Figure 18.1 shows a diagram of an ATM switch with computers connected, and explains the connection.

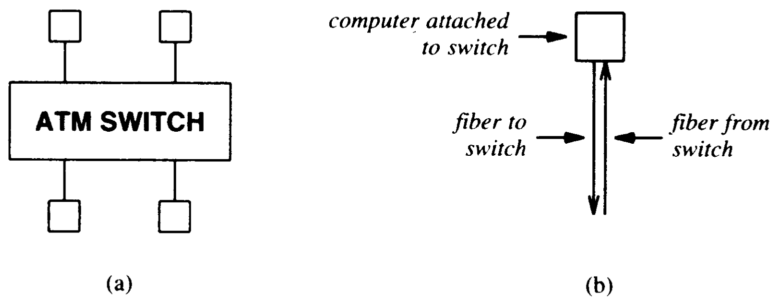


Figure 18.1 (a) The schematic diagram of a single ATM switch with four computers attached, and (b) the details of each connection. A pair of optical fibers carries data to and from the switch.

Physically, a host interface board plugs into a computer's bus. The interface hardware includes optical transmitters and receivers along with the circuitry needed to convert between electrical signals and the pulses of light that travel down the fiber to the switch. Because each fiber is used to carry light in only one direction, a connection that allows a computer to both send and receive data requires a pair of fibers.

18.3 Large ATM Networks

Although a single ATM switch has finite capacity, multiple switches can be interconnected to form a larger network. In particular, to connect computers at two sites to the same network, a switch can be installed at each site, and the two switches can then be connected. The connection between two switches differs slightly from the connection between a host computer and a switch. For example, interswitch connections usually operate at higher speeds, and use slightly modified protocols. Figure 18.2 illustrates the topology, and shows the conceptual difference between a *Network to Network Interface* (*NNI*) and a *User to Network Interface* (*UNI*).

†Switches used in larger networks provide more connections: the point is that the number of computers attached to a given switch is limited.

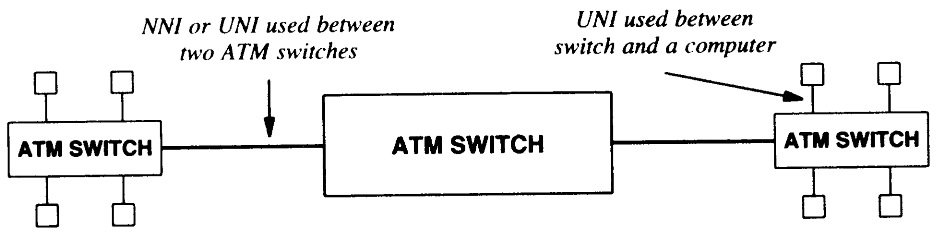

NNI or UNI used between two ATM switches
UNI used between switch and a computer
ATM SWITCH
ATM SWITCH
ATM SWITCH

Figure 18.2 Three ATM switches combined to form a large network. Although an NNI interface is designed for use between switches, UNI connections can be used between ATM switches in a private network.

18.4 The Logical View Of An ATM Network

The goal of ATM is an end-to-end communication system. To a computer attached to an ATM network, an entire fabric of ATM switches appears to be a homogeneous network. Like the voice telephone system, a bridged Ethernet, or an IP internet, ATM hides the details of physical hardware and gives the appearance of a single, physical network with many computers attached. For example, Figure 18.3 illustrates how the ATM switching system in Figure 18.2 appears logically to the eight computers that are attached to it.

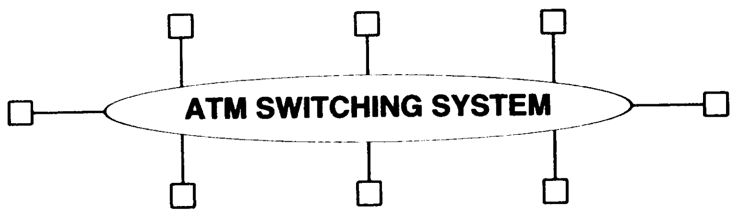


Figure 18.3 The logical view of the ATM switches in Figure 18.2. ATM gives the appearance of a uniform network; any computer can communicate with any other computer.

Thus, ATM provides the same general abstraction across homogeneous ATM hardware that TCP/IP provides for heterogeneous systems:

Despite a physical architecture that permits a switching fabric to contain multiple switches, ATM hardware provides attached computers with the appearance of a single, physical network. Any computer on an ATM network can communicate directly with any other; the computers remain unaware of the physical network structure.

18.5 The Two ATM Connection Paradigms

ATM provides a connection-oriented interface to attached hosts. That is, before it can send data to a remote destination, a host must establish a *connection*, an abstraction analogous to a telephone call. Although there is only one type of underlying connection, ATM offers two ways to create a connection. The first is known as a *Permanent Virtual Circuit*† (*PVC*), and the second is known as a *Switched Virtual Circuit*† (*SVC*).

18.5.1 Permanent Virtual Circuits

In telephone jargon, a PVC is said to be a *provisioned service*. Provisioning simply means that a person is required to enter the necessary configuration manually into each switch along the path from the source to the destination (e.g., by typing into the console on each switch). Although the terms *PVC* and *provisioned service* may sound esoteric, the concept is not; even the most basic connection-oriented hardware supports PVCs.

On one hand, manual configuration has an obvious disadvantage: it cannot be changed rapidly or easily. Consequently, PVCs are only used for connections that stay in place for relatively long periods of time (weeks or years). On the other hand, manual configuration has advantages: a PVC does not require all switches to agree on a standard signaling mechanism. Thus, switches from two or more vendors may be able to interoperate when using PVCs, even if they cannot when using SVCs. Second, PVCs are often required for network management, maintenance, and debugging operations.

18.5.2 Switched Virtual Circuits

Unlike a PVC, an SVC is created automatically by software, and terminated when no longer needed. Software on a host initiates SVC creation; it passes a request to the local switch. The request includes the compiete address of a remote host computer with which an SVC is needed and parameters that specify the quality of service required (e.g., the bandwidth and delay). The host then waits for the ATM network to create a circuit and respond. The ATM *signaling*‡ system establishes a path from the originating host across the ATM network (possibly through multiple switches) to the remote host computer.

During signaling, each ATM switch along the path and the remote computer must agree to establish the virtual circuit. When it agrees, a switch records information about the circuit, reserves the necessary resources, and sends the request to the next switch along the path. Once all the switches and the remote computer respond, signaling completes, and the switches at each end of the connection report to the hosts that the virtual circuit is in place.

Like all abstractions, connections must be identified. The UNI interface uses a 24-bit integer to identify each virtual circuit. When administrators create PVCs, they assign an identifier to each. When software on a host creates a new SVC, the local ATM switch assigns an identifier and informs the host. Unlike connectionless technolo-

†Although the ATM standard uses the term *virtual channel*, we will follow common practice and call it a *virtual circuit*.

‡The term *signaling* derives from telephone jargon.

gies, a connection-oriented system does not require each packet to carry either a source or destination address. Instead, a host places a circuit identifier in each outgoing packet, and the switch places a circuit identifier in each packet it delivers.

18.6 Paths, Circuits, And Identifiers

We said that a connection-oriented technology assigns a unique integer identifier to each circuit, and that a host uses the identifier when performing I/O operations or when closing the circuit. However, connection-oriented systems do not assign each circuit a globally unique identifier. Instead, the identifier is analogous to an I/O descriptor that is assigned to a program by the operating system. Like an I/O descriptor, a circuit identifier is a shorthand that a program uses in place of the full information that was used to create the circuit. Also like an I/O descriptor, a circuit identifier only remains valid while the circuit is open. Furthermore, a circuit identifier is meaningful only across a single hop — the circuit identifiers obtained by hosts at the two ends of a given virtual circuit usually differ. For example, the sender may be using identifier *17* while the receiver uses identifier *49*; each switch along the path translates the circuit identifier in a packet as the packet flows from one host to the other.

Technically, a circuit identifier used with the UNI interface consists of a 24-bit integer divided into two fields†. Figure 18.4 shows how ATM partitions the 24 bits into an 8-bit *virtual path identifier* (*VPI*) and a 16-bit *virtual circuit identifier* (*VCI*). Often, the entire identifier is referred to as a *VPI/VCI pair*.

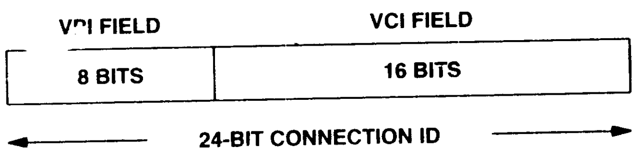


Figure 18.4 The 24-bit connection identifier used with UNI. The identifier is divided into virtual path and virtual circuit parts.

The motivation for dividing a connection identifier into VPI and VCI fields is similar to the reasons for dividing an IP address into network and host fields. If a set of virtual circuits follows the same path, an administrator can arrange for all circuits in the set to use the same VPI. ATM hardware can then use the VPI to route traffic efficiently. Commercial carriers can also use the VPI for accounting — a carrier can charge a customer for a virtual path, and then allow the customer to decide how to multiplex multiple virtual circuits over the path.

†The circuit identifier used with NNI has a slightly different format and a different length.

18.7 ATM Cell Transport

At the lowest level, an ATM network uses fixed-size frames called *cells* to carry data. ATM requires all cells to be the same size because doing so makes it possible to build faster switching hardware and to handle voice as well as data. Each ATM cell is 53 octets long, and consists of a 5-octet header followed by 48 octets of payload (i.e. data). Figure 18.5 shows the format of a cell header.

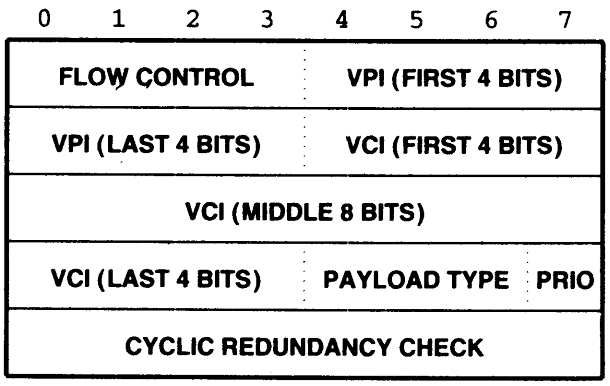


Figure 18.5 The format of the five-octet UNI cell header used between a host and a switch. The diagram shows one octet per line; forty-eight octets of data follow the header.

18.8 ATM Adaptation Layers

Although ATM switches small cells at the lowest level, application programs that transfer data over ATM do not read or write cells. Instead, a computer interacts with ATM through an *ATM Adaptation Layer*, which is part of the ATM standard. The adaptation layer performs several functions, including detection and correction of errors such as lost or corrupted cells. Usually, firmware that implements an ATM adaptation layer is located on a host interface along with hardware and firmware that provide cell transmission and reception. Figure 18.6 illustrates the organization of a typical ATM interface, and shows how data passes from the computer's operating system through the interface board and into an ATM network.

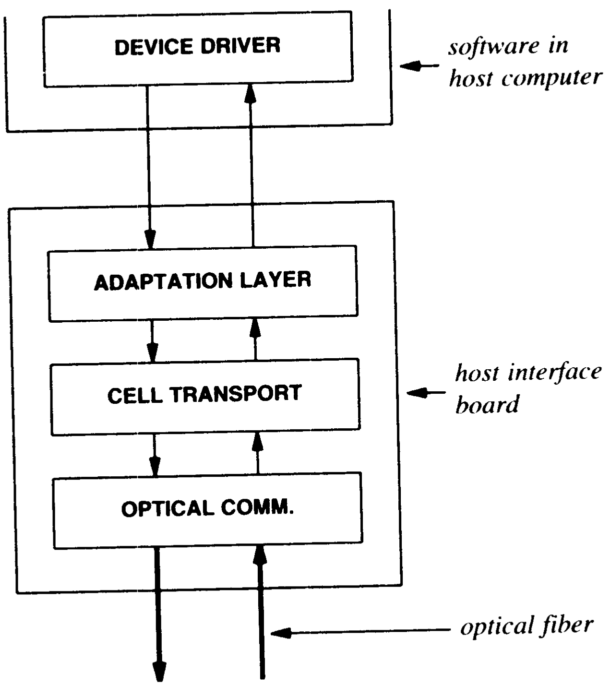


Figure 18.6 The conceptual organization of ATM interface hardware and the flow of data through it. Software on a host interacts with an adaptation layer protocol to send and receive data; the adaptation layer converts to and from cells.

When establishing a connection, a host must specify which adaptation layer protocol to use. Both ends of the connection must agree on the choice, and the adaptation layer cannot be changed once the connection has been established. To summarize:

Although ATM hardware uses small, fixed-size cells to transport data, a higher layer protocol called an ATM Adaptation Layer provides data transfer services for computers that use ATM. When a virtual circuit is created, both ends of the circuit must agree on which adaptation layer protocol will be used.

18.9 ATM Adaptation Layer 5

Computers use *ATM Adaptation Layer 5* (*AAL5*) to send data across an ATM network. Interestingly, although ATM uses small fixed-size cells at the lowest level, AAL5 presents an interface that accepts and delivers large, variable-length packets. Thus, the interface computers use to send data makes ATM appear much like a connectionless technology. In particular, AAL5 allows each packet to contain between 1 and 65,535 octets of data. Figure 18.7 illustrates the packet format that AAL5 uses.

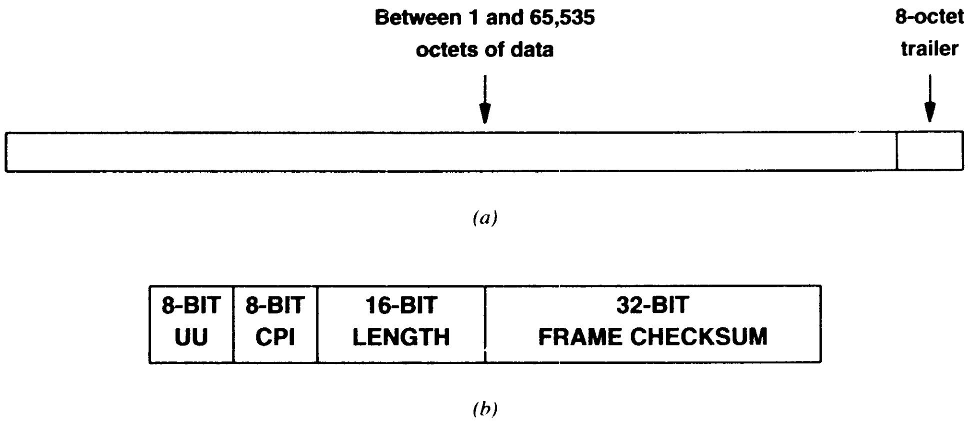


Figure 18.7 (a) The basic packet format that AAL5 accepts and delivers, and (b) the fields in the 8-octet trailer that follows the data.

Unlike most network frames that place control information in a header, AAL5 places control information in an 8-octet trailer at the end of the packet. The AAL5 trailer contains a 16-bit length field, a 32-bit cyclic redundancy check (*CRC*) used as a frame checksum, and two 8-bit fields labeled *UU* and *CPI* that are currently unused†.

Each AAL5 packet must be divided into cells for transport across an ATM network, and then must be recombined to form a packet before being delivered to the receiving host. If the packet, including the 8-octet trailer, is an exact multiple of 48 octets, the division will produce completely full cells. If the packet is not an exact multiple of 48 octets, the final cell will not be full. To accommodate arbitrary length packets, AAL5 allows the final cell to contain between 0 and 40 octets of data, followed by zero padding, followed by the 8-octet trailer. In other words, AAL5 places the trailer in the last 8 octets of the final cell, where it can be found and extracted without knowing the length of the packet.

†Field *UU* can contain any value; field *CPI* must be set to zero.

18.10 AAL5 Convergence, Segmentation, And Reassembly

When an application sends data over an ATM connection using AAL5, the host delivers a block of data to the AAL5 interface. AAL5 generates a trailer, divides the information into 48-octet pieces, and transfers each piece across the ATM network in a single cell. On the receiving end of the connection, AAL5 reassembles incoming cells into a packet, checks the CRC to ensure that all pieces arrived correctly, and passes the resulting block of data to the host software. The process of dividing a block of data into cells and regrouping them is known as *ATM segmentation and reassembly*† (*SAR*).

By separating the functions of segmentation and reassembly from cell transport, AAL5 follows the layering principle. The ATM cell transfer layer is classified as *machine-to-machine* because the layering principle applies from one machine to the next (e.g., between a host and a switch or between two switches). The AAL5 layer is classified as *end-to-end* because the layering principle applies from the source to the destination — AAL5 presents the receiving software with data in exactly the same size blocks as the application passed to AAL5 on the sending end.

How does AAL5 on the receiving side know how many cells comprise a packet? The sending AAL5 uses the low-order bit of the *PAYLOAD TYPE* field of the ATM cell header to mark the final cell in a packet. One can think of it as an *end-of-packet bit.* Thus, the receiving AAL5 collects incoming cells until it finds one with the end-of-packet bit set. ATM standards use the term *convergence* to describe mechanisms that recognize the end of a packet. Although AAL5 uses a single bit in the cell header for convergence, other ATM adaptation layer protocols are free to use other convergence mechanisms.

To summarize:

> *A computer uses ATM Adaptation Layer 5 to transfer a large block of data over an ATM virtual circuit. On the sending host, AAL5 generates a trailer, segments the block of data into cells, and transmits each cell over the virtual circuit. On the receiving host, AAL5 reassembles the cells to reproduce the original block of data, strips off the trailer, and delivers the block of data to the receiving host software. A single bit in the cell header marks the final cell of a given data block.*

18.11 Datagram Encapsulation And IP MTU Size

We said that IP uses AAL5 to transfer datagrams across an ATM network. Before data can be sent, a virtual circuit (PVC or SVC) must be in place to the destination computer and both ends must agree to use AAL5 on the circuit. To transfer a datagram, the sender passes it to AAL5 along with the VPI/VCI identifying the circuit. AAL5 generates a trailer, divides the datagram into cells, and transfers the cells across the net-

†Use of the term *reassembly* suggests the strong similarity between AAL5 segmentation and IP fragmentation: both mechanisms divide a large block of data into smaller units for transfer.

work. At the receiving end, AAL5 reassembles the cells, checks the CRC to verify that no bits were lost or corrupted, extracts the datagram, and passes it to IP.

In reality, AAL5 uses a 16-bit length field, making it possible to send 64K octets in a single packet. Despite the capabilities of AAL5, TCP/IP restricts the size of datagrams that can be sent over ATM. The standards impose a default of 9180 octets† per datagram. As with any network interface, when an outgoing datagram is larger than the network MTU, IP fragments the datagram, and passes each fragment to AAL5. Thus, AAL5 accepts, transfers, and delivers datagrams of 9180 octets or less. To summarize:

> *When TCP/IP sends data across an ATM network, it transfers an entire datagram using ATM Adaptation Layer 5. Although AAL5 can accept and transfer packets that contain up to 64K octets, the TCP/IP standards specify a default MTU of 9180 octets. IP must fragment any datagram larger than 9180 octets before passing it to AAL5.*

18.12 Packet Type And Multiplexing

Observant readers will have noticed that the AAL5 trailer does not include a *type* field. Thus, an AAL5 frame is not self-identifying. As a result, the simplest form of encapsulation described above does not suffice if the two ends want to send more than one type of data across a single VC (e.g., packets other than IP). Two possibilities exist:

- The two computers at the ends of a virtual circuit agree *a priori* that the circuit will be used for a specific protocol (e.g., the circuit will only be used to send IP datagrams).
- The two computers at the ends of a virtual circuit agree *a priori* that some octets of the data area will be reserved for use as a type field.

The former scheme, in which the computers agree on the high-level protocol for a given circuit, has the advantage of not requiring additional information in a packet. For example, if the computers agree to transfer IP, a sender can pass each datagram directly to AAL5 to transfer; nothing needs to be sent besides the datagram and the AAL5 trailer. The chief disadvantage of such a scheme lies in duplication of virtual circuits: a computer must create a separate virtual circuit for each high-level protocol. Because most carriers charge for each virtual circuit, customers try to avoid using multiple circuits because it adds unnecessary cost.

The latter scheme, in which two computers use a single virtual circuit for multiple protocols, has the advantage of allowing all traffic to travel over the same circuit, but the disadvantage of requiring each packet to contain octets that identify the protocol type. The scheme also has the disadvantage that packets from all protocols travel with the same delay and priority.

†The size 9180 was chosen to make ATM compatible with an older technology called *Switched Multimegabit Data Service (SMDS)*; a value other than 9180 can be used if both ends agree.

The TCP/IP standards specify that computers can choose between the two methods of using AAL5. Both the sender and receiver must agree on how the circuit will be used; the agreement may involve manual configuration. Furthermore, the standards suggest that when computers choose to include type information in the packet, they should use a standard IEEE 802.2 *Logical Link Control* (*LLC*) header followed by a *SubNetwork Attachment Point* (*SNAP*) header. Figure 18.8 illustrates the LLC/SNAP information prefixed to a datagram before it is sent over an ATM virtual circuit.

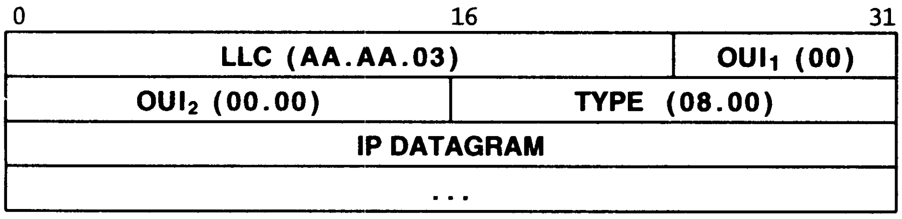


Figure 18.8 The packet format used to send a datagram over AAL5 when multiplexing multiple protocols on a single virtual circuit. The 8-octet LLC/SNAP header identifies the contents as an IP datagram.

As the figure shows, the LLC field consists of three octets that contain the hexadecimal values *AA.AA.03*†. The SNAP header consists of five octets: three that contain an *Organizationally Unique Identifier* (*OUI*) and two for a type‡. Field *OUI* identifies an organization that administers values in the *TYPE* field, and the *TYPE* field identifies the packet type. For an IP datagram, the *OUI* field contains *00.00.00* to identify the organization responsible for Ethernet standards, and the *TYPE* field contains *08.00*, the value used when encapsulating IP in an Ethernet frame. Software on the sending host must prefix the LLC/SNAP header to each packet before sending it to AAL5, and software on the receiving host must examine the header to determine how to handle the packet.

18.13 IP Address Binding In An ATM Network

We have seen that encapsulating a datagram for transmission across an ATM network is straightforward. By contrast, IP address binding in a Non-Broadcast Multiple-Access (*NBMA*) environment can be difficult. Like other network technologies, ATM assigns each attached computer a physical address that must be used when establishing a virtual circuit. On one hand, because an ATM physical address is larger than an IP address, an ATM physical address cannot be encoded within an IP address. Thus, IP cannot use static address binding for ATM networks. On the other hand, ATM

†The notation represents each octet as a hexadecimal value separated by decimal points.

‡To avoid unnecessary fragmentation, the eight octets of an LLC/SNAP header are ignored in the MTU computation (i.e., the effective MTU of an ATM connection that uses an LLC/SNAP header is 9188).

hardware does not support broadcast. Thus, IP cannot use conventional ARP to bind addresses on ATM networks.

ATM permanent virtual circuits further complicate address binding. Because a manager configures each permanent virtual circuit manually, a host only knows the circuit's VPI/VCI pair. Software on the host may not know the IP address nor the ATM hardware address of the remote endpoint. Thus, an IP address binding mechanism must provide for the identification of a remote computer connected over a PVC as well as the dynamic creation of SVCs to known destinations.

Switched connection-oriented technologies further complicate address binding because they require two levels of binding. First, when creating a virtual circuit over which datagrams will be sent, the IP address of the destination must be mapped to an ATM endpoint address. The endpoint address is used to create a virtual circuit. Second, when sending a datagram to a remote computer over an existing virtual circuit, the destination's IP address must be mapped to the VPI/VCI pair for the circuit. The second binding is used each time a datagram is sent over an ATM network; the first binding is necessary only when a host creates an SVC.

18.14 Logical IP Subnet Concept

Although no protocol has been proposed to solve the general case of address binding for NBMA networks like ATM, a protocol has been devised for a restricted form. The restricted form arises when a group of computers uses an ATM network in place of a single (usually local) physical network. The group forms a *Logical IP Subnet* (*LIS*). Multiple logical IP subnets can be defined among a set of computers that all attach to the same ATM hardware network. For example, Figure 18.9 illustrates eight computers attached to an ATM network divided into two LIS.

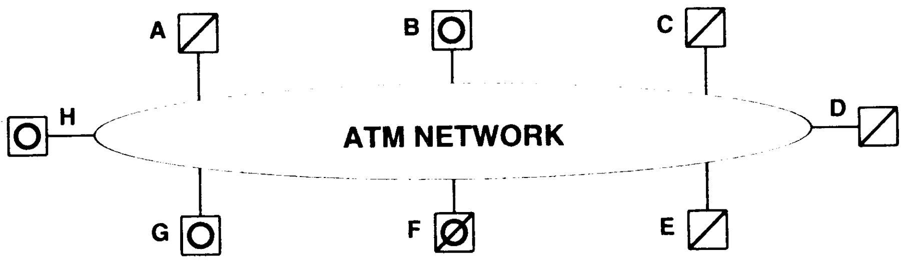


Figure 18.9 Eight computers attached to an ATM network participating in two Logical IP Subnets. Computers marked with a slash participate in one LIS, while computers marked with a circle participate in the other LIS.

As the figure shows, all computers attach to the same physical ATM network. Computers *A*, *C*, *D*, *E*, and *F* participate in one LIS, while computers *B*, *F*, *G*, and *H* participate in another. Each logical IP subnet functions like a separate LAN. The computers participating in an LIS establish virtual circuits among themselves to exchange datagrams†. Because each LIS forms a conceptually separate network, IP applies the standard rules for a physical network to each LIS. For example, all computers in an LIS share a single IP network prefix, and that prefix differs from the prefixes used by other logical subnets. Furthermore, although the computers in an LIS can choose a non-standard MTU, all computers must use the same MTU on all virtual circuits that comprise the LIS. Finally, despite the ATM hardware that provides potential connectivity, a host in one LIS is forbidden from communicating directly with a host in another LIS. Instead, all communication between logical subnets must proceed through a router just as communication between two physical Ethernets proceeds through a router. In Figure 18.9, for example, machine *F* represents an IP router because it participates in both logical subnets.

To summarize:

> *TCP/IP allows a subset of computers attached to an ATM network to operate like an independent LAN. Such a group is called a* Logical IP Subnet *(LIS); computers in an LIS share a single IP network prefix. A computer in an LIS can communicate directly with any other computer in the same LIS, but is required to use a router when communicating with a computer in another LIS.*

18.15 Connection Management

Hosts must manage ATM virtual circuits carefully because creating a circuit takes time and, for commercial ATM services, can incur additional economic cost. Thus, the simplistic approach of creating a virtual circuit, sending one datagram, and then closing the circuit is too expensive. Instead, a host must maintain a record of open circuits so they can be reused.

Circuit management occurs in the network interface software below IP. When a host needs to send a datagram, it uses conventional IP routing to find the appropriate next-hop address, *N*‡, and passes it along with the datagram to the network interface. The network interface examines its table of open virtual circuits. If an open circuit exists to *N*, the host uses AAL5 to send the datagram. Otherwise, before the host can send the datagram, it must locate a computer with IP address *N*, create a circuit, and add the circuit to its table.

The concept of logical IP subnets constrains IP routing. In a properly configured routing table, the next-hop address for each destination must be a computer within the same logical subnet as the sender. To understand the constraint, remember that each LIS is designed to operate like a single LAN. The same constraint holds for a host at-

†The standard specifies the use of LLC/SNAP encapsulation within an LIS.

‡As usual, a next-hop address is an IP address.

tached to a LAN, namely, each next-hop address in the routing table must be a router attached to the LAN.

One of the reasons for dividing computers into logical subnets arises from hardware and software constraints. A host cannot maintain an arbitrarily large number of open virtual circuits at the same time because each circuit requires resources in the ATM hardware and in the operating system. Dividing computers into logical subnets limits the maximum number of simultaneously open circuits to the number of computers in the LIS.

18.16 Address Binding Within An LIS

When a host creates a virtual circuit to a computer in its LIS, the host must specify an ATM hardware address for the destination. How can a host map a next-hop address into an appropriate ATM hardware address? The host cannot broadcast a request to all computers in the LIS because ATM does not offer hardware broadcast. Instead, it contacts a server to obtain the mapping. Communication between the host and server uses *ATMARP*, a variant of the ARP protocol described in Chapter 5.

As with conventional ARP, a sender forms a request that includes the sender's IP and ATM hardware addresses as well as the IP address of a target for which the ATM hardware address is needed. The sender then transmits the request to the *ATMARP server* for the logical subnet. If the server knows the ATM hardware address, it sends an *ATMARP reply*. Otherwise, the server sends a *negative ATMARP reply*.

18.17 ATMARP Packet Format

Figure 18.10 illustrates the format of an ATMARP packet. As the figure shows, ATMARP modifies the ARP packet format slightly. The major change involves additional address length fields to accommodate ATM addresses. To appreciate the changes, one must understand that multiple address forms have been proposed for ATM, and that no single form appears to be the emerging standard. Telephone companies that offer public ATM networks use an 8-octet format where each address is an ISDN telephone number defined by ITU standard document *E.164*. By contrast, the ATM Forum† allows each computer attached to a private ATM network to be assigned a 20-octet *Network Service Access Point* (*NSAP*) address. Thus, a two-level hierarchical address may be needed that specifies an E.164 address for a remote site and an NSAP address of a host on a local switch at the site.

To accommodate multiple address formats and a two-level hierarchy, an ATMARP packet contains two length fields for each ATM address as well as a length field for each protocol address. As Figure 18.10 shows, an ATMARP packet begins with fixed-size fields that specify address lengths. The first two fields follow the same format as conventional ARP. The field labeled *HARDWARE TYPE* contains the hexadecimal

†The ATM Forum is a consortium of industrial members that recommends standards for private ATM networks.

value *0x0013* for ATM, and the field labeled *PROTOCOL TYPE* contains the hexadecimal value *0x0800* for IP.

Because the address format of the sender and target can differ, each ATM address requires a length field. Field *SEND HLEN* specifies the length of the sender's ATM address, and field *SEND HLEN2* specifies the length of the sender's ATM subaddress. Fields *TAR HLEN* and *TAR HLEN2* specify the lengths of the target's ATM address and subaddress. Finally, fields *SEND PLEN* and *TAR PLEN* specify the lengths of the sender's and target's protocol addresses.

Following the length fields in the header, an ATMARP packet contains six addresses. The first three address fields contain the sender's ATM address, ATM subaddress, and protocol address. The last three fields contain the target's ATM address, ATM subaddress, and protocol address. In the example in Figure 18.10, both the sender and target subaddress length fields contain zero, and the packet does not contain octets for subaddresses.

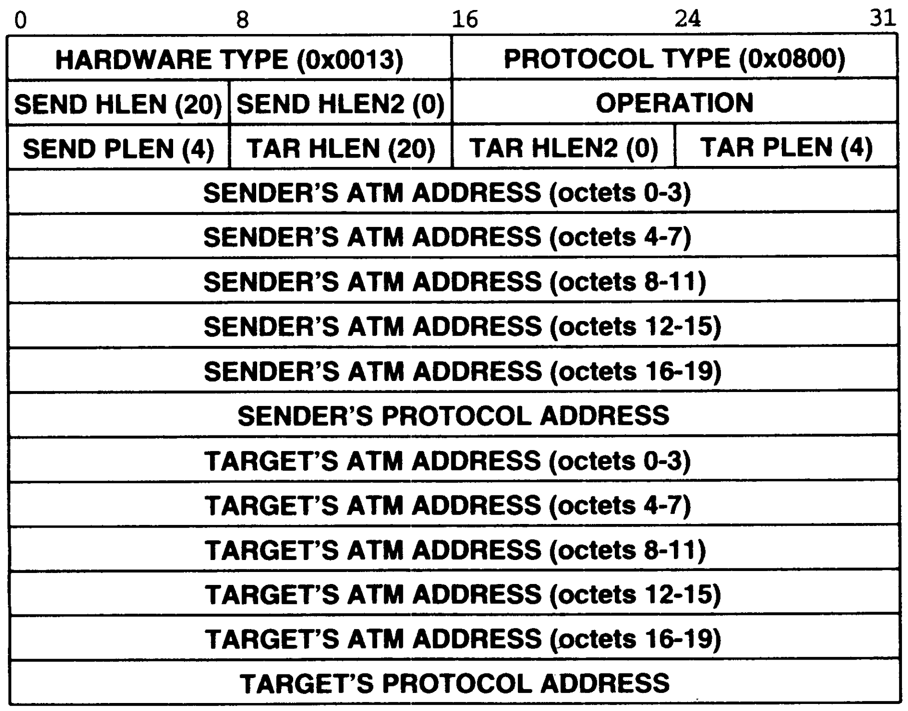

0
8
16
24
31
HARDWARE TYPE (0x0013)
PROTOCOL TYPE (0x0800)
SEND HLEN (20)
SEND HLEN2 (0)
OPERATION
SEND PLEN (4)
TAR HLEN (20)
TAR HLEN2 (0)
TAR PLEN (4)
SENDER'S ATM ADDRESS (octets 0-3)
SENDER'S ATM ADDRESS (octets 4-7)
SENDER'S ATM ADDRESS (octets 8-11)
SENDER'S ATM ADDRESS (octets 12-15)
SENDER'S ATM ADDRESS (octets 16-19)
SENDER'S PROTOCOL ADDRESS
TARGET'S ATM ADDRESS (octets 0-3)
TARGET'S ATM ADDRESS (octets 4-7)
TARGET'S ATM ADDRESS (octets 8-11)
TARGET'S ATM ADDRESS (octets 12-15)
TARGET'S ATM ADDRESS (octets 16-19)
TARGET'S PROTOCOL ADDRESS

Figure 18.10 The format of an ATMARP packet when used with 20-octet ATM addresses such as those recommended by the ATM Forum.

18.17.1 Format Of ATM Address Length Fields

Because ATMARP is designed for use with either E.164 addresses or 20-octet NSAP addresses, fields that contain an ATM address length include a bit that specifies the address format. Figure 18.11 illustrates how ATMARP encodes the address type and length in an 8-bit field.

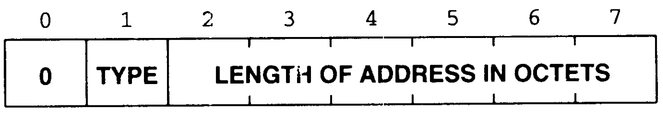


Figure 18.11 The encoding of ATM address type and length in an 8-bit field. Bit *1* distinguishes the two types of ATM addresses.

A single bit encodes the type of an ATM address because only two forms are available. If bit *1* contains zero, the address is in the NSAP format recommended by the ATM Forum. If bit *1* contains one, the address is in the E.164 format recommended by the ITU. Because each ATM address length field in an ATMARP packet has the form shown in Figure 18.11, a single packet can contain multiple types of ATM addresses.

18.17.2 Operation Codes Used With The ATMARP Protocol

The packet format shown in Figure 18.10 is used to request an address binding, reply to a request, or request an inverse address binding. When a computer sends an ATMARP packet, it must set the *OPERATION* field to specify the type of binding. The table in Figure 18.12 shows the values that can be used in the *OPERATION* field, and gives the meaning of each. The remainder of this section explains how the protocol works.

Code	Meaning
1	ATMARP Request
2	ATMARP Reply
8	Inverse ATMARP Request
9	Inverse ATMARP Reply
10	ATMARP Negative Ack

Figure 18.12 The values that can appear in the *OPERATION* field of an ATMARP packet and their meanings. When possible, values have been chosen to agree with the operation codes used in conventional ARP.

18.18 Using ATMARP Packets To Determine An Address

Performing address binding for connection-oriented hardware is slightly more complex than for connectionless hardware. Because ATM hardware supports two types of virtual circuits, two cases arise. First, we will consider the case of permanent virtual circuits. Second, we will consider the case of switched virtual circuits.

18.18.1 Permanent Virtual Circuits

To understand the problems PVCs introduce, recall how ATM hardware operates. A network administrator must configure each PVC; hosts themselves do not participate in PVC setup. In particular, a host begins operation with PVCs in place, and does not receive any information from the hardware about the address of the remote endpoint. Thus, unless address information has been configured into the hosts (e.g., stored on disk), the host does not know the IP address or ATM address of the computer to which a PVC connects.

The *Inverse ATMARP* protocol (*InATMARP*) solves the problem of finding addresses when using PVCs. To use the protocol, a computer must know each of the permanent virtual circuits that have been configured. To determine the IP and ATM addresses of the remote endpoint, a computer sends an Inverse ATMARP request packet with the *OPERATION* field set to *8*. Whenever such a request arrives over a PVC, the receiver generates an Inverse ATMARP reply with the *OPERATION* field set to *9*. Both the request and the reply contain the sender's IP address and ATM address. Thus, a computer at each end of the connection learns the binding for the computer at the other end. In summary,

> *Two computers that communicate over a permanent virtual circuit use Inverse ATMARP to discover each others' IP and ATM addresses. One computer sends an Inverse ATMARP request, to which the other sends a reply.*

18.18.2 Switched Virtual Circuits

Within an LIS, computers create switched virtual circuits on demand. When computer *A* needs to send a datagram to computer *B* and no circuit currently exists to *B*, *A* uses ATM signaling to create the necessary circuit. Thus, *A* begins with *B*'s IP address, which must be mapped to an equivalent ATM address. We said that each LIS has an ATMARP server, and all computers in an LIS must be configured so they know how to reach the server (e.g., a computer can have a PVC to the server or can have the server's ATM address stored on disk). A server does not form connections to other computers; the server merely waits for computers in the LIS to contact it. To map address *B* to an ATM address, computer *A* must have a virtual circuit open to the ATMARP server for the LIS. Computer *A* forms an ATMARP request packet and sends it over the connection to the server. The *OPERATION* field in the packet contains *1*, and the target's protocol address field contains *B*'s IP address.

An ATMARP server maintains a database of mappings from IP addresses to ATM addresses. If the server knows *B*'s ATM address, the ATMARP protocol operates similar to proxy ARP. The server forms an ATMARP reply by setting the *OPERATION* code to *2* and filling in the ATM address that corresponds to the target IP address. As in conventional ARP, the server exchanges sender and target entries before returning the reply to the computer that sent the request.

If the server does not know the ATM address that corresponds to the target IP address in a request, ATMARP's behavior differs from conventional ARP. Instead of ignoring the request, the server returns a negative acknowledgement (an ATMARP packet with an *OPERATION* field of *10*). A negative acknowledgement distinguishes between addresses for which a server does not have a binding and a malfunctioning server. Thus, when a host sends a request to an ATMARP server, it determines one of three outcomes unambiguously. The host can learn the ATM address of the target, that the target is not currently available in the LIS, or that the server is not currently responding.

18.19 Obtaining Entries For A Server Database

An ATMARP server builds and maintains its database of bindings automatically. To do so, it uses Inverse ATMARP. Whenever a host or router first opens a virtual circuit to an ATMARP server, the server immediately sends an Inverse ATMARP request packet†. The host or router must answer by sending an Inverse ATMARP reply packet. When it receives an Inverse ATMARP reply, the server extracts the sender's IP and ATM addresses, and stores the binding in its database. Thus, each computer in an LIS must establish a connection to the ATMARP server, even if the computer does not intend to look up bindings.

> *Each host or router in an LIS must register its IP address and corresponding ATM address with the ATMARP server for the LIS. Registration occurs automatically whenever a computer establishes a virtual circuit to an ATMARP server because the server sends an Inverse ATMARP to which the computer must respond.*

18.20 Timing Out ATMARP Information In A Server

Like the bindings in a conventional ARP cache, bindings obtained via ATMARP must be timed out and removed. How long should an entry persist in a server? Once a computer registers its binding with an ATMARP server, the server keeps the entry for a minimum of 20 minutes. After 20 minutes, the server examines the entry. If no circuit exists to the computer that sent the entry, the server deletes the entry‡. If the computer that sent the entry has maintained an open virtual circuit, the server attempts to revali-

†The circuit must use AAL5 with LLC/SNAP type identification.

‡A server does not automatically delete an entry when a circuit is closed; it waits for the timeout period.

date the entry. The server sends an Inverse ATMARP request and awaits a response. If the response verifies information in the entry, the server resets the timer and waits another 20 minutes. If the Inverse ATMARP response does not match the information in the entry, the server closes the circuit and deletes the entry.

To help reduce traffic, the ATMARP standard permits an optimization. It allows a host to use a single virtual circuit for all communication with an ATMARP server. When the host sends an ATMARP request, the request contains the host's binding in the *SENDER*'s field. The server can extract the binding and use it to revalidate its stored information. Thus, if a host sends more than one ATMARP request every 20 minutes, the server will not need to send the host an Inverse ATMARP request.

18.21 Timing Out ATMARP Information In A Host Or Router

A host or router must also use timers to invalidate information obtained from an ATMARP server. In particular, the standard specifies that a computer can keep a binding obtained from the ATMARP server for at most 15 minutes. When 15 minutes expire, the entry must be removed or revalidated. If an address binding expires and the host does not have an open virtual circuit to the destination, the host removes the entry from its ARP cache. If a host has an open virtual circuit to the destination, the host attempts to revalidate the address binding. Expiration of an address binding can delay traffic because:

A host or router must stop sending data to any destination for which the address binding has expired until the binding can be revalidated.

The method a host uses to revalidate a binding depends on the type of virtual circuit being used. If the host can reach the destination over a PVC, the host sends an Inverse ATMARP request on the circuit and awaits a reply. If the host has an SVC open to the destination, the host sends an ATMARP request to the ATMARP server.

18.22 IP Switching Technologies

So far, we have described ATM as a connection-oriented network technology that IP uses to transfer datagrams. However, engineers have also investigated a more fundamental union of the two technologies. They began with the question: ``can switching hardware be exploited to forward IP traffic at higher speeds?'' The assumption underlying the effort is that hardware will be able to switch more packets per second than to route them. If the assumption is correct, the question makes sense because router vendors are constantly trying to find ways to increase router performance and scale.

Ipsilon Corporation was one of the first companies to produce products that combined IP and hardware switches; they used ATM, called their technology *IP switching*, and called the devices they produced *IP switches*. Since Ipsilon, other companies have produced a series of designs and names, including *tag switching*, *layer 3 switching*, and *label switching*. Several of the ideas have been folded into a standard endorsed by the IETF that is known as *Multi-Protocol Label Switching* (*MPLS*)†. Contributors to the open standard hope that it will allow products from multiple vendors to interoperate.

18.23 Switch Operation

How do IP switching technologies work? There are two general answers. Early technologies all assumed the presence of a conventional NBMA network (usually ATM). The goal was to optimize IP routing to send datagrams across the ATM fabric instead of other networks whenever possible. In addition to proposing ways to optimize routes, later efforts also proposed modifying the switching hardware to optimize it for IP traffic. In particular, two optimizations have been proposed. First, if switching hardware can be redesigned to either use large cells or to allow variable-length frames, header overhead will be reduced‡. Second, if hardware can be built to parse IP headers and extract needed fields, an incoming datagram can be forwarded faster.

Forwarding is at the heart of all label switching. There are three aspects. First, at the IP layer, a forwarding device must function as a conventional IP router to transfer datagrams between a local network and the switched fabric. Thus, the device must learn about remote destinations, and must map an IP destination address into a next-hop address. Second, at the network interface layer, a forwarding device must be able to create and manage connections through the switched fabric (i.e., by mapping IP addresses to underlying hardware addresses and creating SVCs as needed). Third, a forwarding device must optimize paths through the switched fabric.

18.24 Optimized IP Forwarding

Optimized forwarding involves high-speed classification and *shortcut paths*. To understand shortcut paths, imagine three switches, S_1, S_2, and S_3, and suppose that to reach a given destination the IP routing table in S_1 specifies forwarding to S_2, which forwards to S_3, which delivers to the destination. Further suppose that all three devices are connected to the same fabric. If S_1 observes that many datagrams are being sent to the destination, it can optimize routing by bypassing S_2 and setting up a shortcut path (i.e., a virtual circuit) directly to S_3. Of course, many details need to be handled. For example, although our example involves only three devices, a real network may have many. After it learns the path a datagram will travel to its destination, S_1 must find the last hop along the path that is reachable through the switched network, translate the IP address of that hop to an underlying hardware address, and form a connection. Recognizing whether a given hop on the path connects to the same switching fabric and translating addresses are not easy; complex protocols are needed to pass the necessary information. To give IP the illusion that datagrams are following the routes specified by IP, either S_1 or S_3 must agree to account for the bypassed router when decrementing the TTL field in

†Despite having "multi-protocol" in the name, MPLS is focused almost exclusively on finding ways to put IP over an NBMA switched hardware platform.

‡In the industry, ATM header overhead is known as the *cell tax*.

the datagram header. Furthermore, S_1 must continue to receive routing updates from S_2 so it can revert to the old path in case routes change.

18.25 Classification, Flows, And Higher Layer Switching

A *classification scheme* examines each incoming datagram and chooses a connection over which the datagram should travel. Building a classification scheme in hardware further enhances the technology by allowing a switch to make the selection at high speed. Most of the proposed classification schemes use a two-level hierarchy. First, the switch classifies a datagram into one of many possible *flows*, and then the flow is mapped onto a given connection. One can think of the mapping mathematically as a pair of functions:

$$f = c_1(\, datagram \,)$$

and

$$vc = c_2(f)$$

where f identifies a particular flow, and vc identifies a connection. We will see below that separating the two functions provides flexibility in the possible mappings.

In practice function c_1 does not examine the entire datagram. Instead, only header fields are used. Strict *layer 3 classification* restricts computation to fields in the IP header such as the source and destination IP addresses and type of service. Most vendors implement *layer 4 classification*†, and some offer *layer 5 classification*. In addition to examining fields in the IP header, layer 4 classification schemes also examine protocol port numbers in the TCP or UDP header. Layer 5 schemes look further into the datagram and consider the application.

The concept of flows is important in switching IP because it allows the switch to track activity. For example, imagine that as it processes datagrams, a switch makes a list of (source, destination) pairs and keeps a counter with each. It does not make sense for a switch to optimize all routes because some flows only contain a few packets (e.g., when someone pings a remote computer). The count of flow activity provides a measure — when the count reaches a threshold, the switch begins to look for an optimized route. Layer 4 classification helps optimize flows because it allows the switch to know the approximate duration of a connection and whether traffic is caused by multiple TCP connections or a single connection.

Flows are also an important tool to make switched schemes work well with TCP. If a switch begins using a shortcut on a path that TCP is using, the round-trip time changes and some segments arrive out of order, causing TCP to adjust its retransmission timer. Thus, a switch using layer 4 classification can map each TCP session to a different flow, and then choose whether to map a flow to the original path or the shortcut. Most switching technologies employ hysteresis by retaining the original path for existing TCP connections, but using a shortcut for new connections (i.e., moving existing

†Vendors use the term *layer 4 switching* to characterize products that implement layer 4 classification.

connections to the shortcut after a fixed amount of time has elapsed or if the connection is idle).

18.26 Applicability Of Switching Technology

Although many vendors are pushing products that incorporate switched IP, there are several reasons why the technology has not had more widespread acceptance. First, in many cases switching costs more than conventional routing, but does not offer much increase in performance. The difference is most significant in the local area environment where inexpensive LANs, like Ethernet, have sufficient capacity and inexpensive routers work. In fact, computer scientists continue to find ways to improve IP forwarding schemes, which means that traditional routers can process more datagrams per second without requiring an increase in hardware speed. Second, the availability of inexpensive higher-speed LANs, such as gigabit Ethernet, has made organizations unwilling to use more expensive connection-oriented technology for an entire organization. Third, although switching IP appears straightforward, the details make it complex. Consequently, the protocols are significantly more complex than other parts of IP, which makes them more difficult to build, install, configure, and manage. We conclude that although there may be advantages to switched IP, it will not replace all traditional routers.

18.27 Summary

IP can be used over connection-oriented technologies; we examined ATM as a specific example. ATM is a high-speed network technology in which a network consists of one or more switches interconnected to form a switching fabric. The resulting system is characterized as a Non-Broadcast Multiple-Access technology because it appears to operate as a single, large network that provides communication between any two attached computers, but does not allow a single packet to be broadcast to all of them.

Because ATM is connection-oriented, two computers must establish a virtual circuit through the network before they can transfer data; a host can choose between a switched or permanent type of virtual circuit. Switched circuits are created on demand; permanent circuits require manual configuration. In either case, ATM assigns each open circuit an integer identifier. Each frame a host sends and each frame the network delivers contains a circuit identifier; a frame does not contain a source or destination address.

Although the lowest levels of ATM use 53-octet cells to transfer information, IP always uses ATM Adaptation Layer 5 (AAL5). AAL5 accepts and delivers variable-size blocks of data, where each block can be up to 64K octets. To send an IP datagram across ATM, the sender must form a virtual circuit connection to the destination, specify using AAL5 on the circuit, and pass each datagram to AAL5 as a single block of data. AAL5 adds a trailer, divides the datagram and trailer into cells for transmission across the network, and then reassembles the datagram before passing it to the operating system on the destination computer. IP uses a default MTU of 9180, and AAL5 performs the segmentation into cells.

A Logical IP Subnet (LIS) consists of a set of computers that use ATM in place of a LAN; the computers form virtual circuits among themselves over which they exchange datagrams. Because ATM does not support broadcasting, computers in an LIS use a modified form of ARP known as ATMARP. An ATMARP server performs all address binding; each computer in the LIS must register with the server by supplying its IP address and ATM address. As with conventional ARP, a binding obtained from ATMARP is aged. After the aging period, the binding must be revalidated or discarded. A related protocol, Inverse ATMARP, is used to discover the ATM and IP addresses of a remote computer connected by a permanent virtual circuit.

Switching hardware technology can be used with IP. An IP switch acts as a router, but also classifies IP datagrams and sends them across the switched network when possible. Layer 3 classification uses only the datagram header; layer 4 classification also examines the TCP or UDP header. MPLS is a new standard for switching IP that is designed to allow systems from multiple vendors to interoperate.

FOR FURTHER STUDY

Newman et. al. [April 1998] describes IP switching. Laubach and Halpern [RFC 2225] introduces the concept of Logical IP Subnet, defines the ATMARP protocol, and specifies the default MTU. Grossman and Heinanen [RFC 2684] describes the use of LLC/SNAP headers when encapsulating IP in AAL5.

Partridge [1994] describes gigabit networking in general, and the importance of cell switching in particular. De Prycker [1993] considers many of the theoretical underpinnings of ATM and discusses its relationship to telephone networks.

EXERCISES

18.1 If your organization has an ATM switch or ATM service, find the technical and economic specifications, and then compare the cost of using ATM with the cost of another technology such as Ethernet.

18.2 A typical connection between a host and a private ATM switch operates at 155 Mbps. Consider the speed of the bus on your favorite computer. What percentage of the bus is required to keep an ATM interface busy?

18.3 Many operating systems choose TCP buffer sizes to be multiples of 8K octets. If IP fragments datagrams for an MTU of 9180 octets, what size fragments result from a datagram that carries a TCP segment of 16K octets? of 24K octets?

18.4 Look at the definition of IPv6 described in Chapter 33. What new mechanism relates directly to ATM?

18.5 ATM is a best-effort delivery system in which the hardware can discard cells if the network becomes congested. What is the probability of datagram loss if the probability of loss of a single cell is $1/P$ and the datagram is 576 octets long? 1500 octets? 4500 octets? 9180 octets?

18.6 A typical remote login session using TCP generates datagrams of 41 octets: 20 octets of IP header, 20 octets of TCP header, and 1 octet of data. How many ATM cells are required to send such a datagram using the default IP encapsulation over AAL5?

18.7 How many cells, octets, and bits can be present on a fiber that connects to an ATM switch if the fiber is 3 meters long? 100 meters? 3000 meters? To find out, consider an ATM switch transmitting data at 155 Mbps. Each bit is a pulse of light that lasts $1/(155\times10^6)$ second. Assume the pulse travels at the speed of light, calculate its length, and compare to the length of the fiber.

18.8 A host can specify a two-level ATM address when requesting an SVC. What ATM network topologies are appropriate for a two-level addressing scheme? Characterize situations for which additional levels of hierarchy are useful.

18.9 An ATM network guarantees to deliver cells in order, but may drop cells if it becomes congested. Is it possible to modify TCP to take advantage of cell ordering to reduce protocol overhead? Why or why not?

18.10 Read about the LANE and MPOA standards that allow ATM to emulate an Ethernet or other local area network. What is the chief advantage of using ATM to emulate LANs? The chief disadvantage?

18.11 A large organization that uses ATM to interconnect IP hosts must divide hosts into logical IP subnets. Two extremes exist: the organization can place all hosts in one large LIS, or the organization can have many LIS (e.g., each pair of hosts forms an LIS). Explain why neither extreme is desirable.

18.12 How many ATM cells are required to transfer a single ATMARP packet when each ATM address and subaddress is 20 octets and each protocol address is 4 octets?

18.13 ATM allows a host to establish multiple virtual circuits to a given destination. What is the major advantage of doing so?

18.14 Measure the throughput and delay of an ATM switch when using TCP. If your operating system permits, repeat the experiment with the TCP transmit buffer set to various sizes (if your system uses sockets, refer to the manual for details on how to set the buffer size). Do the results surprise you?

18.15 IP does not have a mechanism to associate datagrams traveling across an ATM network with a specific ATM virtual circuit. Under what circumstances would such a mechanism be useful?

18.16 A server does not immediately remove an entry from its cache when the host that sent the information closes its connection to the server. What is the chief advantage of such a design? What is the chief disadvantage?

18.17 Is IP switching worthwhile for applications you run? To find out, monitor the traffic from your computer and find the average duration of TCP connections, the number of simultaneous connections, and the number of IP destinations you contact in a week.

18.18 Read about MPLS. Should MPLS accommodate layer 2 forwarding (i.e., bridging) as well as optimized IP forwarding? Why or why not?

19

Mobile IP

19.1 Introduction

Previous chapters describe the original IP addressing and routing schemes used with stationary computers. This chapter considers a recent extension of IP designed to allow portable computers to move from one network to another.

19.2 Mobility, Routing, and Addressing

In the broadest sense, the term *mobile computing* refers to a system that allows computers to move from one location to another. Mobility is often associated with wireless technologies that allow movement across long distances at high speed. However, speed is not the central issue for IP. Instead, a challenge only arises when a host changes from one network to another. For example, a notebook computer attached to a wireless LAN can move around the range of the transmitter rapidly without affecting IP, but simply unplugging a desktop computer and plugging it into a different network requires reconfiguring IP.

The IP addressing scheme, which was designed and optimized for a stationary environment, makes mobility difficult. In particular, because a host's IP address includes a network prefix, moving the host to a new network means either:

- The host's address must change.
- Routers must propagate a host-specific route across the entire internet.

Neither alternative works well. On one hand, changing an address is time-consuming, usually requires rebooting the computer, and breaks all existing transport-layer connections. In addition, if the host contacts a server that uses addresses to authenticate, an additional change to DNS may be required. On the other hand, a host-specific routing approach cannot scale because it requires space in routing tables proportional to the number of hosts, and because transmitting routes consumes excessive bandwidth.

19.3 Mobile IP Characteristics

The IETF devised a solution to the mobility problem that overcomes some of the limitations of the original IP addressing scheme. Officially named *IP mobility support*, it is popularly called *mobile IP*. The general characteristics include the following.

Transparency. Mobility is transparent to applications and transport layer protocols as well as to routers not involved in the change. In particular, as long as they remain idle, all open TCP connections survive a change in network and are ready for further use.

Interoperability with IPv4. A host using mobile IP can interoperate with stationary hosts that run conventional IPv4 software as well as with other mobile hosts. Furthermore, no special addressing is required — the addresses assigned to mobile hosts do not differ from addresses assigned to fixed hosts.

Scalability. The solution scales to large internets. In particular, it permits mobility across the global Internet.

Security. Mobile IP provides security facilities that can be used to ensure all messages are authenticated (i.e., to prevent an arbitrary computer from impersonating a mobile host).

Macro mobility. Rather than attempting to handle rapid network transitions such as one encounters in a wireless cellular system, mobile IP focuses on the problem of long-duration moves. For example, mobile IP works well for a user who takes a portable computer on a business trip, and leaves it attached to the new location for a week.

19.4 Overview Of Mobile IP Operation

The biggest challenge for mobility lies in allowing a host to retain its address without requiring routers to learn host-specific routes. Mobile IP solves the problem by allowing a single computer to hold two addresses simultaneously. The first address, which can be thought of as the computer's *primary address*, is permanent and fixed. It is the address applications and transport protocols use. The second address, which can be thought of as a *secondary address*, is temporary — it changes as the computer moves, and is valid only while the computer visits a given location.

A mobile host obtains a primary address on its original, *home* network. After it moves to a *foreign* network and obtains a secondary address, the mobile must send the secondary address to an *agent* (usually a router) at home. The agent agrees to intercept datagrams sent to the mobile's primary address, and uses *IP-in-IP* encapsulation to *tunnel* each datagram to the secondary address†.

†Chapter 17 illustrates IP-in-IP encapsulation.

If the mobile moves again, it obtains a new secondary address, and informs the home agent of its new location. When the mobile returns home, it must contact the home agent to *deregister*, meaning that the agent will stop intercepting datagrams. Similarly, a mobile can choose to deregister at any time (e.g., when leaving a remote location).

We said that mobile IP is designed for macroscopic mobility rather than high-speed movement. The reason should be clear: overhead. In particular, after it moves, a mobile must detect that it has moved, communicate across the foreign network to obtain a secondary address, and then communicate across the internet to its agent at home to arrange forwarding. The point is:

Because it requires considerable overhead after each move, mobile IP is intended for situations in which a host moves infrequently and remains at a given location for a relatively long period of time.

19.5 Mobile Addressing Details

A mobile's primary or *home address* is assigned and administered by the network administrator of the mobile's home network; there is no distinction between an address assigned to a stationary computer and a home address assigned to a mobile computer. Applications on a mobile computer always use the home address.

Whenever it connects to a network other than its home, a mobile must obtain a temporary address. Known as a *care of* address, the temporary address is never known or used by applications. Instead, only IP software on the mobile and agents on the home or foreign networks use the temporary address. A care-of address is administered like any other address on the foreign network, and a route to the care-of address is propagated using conventional routing protocols.

In practice, there are two types of care-of addresses; the type used by a mobile visiting a given network is determined by the network's administrator. The two types differ in the method by which the address is obtained and in the entity responsible for forwarding. The first form, which is known as a *co-located care-of address*, requires a mobile computer to handle all forwarding itself. In essence, a mobile that uses a co-located care-of address has software that uses two addresses simultaneously — applications use the home address, while lower layer software uses the care-of address to receive datagrams. The chief advantage of a co-located address lies in its ability to work with existing internet infrastructure. Routers on the foreign network do not know whether a computer is mobile; care-of addresses are allocated to mobile computers by the same mechanisms used to allocate addresses to fixed computers (e.g., the DHCP protocol discussed in Chapter 23). The chief disadvantage of the co-located form arises from the extra software required — the mobile must contain facilities to obtain an address and to communicate with the home agent.

The second form, which is known as a *foreign agent care-of address*, requires an active participant on the remote network. The active entity, also a router, is called a *foreign agent* to distinguish it from the *home agent* on the mobile's home network. When using a foreign agent care-of address, a mobile must first discover the identity of an agent, and then contact the agent to obtain a care-of address. Surprisingly, a foreign agent does not need to assign the mobile a unique address. Instead, we will see that the agent can supply one of its IP addresses, and agree to forward datagrams to the mobile. Although assigning a unique address makes communication slightly easier, using an existing address means that visiting mobiles do not consume IP addresses.

19.6 Foreign Agent Discovery

Known as *agent discovery*, the process of finding a foreign agent uses the ICMP *router discovery* mechanism. Recall from Chapter 9 that router discovery requires each router to periodically send an ICMP *router advertisement* message, and allows a host to send an ICMP *router solicitation* to prompt for an advertisement†. Agent discovery piggybacks additional information on router discovery messages to allow a foreign agent to advertise its presence or a mobile to solicit an advertisement. The additional information appended to each message is known as a *mobility agent extension*‡. Mobility extensions do not use a separate ICMP message type. Instead, a mobile host deduces that the extension is present when the datagram length specified in the IP header is greater than the length of the ICMP router discovery message. Figure 19.1 illustrates the extension format.

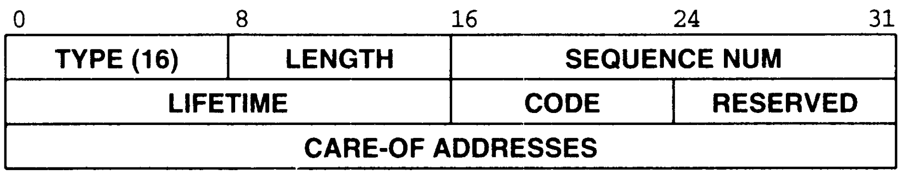


Figure 19.1 The format of a Mobility Agent Advertisement Extension message. This extension is appended to an ICMP router advertisement.

Each message begins with a 1-octet *TYPE* field followed by a 1-octet *LENGTH* field. The *LENGTH* field specifies the size of the extension message in octets, excluding the *TYPE* and *LENGTH* octets. The *LIFETIME* field specifies the maximum amount of time in seconds that the agent is willing to accept registration requests, with all 1s indicating *infinity*. Field *SEQUENCE NUM* specifies a sequence number for the message to allow a recipient to determine when a message is lost. Each bit in the *CODE* field defines a specific feature of the agent as listed in Figure 19.2.

†A mobile that does not know an agent's IP address can multicast to the *all agents group* (*224.0.0.11*).

‡A mobility agent also appends a *prefix extension* to the message that specifies the IP prefix being used on the network; a mobile uses the prefix extension to determine when it has moved to a new network.

Bit	Meaning
0	Registration with an agent is required; co-located care-of addressing is not permitted
1	The agent is busy and is not accepting registrations
2	Agent functions as a home agent
3	Agent functions as a foreign agent
4	Agent uses minimal encapsulation
5	Agent uses GRE-style encapsulation†
6	Agent supports header compression when communicating with mobile
7	Unused (must be zero)

Figure 19.2 Bits of the CODE field of a mobility agent advertisement.

19.7 Agent Registration

Before it can receive datagrams at a foreign location, a mobile host must register. The *registration* procedure allows a host to:

- Register with an agent on the foreign network.
- Register directly with its home agent to request forwarding.
- Renew a registration that is due to expire.
- Deregister after returning home.

If it obtains a co-located care-of address, a mobile performs all necessary registration directly; the mobile can use the address to communicate with its home agent and register. If it obtains a care-of address from a foreign agent, however, a mobile cannot use the address to communicate directly with its home agent. Instead, the mobile must send registration requests to the foreign agent, which then contacts the mobile's home agent on its behalf. Similarly, the foreign agent must forward messages it receives that are destined for the mobile host.

19.8 Registration Message Format

All registration messages are sent via UDP. Agents listen to well-known port 434; requests may be sent from an arbitrary source port to destination port 434. An agent reverses the source and destination points, so a reply is sent from source port 434 to the port the requester used.

A registration message begins with a set of fixed-size fields followed by variable-length *extensions*. Each request is required to contain a *mobile-home authentication extension* that allows the home agent to verify the mobile's identity. Figure 19.3 illustrates the message format.

†*GRE*, which stands for *Generic Routing Encapsulation*, refers to a generalized encapsulation scheme that allows an arbitrary protocol to be encapsulated; IP-in-IP is one particular case.

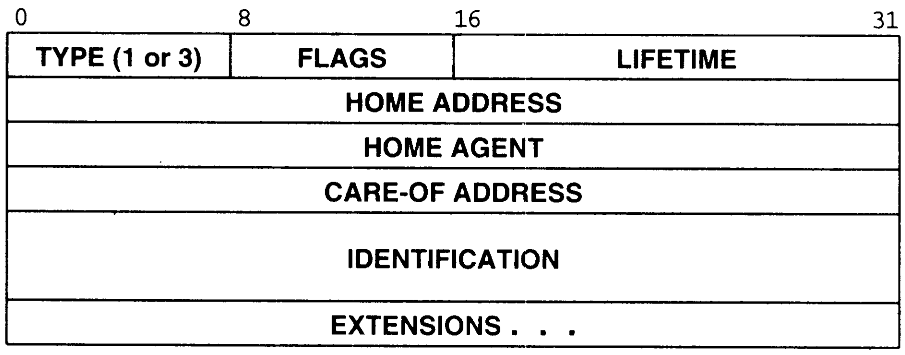


Figure 19.3 The format of a mobile IP registration message.

The *TYPE* field specifies whether the message is a registration request (*1*) or a registration reply (*3*). The *LIFETIME* field specifies the number of seconds the registration is valid (a zero requests immediate deregistration, and all 1s specifies an infinite lifetime). The *HOME ADDRESS*, *HOME AGENT*, and *CARE-OF ADDRESS* fields specify the two IP addresses of the mobile and the address of its home agent, and the *IDENTIFICATION* field contains a 64-bit number generated by the mobile that is used to match requests with incoming replies and to prevent the mobile from accepting old messages. Bits of the *FLAGS* field are used to specify forwarding details as listed in Figure 19.4.

Bit	Meaning
0	This is a simultaneous (additional) address rather than a replacement.
1	Mobile requests home agent to tunnel a copy of each broadcast datagram
2	Mobile is using a co-located care-of address and will decapsulate datagrams itself
3	Mobile requests agent to use minimal encapsulation
4	Mobile requests agent to use GRE encapsulation
5	Mobile requests header compression
6-7	Reserved (must be zero)

Figure 19.4 The meaning of *FLAGS* bits in a mobile registration request.

If it has a co-located care-of address, a mobile can send a registration request directly to its home agent. Otherwise, the mobile sends the request to a foreign agent, which then forwards the request to the home agent. In the latter case, both the foreign and home agents process the request, and both must approve. For example, either the home or foreign agents can limit the registration lifetime.

19.9 Communication With A Foreign Agent

We said that a foreign agent can assign one of its IP addresses for use as a care-of address. Doing so causes a problem because it means a mobile will not have a unique address on the foreign network. The question then becomes: how can a foreign agent and a mobile host communicate over a network if the mobile does not have a valid IP address on the network? Communication requires relaxing the rules for IP addressing and using an alternative scheme for address binding. In particular, when a mobile host sends to a foreign agent, the mobile is allowed to use its home address as an IP source address. Furthermore, when a foreign agent sends a datagram to a mobile, the agent is allowed to use the mobile's home address as an IP destination address.

Although the mobile's home address can be used, an agent is not allowed to ARP for the address (i.e., ARP is still restricted to IP addresses that are valid on the network). To perform address binding without ARP, an agent is required to record all information about a mobile when a registration request arrives and to keep the information during communication. In particular, an agent must record the mobile's hardware address. When it sends a datagram to the mobile, the agent consults its stored information to determine the appropriate hardware address. Thus, although ARP is not used, the agent can send datagrams to a mobile via hardware unicast. We can summarize:

> *If a mobile does not have a unique foreign address, a foreign agent must use the mobile's home address for communication. Instead of relying on ARP for address binding, the agent records the mobile's hardware address when a request arrives and uses the recorded information to supply the necessary binding.*

19.10 Datagram Transmission And Reception

Once it has registered, a mobile host on a foreign network can communicate with an arbitrary computer. To do so, the mobile creates a datagram that has the computer's address in the destination field and the mobile's home address in the source field†. The datagram follows the shortest path from the foreign network to the destination. However, a reply will not follow the shortest path directly to the mobile. Instead, the reply will travel to the mobile's home network. The home agent, which has learned the mobile's location from the registration, intercepts the datagram and uses *IP-in-IP* encapsulation to tunnel the datagram to the care-of address. If a mobile has a co-located care-of address, the encapsulated datagram passes directly to the mobile, which discareds the outer datagram and then processes the inner datagram. If a mobile is using a foreign agent for communication, the care-of address on the outer datagram specifies the foreign agent. When it receives a datagram from a home agent, a foreign agent decapsulates the datagram, consults its table of registered mobiles, and transmits the datagram across the local network to the appropriate mobile. To summarize:

†The foreign network and the ISP that connects it to the rest of the internet must agree to transmit datagrams with an arbitrary source address.

Because a mobile uses its home address as a source address when communicating with an arbitrary destination, each reply is forwarded to the mobile's home network, where an agent intercepts the datagram, encapsulates it in another datagram, and forwards it either directly to the mobile or to the foreign agent the mobile is using.

19.11 The Two-Crossing Problem

The description above highlights the major disadvantage of mobile IP: inefficient routing. Because a mobile uses its home address, a datagram sent to the mobile will be forwarded to the mobile's home network first and then to the mobile. The problem is especially severe because computer communication often exhibits *spatial locality of reference*, which means that a mobile visiting a foreign network will tend to communicate with computers on that network. To understand why mobile IP handles spatial locality poorly, consider Figure 19.5.

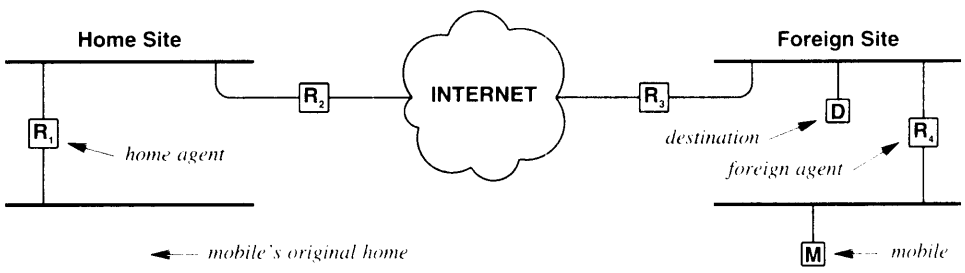


Figure 19.5 A topology in which mobile IP routing is inefficient. When mobile M communicates with local destination D, datagrams from D travel across the internet to the mobile's home agent and then back to the mobile.

In the figure, mobile M has moved from it's original home to a foreign network. We assume the mobile has registered with its home agent, router R_1, and the home agent has agreed to forward datagrams. Now consider communication between the mobile and destination D, which is located at the same site as the mobile. Datagrams from M to D travel through router R_4 and are then delivered to D. However, because datagrams sent from D to M contain M*'s* home address, they follow a path through R_3 and across the internet to the mobile's home network. When the datagrams reach R_1 (the mobile's home agent), they are tunneled back across the internet to the foreign site (either directly to M or to a foreign agent). Because crossing an internet is much more expensive than local delivery, the situation described above is known as the *two-crossing problem*, and is sometimes called the *2X problem*†.

†If destination D is not close to the mobile, a slightly less severe version of the problem occurs which is known as *triangle forwarding* or *dog-leg forwarding*.

Mobile IP does not guarantee to solve the 2X problem. However, some route optimization is possible. In particular, if a site expects a visiting mobile to interact heavily with local computers, the site can arrange to propagate a host-specific route for the mobile. To ensure correct routing, the host-specific route must be deleted when the mobile leaves. Of course, the problem remains whenever a mobile communicates with a destination outside the region where the host-specific route has been propagated. For example, suppose mobiles move frequently between two corporations in cities *A* and *B*. The network managers at the two sites can agree to propagate host-specific routes for all visiting mobiles, meaning that when a mobile communicates with other computers at the foreign site, traffic stays local to the site. However, because host-specific routes are limited to the two corporate sites, communication between the mobile and any other destination in the foreign city will result in replies being forwarded through the mobile's home agent. Thus, the 2X problem remains for any destination outside the corporation.

We can summarize:

> *Mobile IP introduces a routing inefficiency known as the 2X problem that occurs when a mobile visits a foreign network far from its home and then communicates with a computer near the foreign site. Each datagram sent to the mobile travels across the internet to the mobile's home agent which then forwards the datagram back to the foreign site. Eliminating the problem requires propagating host-specific routes; the problem remains for any destination that does not receive the host-specific route.*

19.12 Communication With Computers On the Home Network

We said that when a mobile is visiting a foreign network, the mobile's home agent must intercept all datagrams sent to the mobile. Normally, the home agent is the router that connects the mobile's home network to the rest of the internet. Thus, all datagrams that arrive for the host pass through the home agent. Before forwarding a datagram, the home agent examines its table of mobile hosts to determine whether the destination host is currently at home or visiting a foreign network.

Although a home agent can easily intercept all datagrams that arrive for a mobile host from outside, there is one additional case that the agent must handle: datagrams that originate locally. In particular, consider what happens when a host on the mobile's home network sends a datagram to a mobile. Because IP specifies direct delivery over the local network, the sender will not forward the datagram to a router. Instead, the sender will ARP for the mobile's hardware address, encapsulate the datagram, and transmit it.

If a mobile has moved to a foreign network, the home agent must intercept all datagrams, including those sent by local hosts. To guarantee that it can intercept datagrams from local hosts, the home agent uses *proxy ARP*. That is, a home agent must listen for ARP requests that specify the mobile as a target, and must answer the requests by supplying its own hardware address. Proxy ARP is completely transparent to local computers — any local system that ARPs for a mobile's address will receive a reply, and will forward the datagram as usual.

The use of proxy ARP also solves the problem of multiple connections. If a mobile's home network has multiple routers that connect to various parts of the internet, only one needs to function as a home agent for the mobile. The other routers remain unaware of mobility; they use ARP to resolve addresses as usual. Thus, because the home agent answers the ARP requests, other routers forward datagrams without distinguishing between mobile and nonmobile hosts.

19.13 Summary

Mobile IP allows a computer to move from one network to another without changing its IP address and without requiring all routers to propagate a host-specific route. When it moves from its original home network to a foreign network, a mobile computer must obtain an additional, temporary address known as a care-of address. Applications use the mobile's original, home address; the care-of address is only used by underlying network software to enable forwarding and delivery across the foreign network.

Once it detects that it has moved, a mobile either obtains a co-located care-of address or discovers a foreign mobility agent and requests the agent to assign a care-of address. After obtaining a care-of address, the mobile registers with its home agent (either directly or indirectly through the foreign agent), and requests the agent to forward datagrams.

Once registration is complete, a mobile can communicate with an arbitrary computer on the internet. Datagrams sent by the mobile are forwarded directly to the specified destination. However, each datagram sent back to the mobile follows a route to the mobile's home network where it is intercepted by the home agent, encapsulated in IP, and then tunneled to the mobile.

FOR FURTHER STUDY

Perkins [RFC 2002] describes IP Mobility Support and defines the details of messages; an Internet draft describes version 2 [draft-ietf-mobileip-v2-00.txt]. Perkins [RFC 2003], Perkins [RFC 2004], and Hanks et. al. {RFC 1701] describe the details of three IP-in-IP encapsulation schemes. Montenegro [RFC 2344] describes a reverse tunneling scheme for mobile IP. Finally, Perkins and Johnson [draft-ietf-mobileip-optim-07.txt] considers route optimization for mobile IP.

EXERCISES

19.1 Compare the encapsulation schemes in RFCs 2003 and 2004. What are the advantages and disadvantages of each?

19.2 Read the mobile IP specification carefully. How frequently must a router send a mobility agent advertisement? Why?

19.3 Consult the mobile IP specification. When a foreign agent forwards a registration request to a mobile's home agent, which protocol ports are used? Why?

19.4 The specification for mobile IP allows a single router to function as both a home agent for a network and a foreign agent that supports visitors on the network. What are the advantages and disadvantages of using a single router for both functions?

19.5 The mobile IP specification defines three conceptually separate forms of authentication: mobile to home agent, mobile to foreign agent, and foreign agent to home agent. What are the advantages of separating them? The disadvantages?

19.6 Read the mobile IP specification to determine how a mobile host joins a multicast group. How are multicast datagrams routed to the mobile? What is the optimal scheme?

20

Private Network Interconnection (NAT, VPN)

20.1 Introduction

Previous chapters describe an internet as a single-level abstraction that consists of networks interconnected by routers. This chapter considers an alternative — a two-level internet architecture in which each organization has a private internet and a central internet interconnects them.

The chapter examines technologies used with a two-level architecture. One solves the pragmatic problem of limited address space, and the other offers increased functionality in the form of *privacy* that prevents outsiders from viewing the data.

20.2 Private And Hybrid Networks

One of the major drawbacks of a single-level internet architecture is the lack of privacy. If an organization comprises multiple sites, the contents of datagrams that travel across the Internet between the sites can be viewed by outsiders because they pass across networks owned by other organizations. A two-level architecture distinguishes between *internal* and *external* datagrams (i.e., datagrams sent between two computers within an organization and datagrams sent between a computer in the organization and a computer in another organization). The goal is to keep internal datagrams *private*, while still allowing external communication.

The easiest way to guarantee privacy among an organization's computers consists of building a completely isolated *private internet*, which is usually referred to as a *private network*. That is, an organization builds its own TCP/IP internet separate from the global Internet. A private network uses routers to interconnect networks at each site, and leased digital circuits to interconnect the sites. All data remains private because no outsiders have access to any part of a private network. Furthermore, because the private network is isolated from the global Internet, it can use arbitrary IP addresses.

Of course, complete isolation is not always desirable. Thus, many organizations choose a *hybrid network* architecture that combines the advantages of private networking with the advantages of global Internet connectivity. That is, the organization uses globally valid IP addresses and connects each site to the Internet. The advantage is that hosts in the organization can access the global Internet when needed, but can be assured of privacy when communicating internally. For example, consider the hybrid architecture illustrated by Figure 20.1 in which an organization has a private network that interconnects two sites and each site has a connection to the Internet.

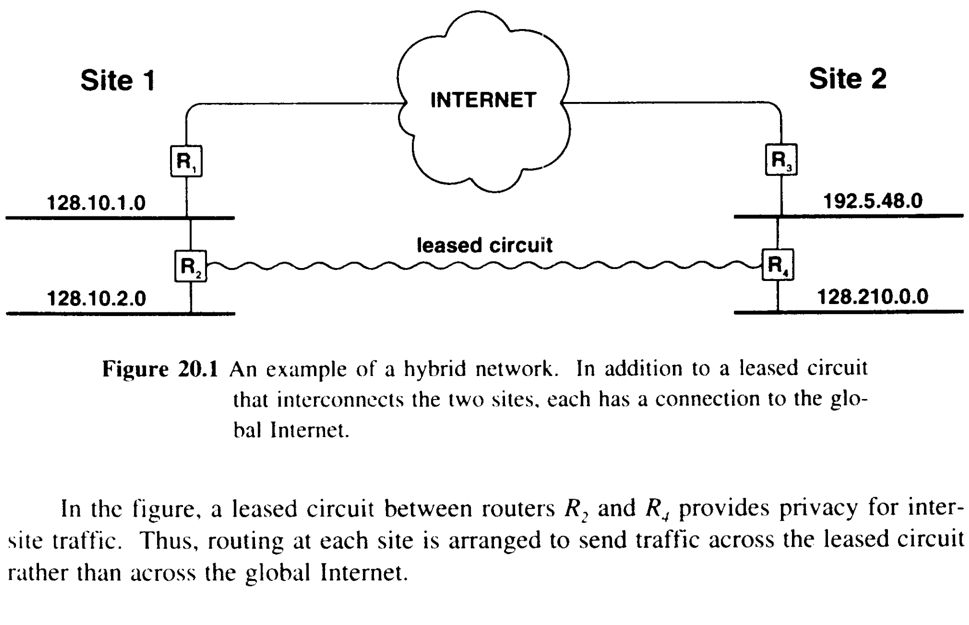


Figure 20.1 An example of a hybrid network. In addition to a leased circuit that interconnects the two sites, each has a connection to the global Internet.

In the figure, a leased circuit between routers R_2 and R_4 provides privacy for inter-site traffic. Thus, routing at each site is arranged to send traffic across the leased circuit rather than across the global Internet.

20.3 A Virtual Private Network (VPN)

The chief disadvantage of either a completely private network or a hybrid scheme arises from the high cost: each leased circuit (e.g., a T1 line) is expensive. Consequently, many organizations seek lower-cost alternatives. One way to reduce costs arises from the use of alternative circuit technologies. For example, a common carrier may change less for a Frame Relay or ATM PVC than for a T-series circuit that has equivalent capacity. Another way to lower costs involves using fewer circuits. Minimum circuit cost is achieved by eliminating all circuits and passing data across the global Internet.

Using the global Internet as an interconnection among sites appears to eliminate the privacy offered by a completely private network. The question becomes:

How can an organization that uses the global Internet to connect its sites keep its data private?

The answer lies in a technology that allows an organization to configure a *Virtual Private Network* (*VPN*)†. A VPN is *private* in the same way as a private network — the technology guarantees that communication between any pair of computers in the VPN remains concealed from outsiders. A VPN is *virtual* because it does not use leased circuits to interconnect sites. Instead, a VPN uses the global Internet to pass traffic from one site to another.

Two basic techniques make a VPN possible: *tunneling* and *encryption*. We have already encountered tunneling in Chapters 17 and 19. VPNs use the same basic idea — they define a tunnel across the global Internet between a router at one site and a router at another, and use *IP-in-IP* encapsulation to forward datagrams across the tunnel.

Despite using the same basic concept, a VPN tunnel differs dramatically from the tunnels described previously. In particular, to guarantee privacy, a VPN encrypts each outgoing datagram before encapsulating it in another datagram for transmission‡. Figure 20.2 illustrates the concept.

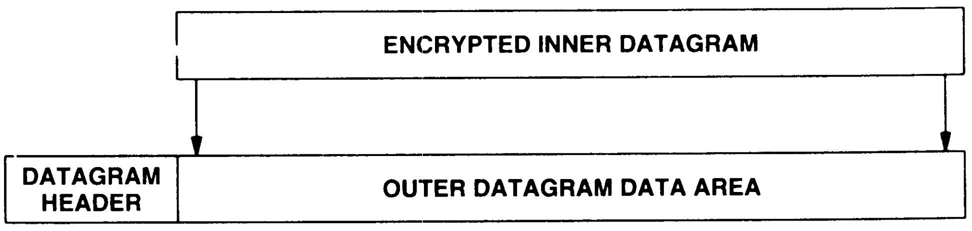


Figure 20.2 Illustration of IP-in-IP encapsulation used with a VPN. To ensure privacy, the inner datagram is encrypted before being sent.

As the figure shows, the entire inner datagram, including the header, is encrypted before being encapsulated. When a datagram arrives over a tunnel, the receiving router decrypts the data area to reproduce the inner datagram, which it then forwards. Although the outer datagram traverses arbitrary networks as it passes across the tunnel, outsiders cannot decode the contents because they do not have the encryption key. Furthermore, even the identity of the original source and destination are hidden because the header of the inner datagram is encrypted as well. Thus, only addresses in the outer datagram header are visible: the source address is the IP address of the router at one end of a tunnel, and the destination address is the IP address of the router at the other end of the tunnel.

†The name is a slight misnomer because the technology actually provides a virtual private internet.

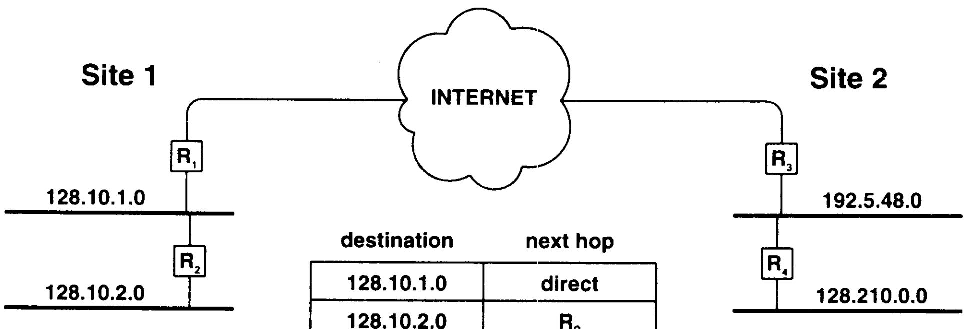
‡Chapter 32 considers IP security, and discusses the encapsulation used with IPsec.

To summarize:

A Virtual Private Network sends data across the Internet, but encrypts intersite transmissions to guarantee privacy.

20.4 VPN Addressing And Routing

The easiest way to understand VPN addressing and routing is to think of each VPN tunnel as a replacement for a leased circuit in a private network. As in the private network case, a router contains explicit routes for destinations within the organization. However, instead of routing data across a leased lined, a VPN routes the data through a tunnel. For example Figure 20.3 shows the VPN equivalent of the private network architecture from Figure 20.1 along with a routing table for a router that handles tunneling.



destination	next hop
128.10.1.0	direct
128.10.2.0	R_2
192.5.48.0	tunnel to R_3
128.210.0.0	tunnel to R_3
default	ISP's router

Routing table in R_1

Figure 20.3 A VPN that spans two sites and R_1's routing table. The tunnel from R_1 to R_3 is configured like a point-to-point leased circuit.

As an example of forwarding in a VPN, consider a datagram sent from a computer on network *128.10.2.0* to a computer on network *128.210.0.0*. The sending host forwards the datagram to R_2, which forwards it to R_1. According to the routing table in R_1, the datagram must be sent across the tunnel to R_3. Therefore, R_1 encrypts the datagram, encapsulates it in the data area of an outer datagram with destination R_3. R_1 then forward the outer datagram through the local ISP and across the Internet. The datagram arrives at R_3, which recognizes it as tunneled from R_1. R_3 decrypts the data area to produce the original datagram, looks up the destination in its routing table, and forwards the datagram to R_4 for delivery.

20.5 A VPN With Private Addresses

A VPN offers an organization the same addressing options as a private network. If hosts in the VPN do not need general Internet connectivity, the VPN can be configured to use arbitrary IP addresses; if hosts need Internet access, a hybrid addressing scheme can be used. A minor difference is that when private addressing is used, one globally valid IP address is needed at each site for tunneling. Figure 20.4 illustrates the concept.

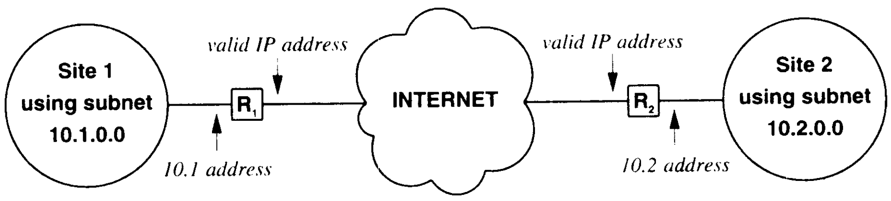


Figure 20.4 Illustration of addressing for a VPN that interconnects two completely private sites over the global Internet. Computers at each site use private addresses.

As the figure shows, site 1 uses subnet 10.1.0.0/16, while site 2 uses subnet 10.2.0.0/16. Only two globally valid addresses are needed. One is assigned to the connection from router R_1 to the Internet, and the other is assigned to the connection from R_2 to the Internet. Routing tables at the sites specify routes for private addresses; only the VPN tunneling software needs to know about or use the globally valid IP addresses.

VPNs use the same addressing structure as a private network. Hosts in a completely isolated VPN can use arbitrary addresses, but a hybrid architecture with valid IP addresses must be employed to provide hosts with access to the global Internet. The question remains: "How can a site provide access to the global Internet without assigning each host a valid IP address?" There are two general solutions.

Known as an *application gateway* approach, the first solution offers hosts access to Internet services without offering IP-level access. Each site has a multi-homed host connected to both the global Internet (with a globally valid IP address) and the internal network (using a private IP address). The multi-homed host runs a set of application programs, known as *application gateways*, that each handle one service. Hosts at the site do not send datagrams to the global Internet. Instead, they send each request to the appropriate application gateway on the multihomed host, which accesses the service on the Internet and then relays the information back across the internal network. For example, Chapter 27 describes an e-mail gateway that can relay e-mail messages between external hosts and internal hosts.

The chief advantage of the application gateway approach lies in its ability to work without changes to the underlying infrastructure or addressing. The chief disadvantage arises from the lack of generality, which can be summarized:

> *Each application gateway handles only one specific service; multiple gateways are required for multiple services.*

Consequently, although they are useful in special circumstances, application gateways do not solve the problem in a general way. Thus, a second solution was invented.

20.6 Network Address Translation (NAT)

A technology has been created that solves the general problem of providing IP-level access between hosts at a site and the rest of the Internet, without requiring each host at the site to have a globally valid IP address. Known as *Network Address Translation* (*NAT*), the technology requires a site to have a single connection to the global Internet and at least one globally valid IP address, *G*. Address *G* is assigned to a computer (a multi-homed host or a router) that connects the site to the Internet and runs NAT software. Informally, we refer to a computer that runs NAT software as a *NAT box*; all datagrams pass through the NAT box as they travel from the site out to the Internet or from the Internet into the site.

NAT translates the addresses in both outgoing and incoming datagrams by replacing the source address in each outgoing datagram with *G* and replacing the destination address in each incoming datagram with the private address of the correct host. Thus, from the view of an external host, all datagrams come from the NAT box and all responses return to the NAT box. From the view of internal hosts, the NAT box appears to be a router that can reach the global Internet.

The chief advantage of NAT arises from its combination of generality and transparency. NAT is more general than application gateways because it allows an arbitrary internal host to access an arbitrary service on a computer in the global Internet. NAT is transparent because it allows an internal host to send and receive datagrams using a private (i.e., nonroutable) address.

To summarize:

> *Network Address Translation technology provides transparent IP-level access to the Internet from a host with a private address.*

20.7 NAT Translation Table Creation

Our overview of NAT omits an important detail because it does not specify how NAT knows which internal host should receive a datagram that arrives from the Internet. In fact, NAT maintains a translation table that it uses to perform the mapping. Each entry in the table specifies two items: the IP address of a host on the Internet and the internal IP address of a host at the site. When an incoming datagram arrives from the Internet, NAT looks up the datagram's destination address in the translation table, extracts the corresponding address of an internal host, replaces the datagram's destination address with the host's address, and forwards the datagram across the local network to the host†.

The NAT translation table must be in place before a datagram arrives from the Internet. Otherwise, NAT has no way to identify the correct internal host to which the datagram should be forwarded. How and when is the table initialized? There are several possibilities:

- *Manual initialization.* A manager configures the translation table manually before any communication occurs.
- *Outgoing datagrams.* The table is built as a side-effect of sending datagrams. When it receives a datagram from an internal host, NAT creates an entry in the translation table to record the address of the host and the address of the destination.
- *Incoming name lookups.* The table is built as a side-effect of handing domain name lookups. When a host on the Internet looks up the domain name of an internal host to find its IP address‡, the domain name software creates an entry in the NAT translation table. and then answers the request by sending address *G*. Thus, from outside the site, it appears that all host names at the site map to address *G*.

Each initialization technique has advantages and disadvantages. Manual initialization provides permanent mappings and allows IP datagrams to be sent in either direction at any time. Using an outgoing datagram to initialize the table has the advantage of being automatic, but does not allow communication to be initiated from the outside. Using incoming domain name lookups requires modifying domain name software. It accommodates communication initiated from outside the site, but only works if the sender performs a domain name lookup before sending datagrams.

Most implementations of NAT use outgoing datagrams to initialize the table; the strategy is especially popular among ISPs. To understand why, consider a small ISP that serves dialup customers. Figure 20.5 illustrates the architecture.

†Of course, whenever it replaces an address in a datagram header, NAT must recompute the header checksum.

‡Chapter 24 describes how the *Domain Name System (DNS)* operates.

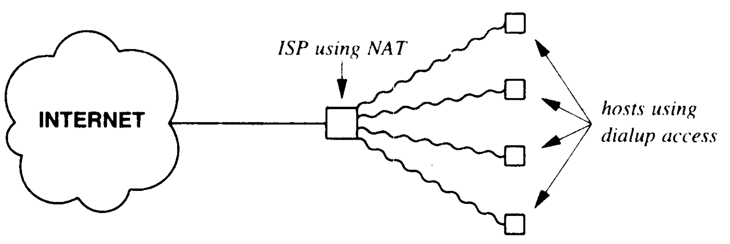


Figure 20.5 The use of NAT by a small ISP that serves dialup customers. NAT translation allows the ISP to assign a private address to each dialup customer.

The ISP must assign an IP address to a customer whenever the customer dials in. NAT permits the ISP to assign private addresses (e.g., the first customer is assigned *10.0.0.1*, the second *10.0.0.2*, and so on). When a customer sends a datagram to a destination on the Internet, NAT uses the outgoing datagram to initialize its translation table.

20.8 Multi-Address NAT

So far, we have described a simplistic implementation of NAT that performs a 1-to-1 address mapping between an external address and an internal address. That is, a 1-to-1 mapping permits at most one computer at the site to access a given machine on the global Internet at any time. In practice, more complex forms of NAT are used that allow multiple hosts at a site to access a given external address concurrently.

One variation of NAT permits concurrency by retaining the 1-to-1 mapping, but allowing the NAT box to hold multiple Internet addresses. Known as *multi-address NAT*, the scheme assigns the NAT box a set of K globally valid addresses, G_1, G_2,... G_k. When the first internal host accesses a given destination, the NAT box chooses address G_1, adds an entry to the translation table, and sends the datagram. If another host initiates contact with the same destination, the NAT box chooses address G_2, and so on. Thus, multi-address NAT allows up to K internal hosts to access a given destination concurrently.

20.9 Port-Mapped NAT

Another popular variant of NAT provides concurrency by translating TCP or UDP protocol port numbers as well as addresses. Sometimes called *Network Address Port Translation* (*NAPT*), the scheme expands the NAT translation table to include additional fields. Besides a pair of source and destination IP addresses, the table contains a pair of source and destination protocol port numbers and a protocol port number used by the NAT box. Figure 20.6 illustrates the contents of the table.

Private Address	Private Port	External Address	External Port	NAT Port	Protocol Used
10.0.0.5	21023	128.10.19.20	80	14003	tcp
10.0.0.1	386	128.10.19.20	80	14010	tcp
10.0.2.6	26600	207.200.75.200	21	14012	tcp
10.0.0.3	1274	128.210.1.5	80	14007	tcp

Figure 20.6 An example of a translation table used by NAPT. The table includes port numbers as well as IP addresses.

The table in the figure has entries for four internal computers that are currently accessing destinations on the global Internet. All communication is using TCP. Interestingly, the table shows two internal hosts, *10.0.0.5* and *10.0.0.1*, both accessing protocol port *80* (a Web server) on computer *128.10.19.20*. In this case, it happens that the two source ports being used for the two connections differ. However, source port uniqueness cannot be guaranteed — it could turn out that two internal hosts happen to choose the same source port number. Thus, to avoid potential conflicts, NAT assigns a unique port number to each communication that is used on the Internet. Recall that TCP identifies each connection with a 4-tuple that represents the IP address and protocol port number of each endpoint. The first two items in the table correspond to TCP connections that the two internal hosts identify with the 4-tuples:

(10.0.0.5, 23023, 128.10.19.20, 80)
(10.0.0.1, 386, 128.10.19.20, 80)

However, the computer in the Internet that receives datagrams after NAPT performs the translation identifies the same two connections with the 4-tuples:

(G, 14003, 128.10.19.20, 80)
(G, 14010, 128.10.19.20, 80)

where *G* is the globally valid address of the NAT box.

The primary advantage of NAPT lies in the generality it achieves with a single globally valid IP address; the primary disadvantage arises because it restricts communication to TCP or UDP. As long as all communication uses TCP or UDP, NAPT allows an internal computer to access multiple external computers, and multiple internal computers to access the same external computer without interference. A port space of 16 bits allows up to 2^{16} pairs of applications to communicate at the same time. To summarize:

Several variants of NAT exist, including the popular NAPT form that translates protocol port numbers as well as IP addresses.

20.10 Interaction Between NAT And ICMP

Even straightforward changes to an IP address can cause unexpected side-effects in higher layer protocols. In particular, to maintain the illusion of transparency, NAT must handle ICMP. For example, suppose an internal host uses *ping* to test reachability of a destination on the Internet. The host expects to receive an ICMP *echo reply* for each ICMP *echo request* message it sends. Thus, NAT must forward incoming echo replies to the correct host. However, NAT does not forward all ICMP messages that arrive from the Internet. If routes in the NAT box are incorrect, for example, an ICMP *redirect* message must be processed locally. Thus, when an ICMP message arrives from the Internet, NAT must first determine whether the message should be handled locally or sent to an internal host. Before forwarding to an internal host, NAT translates the ICMP message.

To understand the need for ICMP translation, consider an ICMP *destination unreachable* message. The message contains the header from a datagram, *D*, that caused the error. Unfortunately, NAT translated addresses before sending *D*, so the source address is not the address the internal host used. Thus, before forwarding the message, NAT must open the ICMP message and translate the addresses in *D* so they appear in exactly the form that the internal host used. After making the change, NAT must recompute the checksum in *D*, the checksum in the ICMP header, and the checksum in the outer datagram header.

20.11 Interaction Between NAT And Applications

Although ICMP makes NAT complex, application protocols have a more serious effect. In general, NAT will not work with any application that sends IP addresses or protocol ports as data. For example, when two programs use the *File Transfer Protocol* (*FTP*) described in Chapter 26, they have a TCP connection between them. As part of the protocol, one program obtains a protocol port on the local machine, converts the number to ASCII, and sends the result across a TCP connection to another program. If the connection between the programs passes through NAPT from an internal host to a host on the Internet, the port number in the data stream must be changed to agree with the port number NAPT has selected instead of the port the internal host is using. In fact, if NAT fails to open the data stream and change the number, the protocol will fail. Implementations of NAT have been created that recognize popular protocols such as FTP and make the necessary change in the data stream. However, there exist applications that cannot use NAT. To summarize:

> *NAT affects ICMP and higher layer protocols; except for a few standard applications like FTP, an application protocol that passes IP addresses or protocol port numbers as data will not operate correctly across NAT.*

Changing items in a data stream increases the complexity of NAPT in two ways. First, it means that NAPT must have detailed knowledge of each application that transfers such information. Second, if the port numbers are represented in ASCII, as is the case with FTP, changing the value can change the number of octets transferred. Inserting even one additional octet into a TCP connection is difficult because each octet in the stream has a sequence number. Because a sender does not know that additional data has been inserted, it continues to assign sequence numbers without the additional data. When it receives additional data, the receiver will generate acknowledgements that account for the data. Thus, after it inserts additional data, NAT must translate the sequence numbers in each outgoing segment and each incoming acknowledgement.

20.12 Conceptual Address Domains

We have described NAT as a technology that can be used to connect a private network to the global Internet. In fact, NAT can be used to interconnect any two *address domains*. Thus, NAT can be used between two corporations that each have a private network using address 10.0.0.0. More important, NAT can be used at two levels: between a customer's private and an ISP's private address domains as well as between the ISP's address domain and the global Internet. Finally, NAT can be combined with VPN technology to form a hybrid architecture in which private addresses are used within the organization, and NAT is used to provide connectivity between each site and the global Internet.

As an example of multiple levels of NAT, consider an individual who works at home from several computers which are connected to a LAN. The individual can assign private addresses to the computers at home, and use NAT between the home network and the corporate intranet. The corporation can also assign private addresses and use NAT between its intranet and the global Internet.

20.13 Slirp And Masquerade

Two implementations of Network Address Translation have become especially popular; both were designed for the Unix operating system. The *slirp* program, derived from 4.4 BSD, comes with program source code. It was designed for use in a dialup architecture like the one shown in Figure 20.5. Slirp combines PPP and NAT into a single program. It runs on a computer that has: a valid IP address, a permanent Internet connection, and one or more dialup modems. The chief advantage of slirp is that it can use an ordinary user account on a Unix system for general-purpose Internet access. A computer that has a private address dials in and runs slirp. Once slirp begins, the dialup line switches from ASCII commands to PPP. The dialup computer starts PPP and obtains access to the Internet (e.g., to access a Web site).

Slirp implements NAPT — it uses protocol port numbers to demultiplex connections, and can rewrite protocol port numbers as well as IP addresses. It is possible to have multiple computers (e.g., computers on a LAN) accessing the Internet at the same time through a single occurrence of slirp running on a UNIX system.

Another popular implementation of NAT has been designed for the Linux operating system. Known as *masquerade*, the program implements NAPT. Unlike slirp, masquerade does not require computers to access it via dialup, nor does masquerade need a user to login to the UNIX system before starting it. Instead, masquerade offers many options; it can be configured to operate like a router between two networks, and it handles most of the NAT variations discussed in this chapter, including the use of multiple IP addresses.

20.14 Summary

Although a private network guarantees privacy, the cost can be high. Virtual Private Network (VPN) technology offers a lower cost alternative that allows an organization to use the global Internet to interconnect multiple sites and use encryption to guarantee that intersite traffic remains private. Like a traditional private network, a VPN can either be completely isolated (in which case hosts are assigned private addresses) or a hybrid architecture that allows hosts to communicate with destinations on the global Internet.

Two technologies exist that provide communication between hosts in different address domains: application gateways and Network Address Translation (NAT). An application gateway acts like a proxy by receiving a request from a host in one domain, sending the request to a destination in another, and then returning the result to the original host. A separate application gateway must be installed for each service.

Network Address Translation provides transparent IP-level access to the global Internet from a host that has a private address. NAT is especially popular among ISPs because it allows customers to access arbitrary Internet services while using a private IP address. Applications that pass address or port information in the data stream will not work with NAT until NAT has been programmed to recognize the application and make the necessary changes in the data; most implementations of NAT only recognize a few (standard) services.

FOR FURTHER STUDY

Many router and software vendors sell Virtual Private Network technologies, usually with a choice of encryption schemes and addressing architecture. Consult the vendors' literature for more information.

Several versions of NAT are also available commercially. The charter of the IETF working group on NAT can be found at:

http://www.ietf.org/html.charters/nat-charter.html

In addition, Srisuresh and Holdrege [RFC 2663] defines NAT terminology, and the Internet Draft repository at

http://www.ietf.org/ID.html

contains several Internet Drafts on NAT.

More details about the masquerade program can be found in the Linux documentation. A resource page can be found at URL:

http://ipmasq.cjb.net

More information on slirp can be found in the program documentation; a resource page for slirp can be found at:

http://blitzen.canberra.edu.au/slirp

EXERCISES

20.1 Under what circumstances will a VPN transfer substantially more packets than conventional IP when sending the same data across the Internet? Hint: think about encapsulation.

20.2 Read the slirp document to find out about *port redirection*. Why is it needed?

20.3 What are the potential problems when three address domains are connected by two NAT boxes?

20.4 In the previous question, how many times will a destination address be translated? A source address?

20.5 Consider an ICMP host unreachable message sent through two NAT boxes that interconnect three address domains. How many address translations will occur? How many translations of protocol port numbers will occur?

20.6 Imagine that we decide to create a new Internet parallel to the existing Internet that allocates addresses from the same address space. Can NAT technology be used to connect the two arbitrarily large Internets that use the same address space? If so, explain how. If not, explain why not.

20.7 Is NAT completely transparent to a host? To answer the question, try to find a sequence of packets that a host can transmit to determine whether it is located behind a NAT box.

20.8 What are the advantages of combining NAT technology with VPN technology? The disadvantages?

20.9 Obtain a copy of slirp and instrument it to measure performance. Does slirp processing overhead ever delay datagrams? Why or why not?

20.10 Obtain NAT and configure it on a Linux system between a private address domain and the Internet. Which well-known services work correctly and which do not?

20.11 Read about a variant of NAT called *twice NAT* that allows communication to be initiated from either side of the NAT box at any time. How does twice NAT ensure that translations are consistent? If two instances of twice NAT are used to interconnect three address domains, is the result completely transparent to all hosts?

21

Client-Server Model Of Interaction

21.1 Introduction

Early chapters present the details of TCP/IP technology, including the protocols that provide basic services and the router architecture that provides needed routing information. Now that we understand the basic technology, we can examine application programs that profit from the cooperative use of a TCP/IP internet. While the example applications are both practical and interesting, they do not comprise the main emphasis. Instead, focus rests on the patterns of interaction among the communicating application programs. The primary pattern of interaction among cooperating applications is known as the *client-server* paradigm†. Client-server interaction forms the basis of most network communication, and is fundamental because it helps us understand the foundation on which distributed algorithms are built. This chapter considers the relationship between client and server; later chapters illustrate the client-server pattern with further examples.

21.2 The Client-Server Model

The term *server* applies to any program that offers a service that can be reached over a network. A server accepts a request over the network, performs its service, and returns the result to the requester. For the simplest services, each request arrives in a single IP datagram and the server returns a response in another datagram.

†Marketing literature sometimes substitutes the term *application-server* for client-server; the underlying scientific principle is unchanged.

An executing program becomes a *client* when it sends a request to a server and waits for a response. Because the client-server model is a convenient and natural extension of interprocess communication on a single machine, it is easy to build programs that use the model to interact.

Servers can perform simple or complex tasks. For example, a *time-of-day server* merely returns the current time whenever a client sends the server a packet. A *web server* receives requests from a browser to fetch a copy of a Web page; the server obtains a copy of the file for the page and returns it to the browser.

Usually, servers are implemented as application programs†. The advantage of implementing servers as application programs is that they can execute on any computing system that supports TCP/IP communication. Thus, the server for a particular service can execute on a timesharing system along with other programs, or it can execute on a personal computer. Multiple servers can offer the same service, and can execute on the same machine or on multiple machines. In fact, managers commonly replicate copies of a given server onto physically independent machines to increase reliability or improve performance. If a computer's primary purpose is support of a particular server program, the term "server" may be applied to the computer as well as to the server program. Thus, one hears statements such as "machine *A* is our file server."

21.3 A Simple Example: UDP Echo Server

The simplest form of client-server interaction uses unreliable datagram delivery to convey messages from a client to a server and back. Consider, for example, a *UDP echo server.* The mechanics are straightforward as Figure 21.1 shows. At the server site, a UDP *echo server process* begins by negotiating with its operating system for permission to use the UDP port ID reserved for the *echo* service, the UDP *echo port.* Once it has obtained permission, the echo server process enters an infinite loop that has three steps: (1) wait for a datagram to arrive at the echo port, (2) reverse the source and destination addresses‡ (including source and destination IP addresses as well as UDP port ids), and (3) return the datagram to its original sender. At some other site, a program becomes a UDP *echo client* when it allocates an unused UDP protocol port, sends a UDP message to the UDP echo server, and awaits the reply. The client expects to receive back exactly the same data as it sent.

The UDP echo service illustrates two important points that are generally true about client-server interaction. The first concerns the difference between the lifetime of servers and clients:

> *A server starts execution before interaction begins and (usually) continues to accept requests and send responses without ever terminating. A client is any program that makes a request and awaits a response; it (usually) terminates after using a server a finite number of times.*

†Many operating systems refer to a running application program as a *process*, a *user process*, or a *task*.
‡One of the exercises suggests considering this step in more detail.

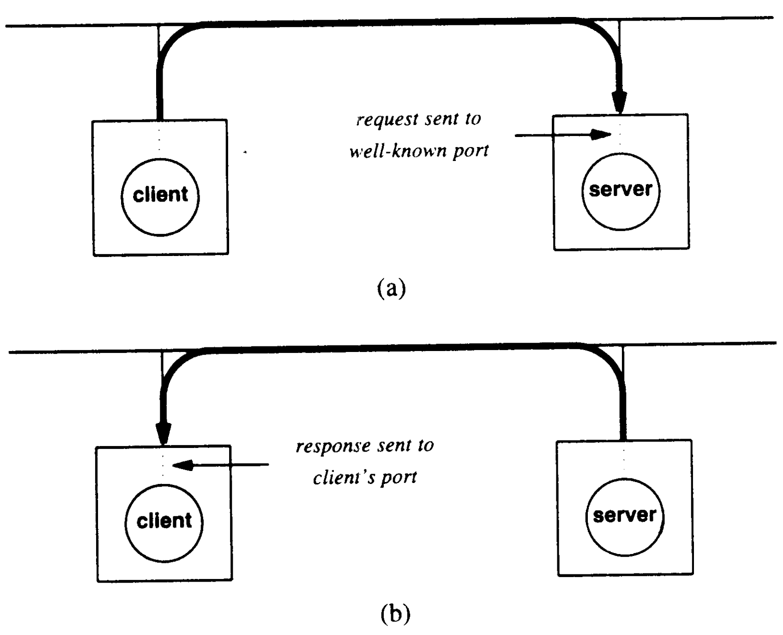


Figure 21.1 UDP echo as an example of the client-server model. In (a) the client sends a request to the server at a known IP address and at a well-known UDP port, and in (b) the server returns a response. Clients use any UDP port that is available.

The second point, which is more technical, concerns the use of reserved and nonreserved port identifiers:

> *A server waits for requests at a well-known port that has been reserved for the service it offers. A client allocates an arbitrary, unused, nonreserved port for its communication.*

In a client-server interaction, only one of the two ports needs to be reserved. Assigning a unique port identifier to each service makes it easy to build both clients and servers.

Who would use an echo service? It is not a service that the average user finds interesting. However, programmers who design, implement, measure, or modify network protocol software, or network managers who test routes and debug communication problems, often use echo servers in testing. For example, an echo service can be used to determine if it is possible to reach a remote machine.

21.4 Time And Date Service

The echo server is extremely simple, and little code is required to implement either the server or client side (provided that the operating system offers a reasonable way to access the underlying UDP/IP protocols). Our second example, a time server, shows that even simple client-server interaction can provide useful services. The problem a time server solves is that of setting a computer's time-of-day clock. The time of day clock is a hardware device that maintains the current date and time, making it available to programs. Once set, the time of day clock keeps time as accurately as a wristwatch.

Some systems solve the problem by asking a programmer to type in the time and date when the system boots. The system increments the clock periodically (e.g., every second). When an application program asks for the date or time, the system consults the internal clock and formats the time of day in human readable form. Client-server interaction can be used to set the system clock automatically when a machine boots. To do so, a manager configures one machine, typically the machine with the most accurate clock, to run a time-of-day server. When other machines boot, they contact the server to obtain the current time.

21.4.1 Representation for the Date and Time

How should an operating system maintain the date and time-of-day? One useful representation stores the time and date as the count of seconds since an epoch date. For example, the UNIX operating system uses the zeroth second of January 1, 1970 as its epoch date. The TCP/IP protocols also define an epoch date and report times as seconds past the epoch. For TCP/IP, the epoch is defined to be the zeroth second of January 1, 1900 and the time is kept in a 32-bit integer, a representation that accommodates all dates in the near future.

Keeping the date as the time in seconds since an epoch makes the representation compact and allows easy comparison. It ties together the date and time of day and makes it possible to measure time by incrementing a single binary integer.

21.4.2 Local and Universal Time

Given an epoch date and representation for the time, to what time zone does the count refer? When two systems communicate across large geographic distances, using the local time zone from one or the other becomes difficult; they must agree on a standard time zone to keep values for date and time comparable. Thus, in addition to defining a representation for the date and choosing an epoch, the TCP/IP time server standard specifies that all values are given with respect to a single time zone. Originally called Greenwich Mean Time, the time zone is now known as *universal coordinated time* or *universal time*.

The interaction between a client and a server that offers time service works much like an echo server. At the server side, the server application obtains permission to use the reserved port assigned to time servers, waits for a UDP message directed to that port, and responds by sending a UDP message that contains the current time in a 32-bit integer. We can summarize:

Sending a datagram to a time server is equivalent to making a request for the current time; the server responds by returning a UDP message that contains the current time.

21.5 The Complexity of Servers

In our examples so far, servers are fairly simple because they are sequential. That is, the server processes one request at a time. After accepting a request, the server forms a reply and sends it before going back to see if another request has arrived. We implicitly assume that the operating system will queue requests that arrive for a server while it is busy, and that the queue will not become too long because the server has only a trivial amount of work to do.

In practice, servers are usually much more difficult to build than clients because they need to accommodate multiple concurrent requests, even if a single request takes considerable time to process. For example, consider a file transfer server responsible for copying a file to another machine on request. Typically, servers have two parts: a single master program that is responsible for accepting new requests, and a set of slaves that are responsible for handling individual requests. The master server performs the following five steps:

Open port
The master opens the well-known port at which it can be reached.

Wait for client
The master waits for a new client to send a request.

Choose port
If necessary, the master allocates a new local protocol port for this request and informs the client (we will see that this step is unnecessary with TCP and most uses of UDP).

Start Slave
The master starts an independent, concurrent slave to handle this request (e.g., in UNIX, it forks a copy of the server process). Note that the slave handles one request and then terminates — the slave does not wait for requests from other clients.

Continue
The master returns to the *wait* step and continues accepting new requests while the newly created slave handles the previous request concurrently.

Because the master starts a slave for each new request, processing proceeds concurrently. Thus, requests that require little time to complete can finish earlier than requests that take longer, independent of the order in which they are started. For example, suppose the first client that contacts a file server requests a large file transfer that takes many minutes. If a second client contacts the server to request a transfer that takes only a few seconds, the second transfer can start and complete while the first transfer proceeds.

In addition to the complexity that results because servers handle concurrent requests, complexity also arises because servers must enforce authorization and protection rules. Server programs usually need to execute with highest privilege because they must read system files, keep logs, and access protected data. The operating system will not restrict a server program if it attempts to access users' files. Thus, servers cannot blindly honor requests from other sites. Instead, each server takes responsibility for enforcing the system access and protection policies.

Finally, servers must protect themselves against malformed requests or against requests that will cause the server program itself to abort. Often, it is difficult to foresee potential problems. For example, one project at Purdue University designed a file server that allowed student operating systems to access files on a UNIX timesharing system. Students discovered that requesting the server to open a file named */dev/tty* caused the server to abort because UNIX associates that name with the control terminal to which a program is attached. The server, created at system startup, had no such terminal. Once an abort occurred, no client could access files until a systems programmer restarted the server.

A more serious example of server vulnerability became known in the fall of 1988 when a student at Cornell University built a *worm* program that attacked computers on the global Internet. Once the worm started running on a machine, it searched the Internet for computers with servers that it knew how to exploit, and used the servers to create more copies of itself. In one of the attacks, the worm used a bug in the UNIX *fingerd* server. Because the server did not check incoming requests, the worm was able to send an illegal string of input that caused the server to overwrite parts of its internal data areas. The server, which executed with highest privilege, then misbehaved, allowing the worm to create copies of itself.

We can summarize our discussion of servers:

Servers are usually more difficult to build than clients because, although they can be implemented with application programs, servers must enforce all the access and protection policies of the computer system on which they run, and must protect themselves against all possible errors.

21.6 RARP Server

So far, all our examples of client-server interaction require the client to know the complete server address. The RARP protocol from Chapter 6 provides an example of client-server interaction with a slightly different twist. Recall that a machine can use RARP to find its IP address at startup. Instead of having the client communicate directly with a server, RARP clients broadcast their requests. One or more machines executing RARP server processes respond, each returning a packet that answers the query.

There are two significant differences between a RARP server and a UDP echo or time server. First, RARP packets travel across the physical network directly in hardware frames, not in IP datagrams. Thus, unlike the UDP echo server which allows a client to contact a server anywhere on an internet, the RARP server requires the client to be on the same physical network. Second, RARP cannot be implemented by an application program. Echo and time servers can be built as application programs because they use UDP. By contrast, a RARP server needs access to raw hardware packets.

21.7 Alternatives To The Client-Server Model

What are the alternatives to client-server interaction, and when might they be attractive? This section gives an answer to these questions.

In the client-server model, programs usually act as clients when they need information, but it is sometimes important to minimize such interactions. The ARP protocol from Chapter 5 gives one example. It uses a modified form of client-server interaction to obtain physical address mappings. Machines that use ARP keep a cache of answers to improve the efficiency of later queries. Caching improves the performance of client-server interaction in cases where the recent history of queries is a good indicator of future use.

Although caching improves performance, it does not change the essence of client-server interaction. The essence lies in our assumption that processing must be driven by demand. We have assumed that a program executes until it needs information and then acts as a client to obtain the needed information. Taking a demand-driven view of the world is natural and arises from experience. Caching helps alleviate the cost of obtaining information by lowering the retrieval cost for all except the first process that makes a request.

How can we lower the cost of information retrieval for the first request? In a distributed system, it may be possible to have concurrent background activities that collect and propagate information *before* any particular program requests it, making retrieval costs low even for the initial request. More important, precollecting information can allow a given system to continue executing even though other machines or the networks connecting them fail.

Precollection is the basis for the 4BSD UNIX *ruptime* command. When invoked, *ruptime* reports the CPU load and time since system startup for each machine on the local network. A background program running on each machine uses UDP to broadcast information about the machine periodically. The same program also collects incoming information and places it in a file. Because machines propagate information continuously, each machine has a copy of the latest information on hand; a client seeking information never needs to access the network. Instead, it reads the information from secondary storage and prints it in a readable form.

The chief advantage of having information collected locally before the client needs it is speed. The *ruptime* command responds immediately when invoked without waiting for messages to traverse the network. A second benefit occurs because the client can find out something about machines that are no longer operating. In particular, if a machine stops broadcasting information, the client can report the time elapsed since the last broadcast (i.e., it can report how long the machine has been off-line).

Precollection has one major disadvantage: it uses processor time and network bandwidth even when no one cares about the data being collected. For example, the ruptime broadcast and collection continues running throughout the night, even if no one is logged in to read the information. If only a few machines connect to a given network, precollection cost is insignificant. It can be thought of as an innocuous background activity. For networks with many hosts, however, the large volume of broadcast traffic generated by precollection makes it too expensive. In particular, the cost of reading and processing broadcast messages becomes high. Thus, precollection is not among the most popular alternatives to client-server.

21.8 Summary

Distributed programs require network communication. Such programs often fall into a pattern of use known as client-server interaction. A server process awaits a request and performs action based on the request. The action usually includes sending a response. A client program formulates a request, sends it to a server, and then awaits a reply.

We have seen examples of clients and servers and found that some clients send requests directly, while others broadcast requests. Broadcast is especially useful on a local network when a machine does not know the address of a server.

We also noted that if servers use internet protocols like UDP, they can accept and respond to requests across an internet. If they communicate using physical frames and physical hardware addresses, they are restricted to a single physical network.

Finally, we considered an alternative to the client-server paradigm that uses precollection of information to avoid delays. An example of precollection came from a machine status service.

FOR FURTHER STUDY

UDP echo service is defined in Postel [RFC 862]. The *UNIX Programmer's Manual* describes the *ruptime* command (also see the related description of *rwho*). Feinler *et. al.* [1985] specifies many standard server protocols not discussed here, including discard, character generation, day and time, active users, and quote of the day. The next chapters consider others.

EXERCISES

21.1 Build a UDP echo client that sends a datagram to a specified echo server, awaits a reply, and compares it to the original message.

21.2 Carefully consider the manipulation of IP addresses in a UDP echo server. Under what conditions is it incorrect to create new IP addresses by reversing the source and destination IP addresses?

21.3 As we have seen, servers can be implemented by separate application programs or by building server code into the protocol software in an operating system. What are the advantages and disadvantages of having an application program (user process) per server?

21.4 Suppose you do not know the IP address of a local machine running a UDP echo server, but you know that it responds to requests sent to port *7*. Is there an IP address you can use to reach it?

21.5 Build a client for the UDP time service.

21.6 Characterize situations in which a server can be located on a separate physical network from its client. Can a RARP server ever be located on a separate physical network from it clients? Why or why not?

21.7 What is the chief disadvantage of having all machines broadcast their status periodically?

21.8 Examine the format of data broadcast by the servers that implement the 4BSD UNIX *ruptime* command. What information is available to the client in addition to machine status?

21.9 What servers are running on computers at your site? If you do not have access to system configuration files that list the servers started for a given computer, see if your system has a command that prints a list of open TCP and UDP ports (e.g., the UNIX *netstat* command).

21.10 Some servers allow a manager to gracefully shut them down or restart them. What is the advantage of graceful shutdown?

22

The Socket Interface

22.1 Introduction

So far, we have concentrated on discussing the principles and concepts that underlie the TCP/IP protocols without specifying the interface between the application programs and the protocol software. This chapter reviews one example of an *Application Program Interface* (*API*), the interface between application programs and TCP/IP protocols. There are two reasons for postponing the discussion of APIs. First, in principle we must distinguish between the interface and TCP/IP protocols because the standards do not specify exactly how application programs interact with protocol software. Thus, the interface architecture is not standardized; its design lies outside the scope of the protocol suite. Second, in practice, it is inappropriate to tie the protocols to a particular API because no single interface architecture works well on all systems. In particular, because protocol software resides in a computer's operating system, interface details depend on the operating system.

Despite the lack of a standard, reviewing an example will help us understand how programmers use TCP/IP. Although the example we have chosen is from the BSD UNIX operating system, it has become, de facto, a standard that is widely accepted and used in many systems. In particular, it forms the basis for Microsoft's *Windows Sockets*† interface. The reader should keep in mind that our goal is merely to give one concrete example, not to prescribe how APIs should be designed. The reader should also remember that the operations listed here are not part of the TCP/IP standards.

†Programmers often use the term *WINSOCK* as a replacement for Windows Sockets.

22.2 The UNIX I/O Paradigm And Network I/O

Developed in the late 1960s and early 1970s, UNIX was originally designed as a timesharing system for single processor computers. It is a process-oriented operating system in which each application program executes as a user level process. An application program interacts with the operating system by making *system calls*. From the programmer's point of view, system calls look and behave exactly like other procedure calls. They take arguments and return one or more results. Arguments can be values (e.g., an integer count) or pointers to objects in the application program (e.g., a buffer to be filled with characters).

Derived from those in Multics and earlier systems, the UNIX input and output (I/O) primitives follow a paradigm sometimes referred to as *open-read-write-close*. Before a user process can perform I/O operations, it calls *open* to specify the file or device to be used and obtains permission. The call to *open* returns a small integer *file descriptor*† that the process uses when performing I/O operations on the opened file or device. Once an object has been opened, the user process makes one or more calls to *read* or *write* to transfer data. *Read* transfers data into the user process; *write* transfers data from the user process to the file or device. Both *read* and *write* take three arguments that specify the file descriptor to use, the address of a buffer, and the number of bytes to transfer. After all transfer operations are complete, the user process calls *close* to inform the operating system that it has finished using the object (the operating system automatically closes all open descriptors if a process terminates without calling *close*).

22.3 Adding Network I/O to UNIX

Originally, UNIX designers cast all I/O operations in the open-read-write-close paradigm described above. The scheme included I/O for character-oriented devices like keyboards and block-oriented devices like disks and data files. An early implementation of TCP/IP under UNIX also used the open-read-write-close paradigm with a special file name, */dev/tcp*.

The group adding network protocols to BSD UNIX decided that because network protocols are more complex than conventional I/O devices, interaction between user processes and network protocols must be more complex than interactions between user processes and conventional I/O facilities. In particular, the protocol interface must allow programmers to create both server code that awaits connections passively as well as client code that forms connections actively. Furthermore, application programs sending datagrams may wish to specify the destination address along with each datagram instead of binding destinations at the time they call *open*. To handle all these cases, the designers chose to abandon the traditional UNIX open-read-write-close paradigm, and added several new operating system calls as well as new library routines. Adding network protocols to UNIX increased the complexity of the I/O interface substantially.

Further complexity arises in the UNIX protocol interface because designers attempted to build a general mechanism to accommodate many protocols. For example,

†The term "file descriptor" arises because in UNIX all devices are mapped into the file system name space. In most cases, I/O operations on files and devices are indistinguishable.

the generality makes it possible for the operating system to include software for other protocol suites as well as TCP/IP, and to allow an application program to use one or more of them at a time. As a consequence, the application program cannot merely supply a 32-bit address and expect the operating system to interpret it correctly. The application must explicitly specify that the 32-bit number represents an IP address.

22.4 The Socket Abstraction

The basis for network I/O in the socket API centers on an abstraction known as the *socket*†. We think of a socket as a generalization of the UNIX file access mechanism that provides an endpoint for communication. As with file access, application programs request the operating system to create a socket when one is needed. The system returns a small integer that the application program uses to reference the newly created socket. The chief difference between file descriptors and socket descriptors is that the operating system binds a file descriptor to a specific file or device when the application calls *open*, but it can create sockets without binding them to specific destination addresses. The application can choose to supply a destination address each time it uses the socket (e.g., when sending datagrams), or it can choose to bind the destination address to the socket and avoid specifying the destination repeatedly (e.g., when making a TCP connection).

Whenever it makes sense, sockets perform exactly like UNIX files or devices, so they can be used with traditional operations like *read* and *write*. For example, once an application program creates a socket and creates a TCP connection from the socket to a foreign destination, the program can use *write* to send a stream of data across the connection (the application program at the other end can use *read* to receive it). To make it possible to use primitives like *read* and *write* with both files and sockets, the operating system allocates socket descriptors and file descriptors from the same set of integers and makes sure that if a given integer has been allocated as a file descriptor, it will not also be allocated as a socket descriptor.

22.5 Creating A Socket

The *socket* function creates sockets on demand. It takes three integer arguments and returns an integer result:

result = socket(pf, type, protocol)

Argument *pf* specifies the protocol family to be used with the socket. That is, it specifies how to interpret addresses when they are supplied. Current families include the TCP/IP internet (PF_INET), Xerox Corporation PUP internet (PF_PUP), Apple Computer Incorporated AppleTalk network (PF_APPLETALK), and UNIX file system (PF_UNIX) as well as many others‡.

†For now, we will describe sockets as part of the operating system as they are implemented in UNIX; later sections describe how other operating systems use library routines to provide a socket API.

‡In UNIX, application programs contain symbolic names like *PF_INET*; system files contain the definitions that specify numeric values for each name.

Argument *type* specifies the type of communication desired. Possible types include reliable stream delivery service (SOCK_STREAM) and connectionless datagram delivery service (SOCK_DGRAM), as well as a raw type (SOCK_RAW) that allows privileged programs to access low-level protocols or network interfaces. Two additional types were planned, but not implemented.

Although the general approach of separating protocol families and types may seem sufficient to handle all cases easily, it does not. First, it may be that a given family of protocols does not support one or more of the possible service types. For example, the UNIX family has an interprocess communication mechanism called a *pipe* that uses a reliable stream delivery service, but has no mechanism for sequenced packet delivery. Thus, not all combinations of protocol family and service type make sense. Second, some protocol families have multiple protocols that support one type of service. For example, it may be that a single protocol family has two connectionless datagram delivery services. To accommodate multiple protocols within a family, the *socket* call has a third argument that can be used to select a specific protocol. To use the third argument, the programmer must understand the protocol family well enough to know the type of service each protocol supplies.

Because the designers tried to capture many of the conventional UNIX operations in their socket design, they needed a way to simulate the UNIX pipe mechanism. It is not necessary to understand the details of pipes; only one salient feature is important: pipes differ from standard network operations because the calling process creates both endpoints for the communication simultaneously. To accommodate pipes, the designers added a *socketpair* function that takes the form:

```
socketpair(pf, type, protocol, sarray)
```

Socketpair has one more argument than the *socket* procedure, *sarray*. The additional argument gives the address of a two-element integer array. *Socketpair* creates two sockets simultaneously and places the two socket descriptors in the two elements of *sarray*. Readers should understand that *socketpair* is not meaningful when applied to the TCP/IP protocol family (it has been included here merely to make our description of the interface complete).

22.6 Socket Inheritance And Termination

UNIX uses the *fork* and *exec* system calls to start new application programs. It is a two-step procedure. In the first step, *fork* creates a separate copy of the currently executing application program. In the second step, the new copy replaces itself with the desired application program. When a program calls *fork*, the newly created copy inherits access to all open sockets just as it inherits access to all open files. When a program calls *exec*, the new application retains access to all open sockets. We will see that master servers use socket inheritance when they create slave servers to handle a specific connection. Internally, the operating system keeps a reference count associated with each socket, so it knows how many application programs (processes) have access to it.

Both the old and new processes have the same access rights to existing sockets, and both can access the sockets. Thus, it is the responsibility of the programmer to ensure that the two processes use the shared socket meaningfully.

When a process finishes using a socket it calls *close*. *Close* has the form:

close(socket)

where argument *socket* specifies the descriptor of a socket to close. When a process terminates for any reason, the system closes all sockets that remain open. Internally, a call to *close* decrements the reference count for a socket and destroys the socket if the count reaches zero.

22.7 Specifying A Local Address

Initially, a socket is created without any association to local or destination addresses. For the TCP/IP protocols, this means no local protocol port number has been assigned and no destination port or IP address has been specified. In many cases, application programs do not care about the local address they use and are willing to allow the protocol software to choose one for them. However, server processes that operate at a well-known port must be able to specify that port to the system. Once a socket has been created, a server uses the *bind* function to establish a local address for it. *Bind* has the following form:

bind(socket, localaddr, addrlen)

Argument *socket* is the integer descriptor of the socket to be bound. Argument *localaddr* is a structure that specifies the local address to which the socket should be bound, and argument *addrlen* is an integer that specifies the length of the address measured in bytes. Instead of giving the address merely as a sequence of bytes, the designers chose to use a structure for addresses as Figure 22.1 illustrates.

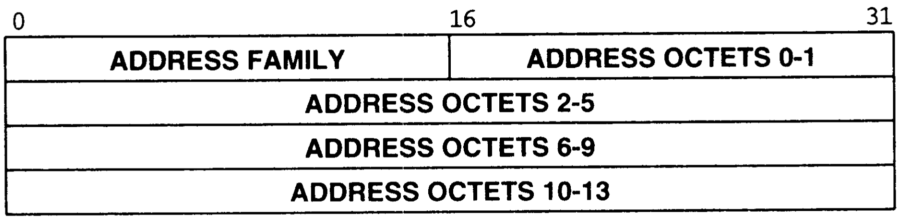


Figure 22.1 The *sockaddr* structure used when passing a TCP/IP address to the socket interface.

The structure, generically named *sockaddr*, begins with a 16-bit *ADDRESS FAMILY* field that identifies the protocol suite to which the address belongs. It is followed by an address of up to *14* octets. When declared in C, the socket address structure is a union of structures for all possible address families.

The value in the *ADDRESS FAMILY* field determines the format of the remaining address octets. For example, the value *2*† in the *ADDRESS FAMILY* field means the remaining address octets contain a TCP/IP address. Each protocol family defines how it will use octets in the address field. For TCP/IP addresses, the socket address is known as *sockaddr_in*. It includes both an IP address and a protocol port number (i.e., an internet socket address structure can contain both an IP address and a protocol port at that address). Figure 22.2 shows the exact format of a TCP/IP socket address.

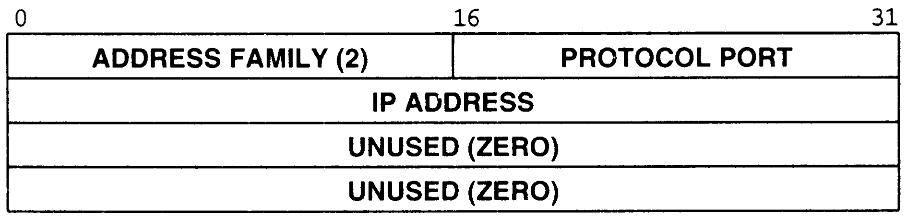


Figure 22.2 The format of a socket address structure (*sockaddr_in*) when used with a TCP/IP address. The structure includes both an IP address and a protocol port at that address.

Although it is possible to specify arbitrary values in the address structure when calling *bind*, not all possible bindings are valid. For example, the caller might request a local protocol port that is already in use by another program, or it might request an invalid IP address. In such cases, the *bind* call fails and returns an error code.

22.8 Connecting Sockets To Destination Addresses

Initially, a socket is created in the *unconnected state*, which means that the socket is not associated with any foreign destination. The function *connect* binds a permanent destination to a socket, placing it in the *connected state*. An application program must call *connect* to establish a connection before it can transfer data through a reliable stream socket. Sockets used with connectionless datagram services need not be connected before they are used, but doing so makes it possible to transfer data without specifying the destination each time.

The *connect* function has the form:

connect(socket, destaddr, addrlen)

†UNIX uses the symbolic name *PF_INET* to denote TCP/IP addresses.

Argument *socket* is the integer descriptor of the socket to connect. Argument *destaddr* is a socket address structure that specifies the destination address to which the socket should be bound. Argument *addrlen* specifies the length of the destination address measured in bytes.

The semantics of *connect* depend on the underlying protocols. Selecting the reliable stream delivery service in the PF_INET family means choosing TCP. In such cases, *connect* builds a TCP connection with the destination and returns an error if it cannot. In the case of connectionless service, *connect* does nothing more than store the destination address locally.

22.9 Sending Data Through A Socket

Once an application program has established a socket, it can use the socket to transmit data. There are five possible functions from which to choose: *send*, *sendto*, *sendmsg*, *write*, and *writev*. *Send*, *write*, and *writev* only work with connected sockets because they do not allow the caller to specify a destination address. The differences between the three are minor. *Write* takes three arguments:

write(socket, buffer, length)

Argument *socket* contains an integer socket descriptor (*write* can also be used with other types of descriptors). Argument *buffer* contains the address of the data to be sent, and argument *length* specifies the number of bytes to send. The call to *write* blocks until the data can be transferred (e.g., it blocks if internal system buffers for the socket are full). Like most system calls, *write* returns an error code to the application calling it, allowing the programmer to know if the operation succeeded.

The system call *writev* works like *write* except that it uses a ''gather write'' form, making it possible for the application program to write a message without copying the message into contiguous bytes of memory. *Writev* has the form:

writev(socket, iovector, vectorlen)

Argument *iovector* gives the address of an array of type *iovec* that contains a sequence of pointers to the blocks of bytes that form the message. As Figure 22.3 shows, a length accompanies each pointer. Argument *vectorlen* specifies the number of entries in *iovector*.

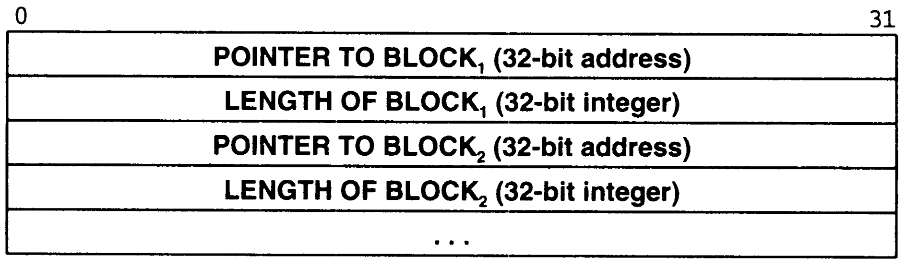

0
31
POINTER TO BLOCK$_1$ (32-bit address)
LENGTH OF BLOCK$_1$ (32-bit integer)
POINTER TO BLOCK$_2$ (32-bit address)
LENGTH OF BLOCK$_2$ (32-bit integer)
. . .

Figure 22.3 The format of an iovector of type *iovec* used with *writev* and *readv*.

The *send* function has the form:

send(socket, message, length, flags)

where argument *socket* specifies the socket to use, argument *message* gives the address of the data to be sent, argument *length* specifies the number of bytes to be sent, and argument *flags* controls the transmission. One value for *flags* allows the sender to specify that the message should be sent out-of-band on sockets that support such a notion. For example, recall from Chapter 13 that out-of-band messages correspond to TCP's notion of urgent data. Another value for *flags* allows the caller to request that the message be sent without using local routing tables. The intention is to allow the caller to take control of routing, making it possible to write network debugging software. Of course, not all sockets support all requests from arbitrary programs. Some requests require the program to have special privileges; others are simply not supported on all sockets.

Functions *sendto* and *sendmsg* allow the caller to send a message through an unconnected socket because they both require the caller to specify a destination. *Sendto*, which takes the destination address as an argument, has the form:

sendto(socket, message, length, flags, destaddr, addrlen)

The first four arguments are exactly the same as those used with the *send* function. The final two arguments specify a destination address and give the length of that address. Argument *destaddr* specifies the destination address using the *sockaddr_in* structure as defined in Figure 22.2.

A programmer may choose to use function *sendmsg* in cases where the long list of arguments required for *sendto* makes the program inefficient or difficult to read. *Sendmsg* has the form:

sendmsg(socket, messagestruct, flags)

where argument *messagestruct* is a structure of the form illustrated in Figure 22.4. The structure contains information about the message to be sent, its length, the destination address, and the address length. This call is especially useful because there is a corresponding input operation (described below) that produces a message structure in exactly the same format.

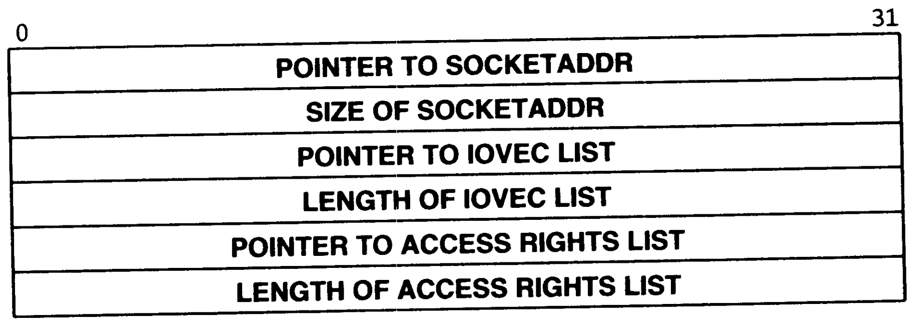


Figure 22.4 The format of message structure *messagestruct* used by *sendmsg*.

22.10 Receiving Data Through A Socket

Analogous to the five different output operations, the socket API offers five functions that a process can use to receive data through a socket: *read*, *readv*, *recv*, *recvfrom*, and *recvmsg*. The conventional input operation, *read*, can only be used when the socket is connected. It has the form:

read(descriptor, buffer, length)

where *descriptor* gives the integer descriptor of a socket or file descriptor from which to read data, *buffer* specifies the address in memory at which to store the data, and *length* specifies the maximum number of bytes to read.

An alternative form, *readv*, allows the caller to use a "scatter read" style of interface that places the incoming data in noncontiguous locations. *Readv* has the form:

readv(descriptor, iovector, vectorlen)

Argument *iovector* gives the address of a structure of type *iovec* (see Figure 22.3) that contains a sequence of pointers to blocks of memory into which the incoming data should be stored. Argument *vectorlen* specifies the number of entries in *iovector*.

In addition to the conventional input operations, there are three additional functions for network message input. Processes call *recv* to receive data from a connected socket. It has the form:

recv(socket, buffer, length, flags)

Argument *socket* specifies a socket descriptor from which data should be received. Argument *buffer* specifies the address in memory into which the message should be placed, and argument *length* specifies the length of the buffer area. Finally, argument *flags* allows the caller to control the reception. Among the possible values for the *flags* argument is one that allows the caller to look ahead by extracting a copy of the next incoming message without removing the message from the socket.

The function *recvfrom* allows the caller to specify input from an unconnected socket. It includes additional arguments that allow the caller to specify where to record the sender's address. The form is:

recvfrom(socket, buffer, length, flags, fromaddr, addrlen)

The two additional arguments, *fromaddr* and *addrlen*, are pointers to a socket address structure and an integer. The operating system uses *fromaddr* to record the address of the message sender and uses *fromlen* to record the length of the sender's address. Notice that the output operation *sendto*, discussed above, takes an address in exactly the same form as *recvfrom* generates. Thus, sending replies is easy.

The final function used for input, *recvmsg*, is analogous to the *sendmsg* output operation. *Recvmsg* operates like *recvfrom*, but requires fewer arguments. Its form is:

recvmsg(socket, messagestruct, flags)

where argument *messagestruct* gives the address of a structure that holds the address for an incoming message as well as locations for the sender's address. The structure produced by *recvmsg* is exactly the same as the structure used by *sendmsg*, making them operate well as a pair.

22.11 Obtaining Local And Remote Socket Addresses

We said that newly created processes inherit the set of open sockets from the process that created them. Sometimes, a newly created process needs to determine the destination address to which a socket connects. A process may also wish to determine the local address of a socket. Two functions provide such information: *getpeername* and *getsockname* (despite their names, both deal with what we think of as ''addresses'').

A process calls *getpeername* to determine the address of the peer (i.e., the remote end) to which a socket connects. It has the form:

getpeername(socket, destaddr, addrlen)

Argument *socket* specifies the socket for which the address is desired. Argument *destaddr* is a pointer to a structure of type *sockaddr* (see Figure 22.1) that will receive the socket address. Finally, argument *addrlen* is a pointer to an integer that will receive the length of the address. *Getpeername* only works with connected sockets.

Function *getsockname* returns the local address associated with a socket. It has the form:

getsockname(socket, localaddr, addrlen)

As expected, argument *socket* specifies the socket for which the local address is desired. Argument *localaddr* is a pointer to a structure of type *sockaddr* that will contain the address, and argument *addrlen* is a pointer to an integer that will contain the length of the address.

22.12 Obtaining And Setting Socket Options

In addition to binding a socket to a local address or connecting it to a destination address, the need arises for a mechanism that permits application programs to control the socket. For example, when using protocols that use timeout and retransmission, the application program may want to obtain or set the timeout parameters. It may also want to control the allocation of buffer space, determine if the socket allows transmission of broadcast, or control processing of out-of-band data. Rather than add new functions for each new control operation, the designers decided to build a single mechanism. The mechanism has two operations: *getsockopt* and *setsockopt*.

Function *getsockopt* allows the application to request information about the socket. A caller specifies the socket, the option of interest, and a location at which to store the requested information. The operating system examines its internal data structures for the socket and passes the requested information to the caller. The call has the form:

getsockopt(socket, level, optionid, optionval, length)

Argument *socket* specifies the socket for which information is needed. Argument *level* identifies whether the operation applies to the socket itself or to the underlying protocols being used. Argument *optionid* specifies a single option to which the request applies. The pair of arguments *optionval* and *length* specify two pointers. The first gives the address of a buffer into which the system places the requested value, and the second gives the address of an integer into which the system places the length of the option value.

Function *setsockopt* allows an application program to set a socket option using the set of values obtained with *getsockopt*. The caller specifies a socket for which the option should be set, the option to be changed, and a value for the option. The call to *setsockopt* has the form:

setsockopt(socket, level, optionid, optionval, length)

where the arguments are like those for *getsockopt*, except that the *length* argument contains the length of the option being passed to the system. The caller must supply a legal value for the option as well as a correct length for that value. Of course, not all options apply to all sockets. The correctness and semantics of individual requests depend on the current state of the socket and the underlying protocols being used.

22.13 Specifying A Queue Length For A Server

One of the options that applies to sockets is used so frequently, a separate function has been dedicated to it. To understand how it arises, consider a server. The server creates a socket, binds it to a well-known protocol port, and waits for requests. If the server uses a reliable stream delivery, or if computing a response takes nontrivial amounts of time, it may happen that a new request arrives before the server finishes responding to an old request. To avoid having protocols reject or discard incoming requests, a server must tell the underlying protocol software that it wishes to have such requests enqueued until it has time to process them.

The function *listen* allows servers to prepare a socket for incoming connections. In terms of the underlying protocols, *listen* puts the socket in a passive mode ready to accept connections. When the server invokes *listen*, it also informs the operating system that the protocol software should enqueue multiple simultaneous requests that arrive at the socket. The form is:

listen(socket, qlength)

Argument *socket* gives the descriptor of a socket that should be prepared for use by a server, and argument *qlength* specifies the length of the request queue for that socket. After the call, the system will enqueue up to *qlength* requests for connections. If the queue is full when a request arrives, the operating system will refuse the connection by discarding the request. *Listen* applies only to sockets that have selected reliable stream delivery service.

22.14 How A Server Accepts Connections

As we have seen, a server process uses the functions *socket*, *bind*, and *listen* to create a socket, bind it to a well-known protocol port, and specify a queue length for connection requests. Note that the call to *bind* associates the socket with a well-known protocol port, but that the socket is not connected to a specific foreign destination. In fact, the foreign destination must specify a *wildcard*, allowing the socket to receive connection requests from an arbitrary client.

Once a socket has been established, the server needs to wait for a connection. To do so, it uses function *accept*. A call to *accept* blocks until a connection request arrives. It has the form:

newsock = accept(socket, addr, addrlen)