10

Dynamic Routing Protocols

10.1 Introduction

Our discussion in the previous chapter dealt with *static routing*. The routing table entries were created by default when an interface was configured (for directly connected interfaces), added by the `route` command (normally from a system bootstrap file), or created by an ICMP redirect (usually when the wrong default was used).

This is fine if the network is small, there is a single connection point to other networks, and there are no redundant routes (where a backup route can be used if a primary route fails). If any of these three conditions is false, dynamic routing is normally used.

This chapter looks at the dynamic routing protocols used by routers to communicate with each other. We concentrate on RIP, the Routing Information Protocol, a widely used protocol that is provided with almost every TCP/IP implementation. We then look at two newer routing protocols, OSPF and BGP. The chapter finishes with an examination of a new routing technique, called classless interdomain routing, that is starting to be implemented across the Internet to converse class B network numbers.

10.2 Dynamic Routing

Dynamic routing occurs when routers talk to adjacent routers, informing each other of what networks each router is currently connected to. The routers must communicate using a *routing protocol*, of which there are many to choose from. The process on the router that is running the routing protocol, communicating with its neighbor routers, is usually called a *routing daemon*. As shown in Figure 9.1, the routing daemon updates the kernel's routing table with information it receives from neighbor routers.

The use of dynamic routing does *not* change the way the kernel performs routing at the IP layer, as we described in Section 9.2. We called this the *routing mechanism*. The kernel still searches its routing table in the same way, looking for host routes, network routes, and default routes. What changes is the information placed into the routing table—instead of coming from `route` commands in bootstrap files, the routes are added and deleted dynamically by a routing daemon, as routes change over time.

As we mentioned earlier, the routing daemon adds a *routing policy* to the system, choosing which routes to place into the kernel's routing table. If the daemon finds multiple routes to a destination, the daemon chooses (somehow) which route is best, and which one to insert into the kernel's table. If the daemon finds that a link has gone down (perhaps a router crashed or a phone line is out of order), it can delete the affected routes or add alternate routes that bypass the problem.

In a system such as the Internet, many different routing protocols are currently used. The Internet is organized into a collection of *autonomous systems* (ASs), each of which is normally administered by a single entity. A corporation or university campus often defines an autonomous system. The NSFNET backbone of the Internet forms an autonomous system, because all the routers in the backbone are under a single administrative control.

Each autonomous system can select its own routing protocol to communicate between the routers in that autonomous system. This is called an *interior gateway protocol* (IGP) or *intradomain routing protocol*. The most popular IGP has been the *Routing Information Protocol* (RIP). A newer IGP is the *Open Shortest Path First* protocol (OSPF). It is intended as a replacement for RIP. An older IGP that has fallen out of use is HELLO—the IGP used on the original NSFNET backbone in 1986.

> The new Router Requirements RFC [Almquist 1993] states that a router that implements any dynamic routing protocol must support both OSPF and RIP, and may support other IGPs.

Separate routing protocols called *exterior gateway protocols* (EGPs) or *interdomain routing protocols* are used between the routers in different autonomous systems. Historically (and confusingly) the predominant EGP has been a protocol of the same name: EGP. A newer EGP is the Border Gateway Protocol (BGP) that is currently used between the NSFNET backbone and some of the regional networks that attach to the backbone. BGP is intended to replace EGP.

10.3 Unix Routing Daemons

Unix systems often run the routing daemon named `routed`. It is provided with almost every implementation of TCP/IP. This program communicates using only RIP, which we describe in the next section. It is intended for small to medium-size networks.

An alternative program is `gated`. It supports both IGPs and EGPs. [Fedor 1988] describes the early development of `gated`. Figure 10.1 compares the various routing protocols supported by `routed` and two different versions of `gated`. Most systems that run a routing daemon run `routed`, unless they need support for the other protocols supported by `gated`.

Daemon	Interior Gateway Protocol			Exterior Gateway Protocol	
	HELLO	RIP	OSPF	EGP	BGP
`routed`		V1			
`gated`, Version 2	•	V1		•	V1
`gated`, Version 3	•	V1, V2	V2	•	V2, V3

Figure 10.1 Routing protocols supported by `routed` and `gated`.

We describe RIP Version 1 in the next section, the differences with RIP Version 2 in Section 10.5, OSPF in Section 10.6, and BGP in Section 10.7.

10.4 RIP: Routing Information Protocol

This section provides an overview of RIP, because it is the most widely used (and most often maligned) routing protocol. The official specification for RIP is RFC 1058 [Hedrick 1988a], but this RFC was written years after the protocol was widely implemented.

Message Format

RIP messages are carried in UDP datagrams, as shown in Figure 10.2. (We talk more about UDP in Chapter 11.)

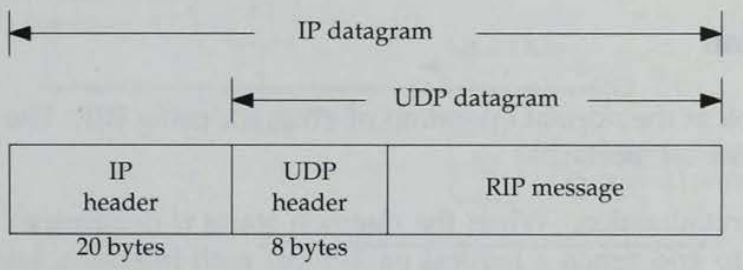


Figure 10.2 RIP message encapsulated within a UDP datagram.

Figure 10.3 shows the format of the RIP message, when used with IP addresses.

A *command* of 1 is a request, and 2 is a reply. There are two other obsolete commands (3 and 4), and two undocumented ones: poll (5) and poll-entry (6). A request asks the other system to send all or part of its routing table. A reply contains all or part of the sender's routing table.

The *version* is normally 1, although RIP Version 2 (Section 10.5) sets this to 2.

The next 20 bytes specify the *address family* (which is always 2 for IP addresses), an *IP address*, and an associated *metric*. We'll see later in this section that RIP metrics are hop counts.

Up to 25 routes can be advertised in a RIP message using this 20-byte format. The limit of 25 is to keep the total size of the RIP message, $20 \times 25 + 4 = 504$, less than 512 bytes. With this limit of 25 routes per message, multiple messages are often required to send an entire routing table.

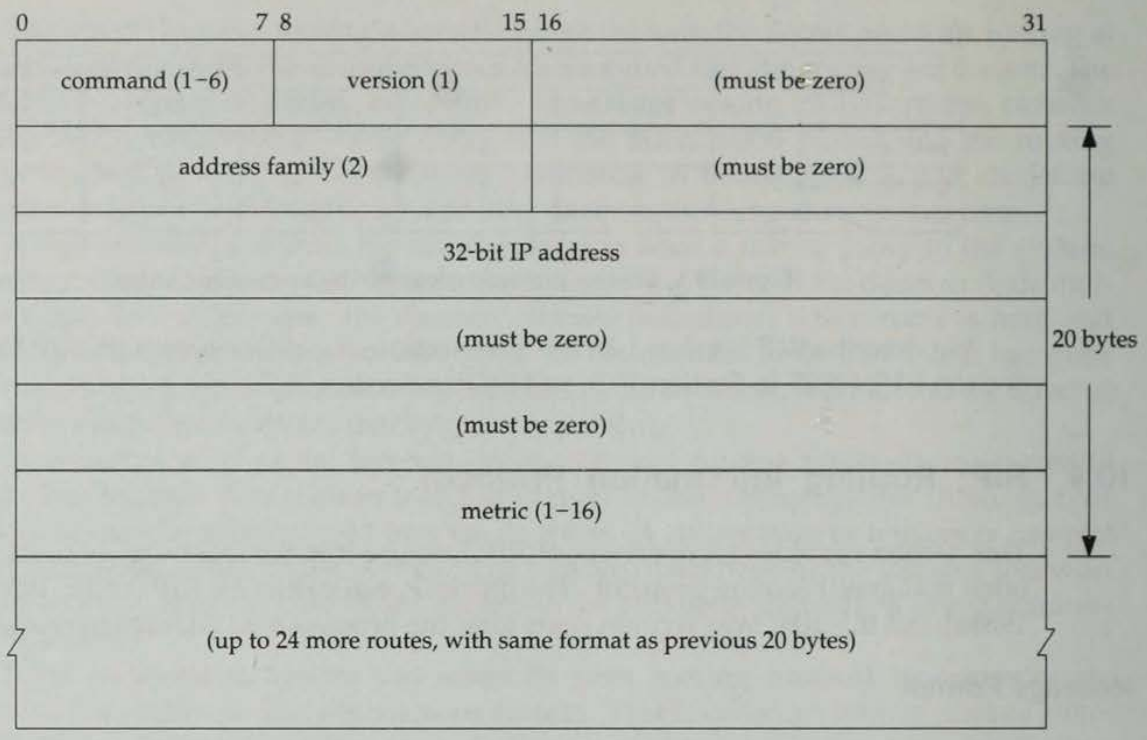


Figure 10.3 Format of a RIP message.

Normal Operation

Let's look at the normal operation of `routed`, using RIP. The well-known port number for RIP is UDP port 520.

- Initialization. When the daemon starts it determines all the interfaces that are up and sends a request packet out each interface, asking for the other router's complete routing table. On a point-to-point link this request is sent to the other end. The request is broadcast if the network supports it. The destination UDP port is 520 (the routing daemon on the other router).

  This request packet has a *command* of 1 but the *address family* is set to 0 and the *metric* is set to 16. This is a special request that asks for a complete routing table from the other end.

- Request received. If the request is the special case we just mentioned, then the entire routing table is sent to the requestor. Otherwise each entry in the request is processed: if we have a route to the specified address, set the metric to our value, else set the metric to 16. (A metric of 16 is a special value called "infinity" and means we don't have a route to that destination.) The response is returned.

- Response received. The response is validated and may update the routing table. New entries can be added, existing entries can be modified, or existing entries can be deleted.

- Regular routing updates. Every 30 seconds, all or part of the router's entire routing table is sent to every neighbor router. The routing table is either broadcast (e.g., on an Ethernet) or sent to the other end of a point-to-point link.
- Triggered updates. These occur whenever the metric for a route changes. The entire routing table need not be sent—only those entries that have changed must be transmitted.

Each route has a timeout associated with it. If a system running RIP finds a route that has not been updated for 3 minutes, that route's metric is set to infinity (16) and marked for deletion. This means we have missed six of the 30-second updates from the router that advertised that route. The deletion of the route from the local routing table is delayed for another 60 seconds to ensure the invalidation is propagated.

Metrics

The metrics used by RIP are hop counts. The hop count for all directly connected interfaces is 1. Consider the routers and networks shown in Figure 10.4. The four dashed lines we show are broadcast RIP messages.

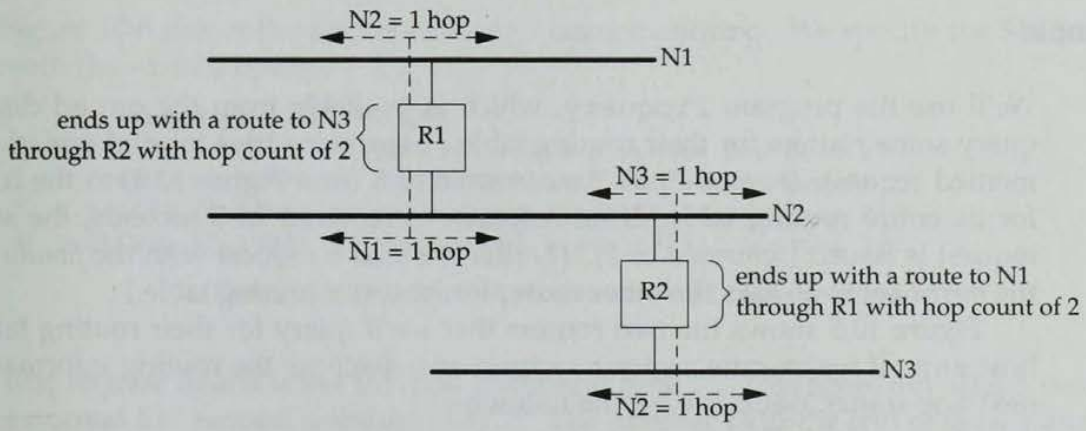


Figure 10.4 Example routers and networks.

Router R1 advertises a route to N2 with a hop count of 1 by sending a broadcast on N1. (It makes no sense to advertise a route to N1 in the broadcast sent on N1.) It also advertises a route to N1 with a hop count of 1 by sending a broadcast on N2. Similarly, R2 advertises a route to N2 with a metric of 1, and a route to N3 with a metric of 1.

If an adjacent router advertises a route to another network with a hop count of 1, then our metric for that network is 2, since we have to send a packet to that router to get to the network. In our example, the metric to N1 for R2 is 2, as is the metric to N3 for R1.

As each router sends its routing tables to its neighbors, a route can be determined to each network within the AS. If there are multiple paths within the AS from a router to a network, the router selects the path with the smallest hop count and ignores the other paths.

The hop count is limited to 15, meaning RIP can be used only within an AS where the maximum number of hops between hosts is 15. The special metric of 16 indicates that no route exists to the IP address.

Problems

As simple as this sounds, there are pitfalls. First, RIP has no knowledge of subnet addressing. If the normal 16-bit host ID of a class B address is nonzero, for example, RIP can't tell if the nonzero portion is a subnet ID or if the IP address is a complete host address. Some implementations use the subnet mask of the interface through which the RIP information arrived, which isn't always correct.

Next, RIP takes a long time to stabilize after the failure of a router or a link. The time is usually measured in minutes. During this settling time routing loops can occur. There are many subtle details in the implementation of RIP that must be followed to help prevent routing loops and to speed convergence. RFC 1058 [Hedrick 1988a] contains many details on how RIP should be implemented.

The use of the hop count as the routing metric omits other variables that should be taken into consideration. Also, a maximum of 15 for the metric limits the sizes of networks on which RIP can be used.

Example

We'll use the program `ripquery`, which is available from the `gated` distribution, to query some routers for their routing table. `ripquery` tries to send one of the undocumented requests (named "poll," a *command* of 5 from Figure 10.3) to the router, asking for its entire routing table. If no response is received in 5 seconds, the standard RIP request is issued (*command* of 1). (Earlier we said a request with the *family* set to 0 and the *metric* set to 16 asks the other router for its entire routing table.)

Figure 10.5 shows the two routers that we'll query for their routing table from the host `sun`. If we execute `ripquery` from `sun`, fetching the routing information from its next-hop router, `netb`, we get the following:

```
sun % ripquery -n netb
504 bytes from netb (140.252.1.183):      first message contains 504 bytes
                                          lots of other lines deleted
        140.252.1.0, metric 1             the top Ethernet in Figure 10.5
        140.252.13.0, metric 1            the bottom Ethernet in Figure 10.5
244 bytes from netb (140.252.1.183):      second message with remaining 244 bytes
                                          lots of other lines deleted
```

As we expect, the metric for our subnet that is announced by `netb` is 1. Additionally, the top Ethernet that `netb` is also attached to (140.252.1.0) has a metric of 1. (The `-n` flag says to print the IP addresses numerically instead of trying to look up the names.) In this example `netb` has been configured to consider all the hosts on the subnet 140.252.13 as directly connected to it—that is, `netb` knows nothing about which hosts are actually on the 140.252.13 subnet. Since there is only one connection point to the 140.252.13 subnet, advertising different metrics for each host makes little practical sense.

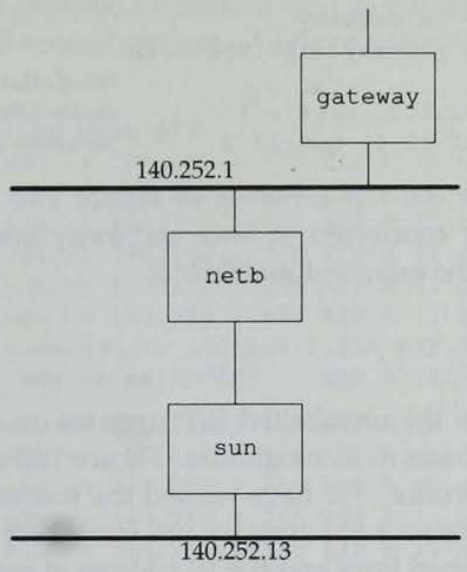


Figure 10.5 Two routers `netb` and `gateway` that we'll query for their routing tables.

Figure 10.6 shows the packet exchange using `tcpdump`. We specify the SLIP interface with the `-i sl0` option.

```
    sun % tcpdump -s600 -i sl0
1   0.0                 sun.2879 > netb.route: rip-poll 24
2   5.014702 (5.0147)   sun.2879 > netb.route: rip-req 24
3   5.560427 (0.5457)   netb.route > sun.2879: rip-resp 25:
4   5.710251 (0.1498)   netb.route > sun.2879: rip-resp 12:
```

Figure 10.6 `tcpdump` output while running `ripquery` program.

The first request issued is the RIP poll command (line 1). This times out after 5 seconds and a normal RIP request is issued (line 2). The number 24 at the end of lines 1 and 2 is the size of the request packets in bytes: the 4-byte RIP header (with the command and version) followed by a single 20-byte address and metric.

Line 3 is the first reply message. The number 25 at the end indicates that 25 address and metric pairs are in the message, which we calculated earlier to be 504 bytes. This is what `ripquery` printed above. We specified the `-s600` option to `tcpdump` telling it to read 600 bytes from the network. This allows it to receive the entire UDP datagram (instead of just the first portion of it) and it then prints the contents of the RIP response. We've omitted that output.

Line 4 is the second response message from the router, with the next 12 address and metric pairs. We can calculate the size of this message to be $12 \times 20 + 4 = 244$, which is what `ripquery` printed earlier.

If we go one router beyond `netb`, to `gateway`, we expect the metric to our subnet (140.252.13.0) to be 2. We can check this by executing:

```
sun % ripquery -n gateway
504 bytes from gateway (140.252.1.4):
                                        lots of other lines deleted
        140.252.1.0, metric 1           the top Ethernet in Figure 10.5
        140.252.13.0, metric 2          the bottom Ethernet in Figure 10.5
```

Here the metric for the top Ethernet in Figure 10.5 (140.252.1.0) stays at 1, since that Ethernet is directly connected to both `gateway` and `netb`. Our subnet 140.252.13.0, however, now has the expected metric of 2.

Another Example

We'll now watch all the unsolicited RIP updates on an Ethernet and see just what RIP sends on a regular basis to its neighbors. Figure 10.7 shows the arrangement of many of the `noao.edu` networks. We have named the routers R*n* for simplicity, where *n* is the subnet number, except for the ones we use elsewhere in the text. We show the point-to-point links with dashed lines and the IP address at each end of these links.

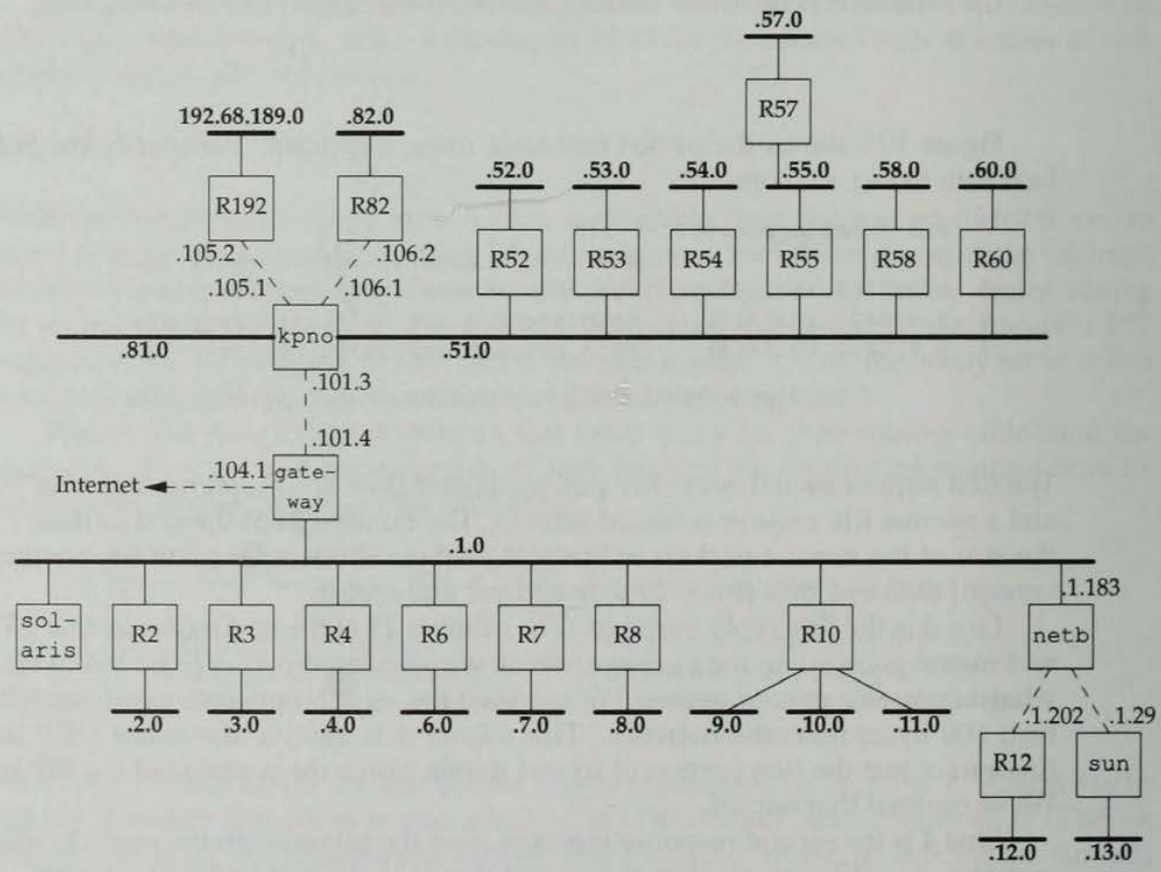


Figure 10.7 Many of the `noao.edu` 140.252 networks.

We'll run the Solaris 2.x program `snoop`, which is similar to `tcpdump`, on the host `solaris`. We can run this program without superuser privileges, but only to capture

broadcast packets, multicast packets, or packets sent to the host. Figure 10.8 shows the packets captured during a 60-second period. We have replaced most of the official hostnames with our notation R*n*.

```
solaris % snoop -P -tr udp port 520
  0.00000 R6.tuc.noao.edu -> 140.252.1.255 RIP R (1 destinations)
  4.49708 R4.tuc.noao.edu -> 140.252.1.255 RIP R (1 destinations)
  6.30506 R2.tuc.noao.edu -> 140.252.1.255 RIP R (1 destinations)
 11.68317 R7.tuc.noao.edu -> 140.252.1.255 RIP R (1 destinations)
 16.19790 R8.tuc.noao.edu -> 140.252.1.255 RIP R (1 destinations)
 16.87131 R3.tuc.noao.edu -> 140.252.1.255 RIP R (1 destinations)
 17.02187 gateway.tuc.noao.edu -> 140.252.1.255 RIP R (15 destinations)
 20.68009 R10.tuc.noao.edu -> BROADCAST    RIP R (4 destinations)

 29.87848 R6.tuc.noao.edu -> 140.252.1.255 RIP R (1 destinations)
 34.50209 R4.tuc.noao.edu -> 140.252.1.255 RIP R (1 destinations)
 36.32385 R2.tuc.noao.edu -> 140.252.1.255 RIP R (1 destinations)
 41.34565 R7.tuc.noao.edu -> 140.252.1.255 RIP R (1 destinations)
 46.19257 R8.tuc.noao.edu -> 140.252.1.255 RIP R (1 destinations)
 46.52199 R3.tuc.noao.edu -> 140.252.1.255 RIP R (1 destinations)
 47.01870 gateway.tuc.noao.edu -> 140.252.1.255 RIP R (15 destinations)
 50.66453 R10.tuc.noao.edu -> BROADCAST    RIP R (4 destinations)
```

Figure 10.8 RIP broadcasts captured at `solaris` over a 60-second period.

The `-P` flag captures packets in nonpromiscuous mode, `-tr` prints the relative timestamps, and `udp port 520` captures only UDP datagrams with a source or destination port of 520.

The first six packets, from R6, R4, R2, R7, R8, and R3, each advertise just one network. If we looked at the packets we would see that R6 advertises a route to 140.252.6.0 with a hop count of 1, R4 advertises a route to 140.252.4.0 with a hop count of 1, and so on.

The router `gateway`, however, advertises 15 routes. We can run `snoop` with the `-v` flag and see the entire contents of the RIP message. (This flag outputs the entire contents of the entire packet: the Ethernet header, the IP header, the UDP header, and the RIP message. We've deleted everything except the RIP information.) Figure 10.9 shows the output.

Compare these advertised hop counts on the 140.252.1 network with the topology shown in Figure 10.7.

A puzzle in the output in Figure 10.8 is why R10 is advertising four networks when Figure 10.7 shows only three. If we look at the RIP packet with `snoop` we see the following advertised routes:

```
RIP:  Address         Metric
RIP:  140.251.0.0     16 (not reachable)
RIP:  140.252.9.0     1
RIP:  140.252.10.0    1
RIP:  140.252.11.0    1
```

The route to the class B network 140.251 is bogus and should not be advertised. (It belongs to another institution, not `noao.edu`.)

```
solaris % snoop -P -v -tr udp port 520 host gateway
                                        many lines deleted
RIP:  Opcode = 2 (route response)
RIP:  Version = 1
RIP:
RIP:  Address         Metric
RIP:
RIP:  140.252.101.0   1
RIP:  140.252.104.0   1
RIP:
RIP:  140.252.51.0    2
RIP:  140.252.81.0    2
RIP:  140.252.105.0   2
RIP:  140.252.106.0   2
RIP:
RIP:  140.252.52.0    3
RIP:  140.252.53.0    3
RIP:  140.252.54.0    3
RIP:  140.252.55.0    3
RIP:  140.252.58.0    3
RIP:  140.252.60.0    3
RIP:  140.252.82.0    3
RIP:  192.68.189.0    3
RIP:
RIP:  140.252.57.0    4
```

Figure 10.9 RIP response from `gateway`.

The notation "BROADCAST" output by `snoop` in Figure 10.8 for the RIP packet sent by R10 means the destination IP address is the limited broadcast address 255.255.255.255 (Section 12.2), instead of the subnet-directed broadcast address (140.252.1.255) that the other routers use.

10.5 RIP Version 2

RFC 1388 [Malkin 1993a] defines newer extensions to RIP, and the result is normally called RIP-2. These extensions don't change the protocol, but pass additional information in the fields labeled "must be zero" in Figure 10.3. RIP and RIP-2 can interoperate if RIP ignores the fields that must be zero.

Figure 10.10 is a redo of that figure, as defined by RIP-2. The *version* is 2 for RIP-2.

The *routing domain* is an identifier of the routing daemon to which this packet belongs. In a Unix implementation this could be the daemon's process ID. This field allows an administrator to run multiple instances of RIP on a single router, each operating within one routing domain.

The *route tag* exists to support exterior gateway protocols. It carries an autonomous system number for EGP and BGP.

The *subnet mask* for each entry applies to the corresponding *IP address*. The *next-hop IP address* is where packets to the corresponding destination IP address should be sent. A value of 0 in this field means packets to the destination should be sent to the system sending the RIP message.

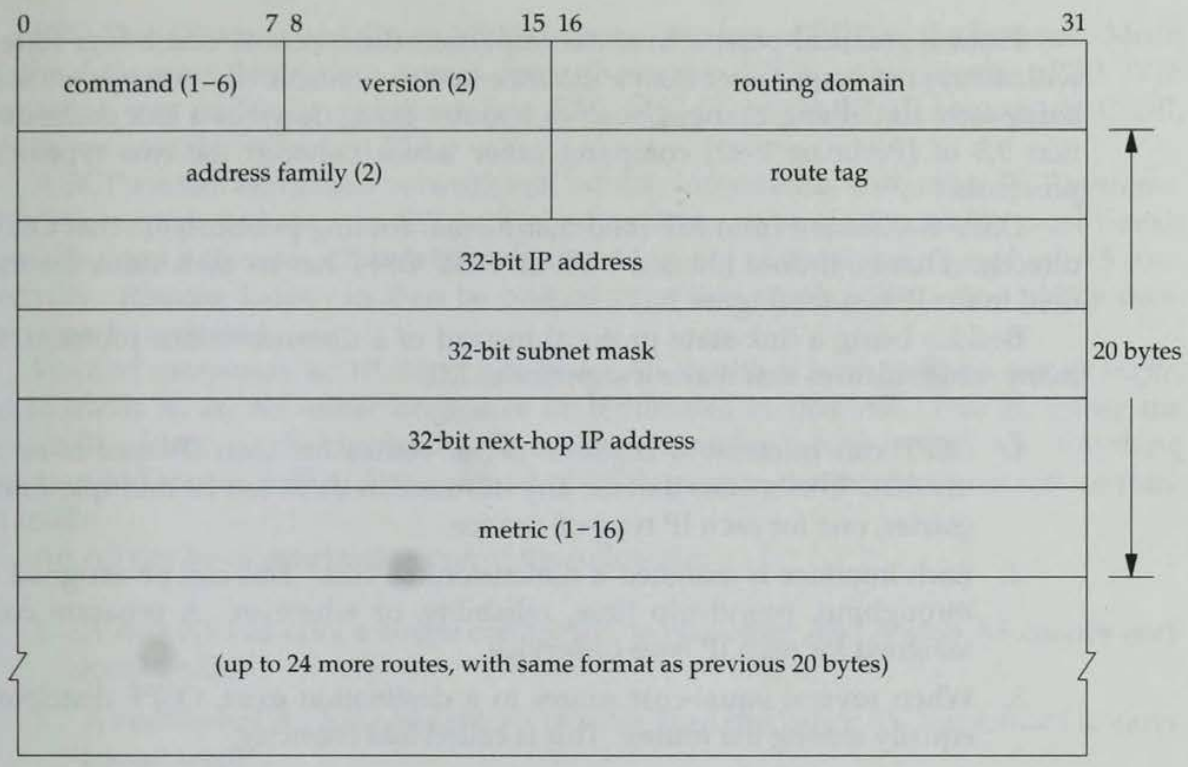


Figure 10.10 Format of a RIP-2 message.

A simple authentication scheme is provided with RIP-2. The first 20-byte entry in a RIP message can specify an *address family* of `0xffff`, with a *route tag* value of 2. The remaining 16 bytes of the entry contain a cleartext password.

Finally, RIP-2 supports multicasting in addition to broadcasting (Chapter 12). This can reduce the load on hosts that are not listening for RIP-2 messages.

10.6 OSPF: Open Shortest Path First

OSPF is a newer alternative to RIP as an interior gateway protocol. It overcomes all the limitations of RIP. OSPF Version 2 is described in RFC 1247 [Moy 1991].

OSPF is a *link-state* protocol, as opposed to RIP, which is a *distance-vector* protocol. The term distance-vector means the messages sent by RIP contain a vector of distances (hop counts). Each router updates its routing table based on the vector of these distances that it receives from its neighbors.

In a link-state protocol a router does not exchange distances with its neighbors. Instead each router actively tests the status of its link to each of its neighbors, sends this information to its other neighbors, which then propagate it throughout the autonomous system. Each router takes this link-state information and builds a complete routing table.

From a practical perspective, the important difference is that a link-state protocol will always converge faster than a distance-vector protocol. By *converge* we mean stabilizing after something changes, such as a router going down or a link going down. Section 9.3 of [Perlman 1992] compares other issues between the two types of routing protocols.

OSPF is different from RIP (and many other routing protocols) in that OSPF uses IP directly. That is, it does not use UDP or TCP. OSPF has its own value for the *protocol* field in the IP header (Figure 3.1).

Besides being a link-state protocol instead of a distance-vector protocol, OSPF has many other features that make it superior to RIP.

1. OSPF can calculate a separate set of routes for each IP type-of-service (Figure 3.2). This means that for any destination there can be multiple routing table entries, one for each IP type-of-service.
2. Each interface is assigned a dimensionless cost. This can be assigned based on throughput, round-trip time, reliability, or whatever. A separate cost can be assigned for each IP type-of-service.
3. When several equal-cost routes to a destination exist, OSPF distributes traffic equally among the routes. This is called *load balancing*.
4. OSPF supports subnets: a subnet mask is associated with each advertised route. This allows a single IP address of any class to be broken into multiple subnets of various sizes. (We showed an example of this in Section 3.7 and called it *variable-length subnets*.) Routes to a host are advertised with a subnet mask of all one bits. A default route is advertised as an IP address of 0.0.0.0 with a mask of all zero bits.
5. Point-to-point links between routers do not need an IP address at each end. These are called *unnumbered* networks. This can save IP addresses—a scarce resource these days!
6. A simple authentication scheme can be used. A cleartext password can be specified, similar to the RIP-2 scheme (Section 10.5).
7. OSPF uses multicasting (Chapter 12), instead of broadcasting, to reduce the load on systems not participating in OSPF.

With most router vendors supporting OSPF, it will start replacing RIP in many networks.

10.7 BGP: Border Gateway Protocol

BGP is an exterior gateway protocol for communication between routers in different autonomous systems. BGP is a replacement for the older EGP that was used on the ARPANET. BGP Version 3 is defined in RFC 1267 [Lougheed and Rekhter 1991].

RFC 1268 [Rekhter and Gross 1991] describes the use of BGP in the Internet. Much of the following description comes from these two RFCs. Also, during 1993 BGP Version 4 was under development (see RFC 1467 [Topolcic 1993]) to support CIDR, which we describe in Section 10.8.

A BGP system exchanges network reachability information with other BGP systems. This information includes the full path of autonomous systems that traffic must transit to reach these networks. This information is adequate to construct a graph of AS connectivity. Routing loops can then be pruned from this graph and routing policy decisions can be enforced.

We first categorize an IP datagram in an AS as either *local traffic* or *transit traffic*. Local traffic in an AS either originates or terminates in that AS. That is, either the source IP address or the destination IP address identifies a host in that AS. Anything else is called transit traffic. A major goal of BGP usage in the Internet is to reduce transit traffic.

An AS can be categorized as one of the following:

1. A *stub AS* has only a single connection to one other AS. A stub AS carries only local traffic.
2. A *multihomed AS* has connections to more than one other AS, but refuses to carry transit traffic.
3. A *transit AS* has connections to more than one other AS and is designed, under certain policy restrictions, to carry both local and transit traffic.

The overall topology of the Internet is then viewed as an arbitrary interconnection of transit, multihomed, and stub ASs. Stub and multihomed ASs need not use BGP—they can run EGP to exchange reachability information with transit ASs.

BGP allows for *policy-based routing*. Policies are determined by the AS administrator and specified to BGP in configuration files. Policy decisions are not part of the protocol, but policy specifications allow a BGP implementation to choose between paths when multiple alternatives exist and to control the redistribution of information. Routing policies are related to political, security, or economic considerations.

BGP is different from RIP and OSPF in that BGP uses TCP as its transport protocol. Two systems running BGP establish a TCP connection between themselves and then exchange the entire BGP routing table. From that point on, incremental updates are sent as the routing table changes.

BGP is a distance vector protocol, but unlike RIP (which announces hops to a destination), BGP enumerates the route to each destination (the sequence of AS numbers to the destination). This removes some of the problems associated with distance-vector protocols. An AS is identified by a 16-bit number.

BGP detects the failure of either the link or the host on the other end of the TCP connection by sending a *keepalive* message to its neighbor on a regular basis. The recommended time between these messages is 30 seconds. This application-level keepalive message is independent of the TCP keepalive option (Chapter 23).

10.8 CIDR: Classless Interdomain Routing

In Chapter 3 we said there is a shortage of class B addresses, requiring sites with multiple networks to now obtain multiple class C network IDs, instead of a single class B network ID. Although the allocation of these class C addresses solves one problem (running out of class B addresses) it introduces another problem: every class C network requires a routing table entry. *Classless Interdomain Routing* (CIDR) is a way to prevent this explosion in the size of the Internet routing tables. It is also called *supernetting* and is described in RFC 1518 [Rekhter and Li 1993] and RFC 1519 [Fuller et al. 1993], with a overview in [Ford, Rekhter, and Braun 1993]. CIDR has the Internet Architecture Board's blessing [Huitema 1993]. RFC 1467 [Topolcic 1993] summarizes the state of deployment of CIDR in the Internet.

The basic concept in CIDR is to allocate multiple IP addresses in a way that allows *summarization* into a smaller number of routing table entries. For example, if a single site is allocated 16 class C addresses, and those 16 are allocated so that they can be summarized, then all 16 can be referenced through a single routing table entry on the Internet. Also, if eight different sites are connected to the same Internet service provider through the same connection point into the Internet, and if the eight sites are allocated eight different IP addresses that can be summarized, then only a single routing table entry need be used on the Internet for all eight sites.

Three features are needed to allow this summarization to take place.

1. Multiple IP addresses to be summarized together for routing must share the same high-order bits of their addresses.
2. The routing tables and routing algorithms must be extended to base their routing decisions on a 32-bit IP address and a 32-bit mask.
3. The routing protocols being used must be extended to carry the 32-bit mask in addition to the 32-bit address. OSPF (Section 10.6) and RIP-2 (Section 10.5) are both capable of carrying the 32-bit mask, as is the proposed BGP Version 4.

As an example, RFC 1466 [Gerich 1993] recommends that new class C addresses in Europe be in the range 194.0.0.0 through 195.255.255.255. In hexadecimal these addresses are from `0xc2000000` through `0xc3ffffff`. This represents 65536 different class C network IDs, but they all share the same high-order 7 bits. In countries other than Europe a single routing table entry with an IP address of `0xc2000000` and a 32-bit mask of `0xfe000000` (254.0.0.0) could be used to route all of these 65536 class C network IDs to a single point. Subsequent bits of the class C address (that is, the bits following 194 or 195) can also be allocated hierarchically, perhaps by country or by service provider, to allow additional summarization within the European routers using additional bits beyond the 7 high-order bits of the 32-bit mask.

CIDR also uses a technique whereby the best match is always the one with the *longest match*: the one with the greatest number of one bits in the 32-bit mask. Continuing the example from the previous paragraph, perhaps one service provider in Europe needs to use a different entry point router than the rest of Europe. If that provider has been allocated the block of addresses 194.0.16.0 through 194.0.31.255 (16 class C network

IDs), routing table entries for just those networks would have an IP address of 194.0.16.0 and a mask of 255.255.240.0 (`0xfffff000`). A datagram being routed to the address 194.0.22.1 would match both this routing table entry and the one for the rest of the European class C networks. But since the mask 255.255.240 is "longer" than the mask 254.0.0.0, the routing table entry with the longer mask is used.

The term "classless" is because routing decisions are now made based on masking operations of the entire 32-bit IP address. Whether the IP address is class A, B, or C makes no difference.

The initial deployment of CIDR is proposed for new class C addresses. Making just this change will slow down the growth of the Internet routing tables, but does nothing for all the existing routes. This is the short-term solution. As a long-term solution, if CIDR were applied to all IP addresses, and if existing IP addresses were reallocated (and all existing hosts renumbered!) according to continental boundaries and service providers, [Ford, Rekhter, and Braun 1993] claim that the current routing table consisting of 10,000 network entries could be reduced to 200 entries.

10.9 Summary

There are two basic types of routing protocols: interior gateway protocols (IGPs), for routers within an autonomous system, and exterior gateway protocols (EGPs), for routers to communicate with routers in other autonomous systems.

The most popular IGP is the Routing Information Protocol (RIP) with OSPF being a newer IGP that is gaining widespread use. A new and popular EGP is the Border Gateway Protocol (BGP). In this chapter we looked at RIP and the types of messages that it exchanges. RIP Version 2 is a recent enhancement that supports subnetting and other minor improvements. We also described OSPF, BGP, and classless interdomain routing (CIDR), a newer technique being deployed to reduce the size of the Internet routing tables.

There are a two other OSI routing protocols that you may encounter. *Interdomain Routing Protocol* (IDRP) started out as a version of BGP modified for use with OSI addresses instead of IP. *Intermediate System to Intermediate System Protocol* (IS–IS) is the OSI standard IGP. It is used for routing CLNP (Connectionless Network Protocol), an OSI protocol similar to IP. IS–IS and OSPF are similar.

Dynamic routing is still a fertile area of internetworking research. The choice of which routing protocol to use, and which routing daemon to run, is complex. [Perlman 1992] provides many of the details.

Exercises

10.1 In Figure 10.9 which of the routes came to `gateway` from the router `kpno`?

10.2 Assume a router has 30 routes to advertise using RIP, requiring one datagram with 25 routes and another with the remaining 5. What happens if once an hour the first datagram with 25 routes is lost?

10.3 The OSPF packet format has a checksum field, but the RIP packet does not. Why?

10.4 What effect does load balancing, as done by OSPF, have on a transport layer?

10.5 Read RFC 1058 for additional details on the implementation of RIP. In Figure 10.8 each router advertises only the routes that it provides, and none of the other routes that it learned about through the other router's broadcasts on the 140.252.1 network. What is this technique called?

10.6 In Section 3.4 we said there are more than 100 hosts on the 140.252.1 subnet in addition to the eight routers we show in Figure 10.7. What do these 100 hosts do with the eight broadcasts that arrive every 30 seconds (Figure 10.8)?

11

UDP: User Datagram Protocol

11.1 Introduction

UDP is a simple, datagram-oriented, transport layer protocol: each output operation by a process produces exactly one UDP datagram, which causes one IP datagram to be sent. This is different from a stream-oriented protocol such as TCP where the amount of data written by an application may have little relationship to what actually gets sent in a single IP datagram.

Figure 11.1 shows the encapsulation of a UDP datagram as an IP datagram.

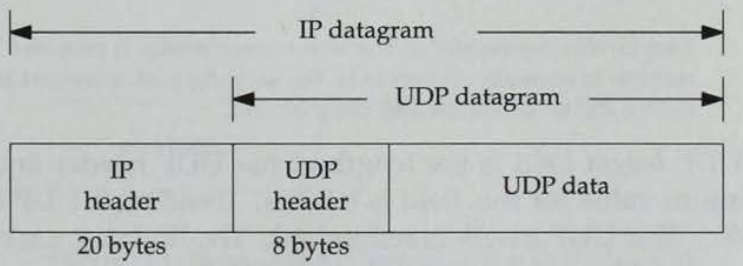


Figure 11.1 UDP encapsulation.

RFC 768 [Postel 1980] is the official specification of UDP.

UDP provides no reliability: it sends the datagrams that the application writes to the IP layer, but there is no guarantee that they ever reach their destination. Given this lack of reliability, we are tempted to think we should avoid UDP and always use a reliable protocol such as TCP. After we describe TCP in Chapter 17 we'll return to this topic and see what types of applications can utilize UDP.

The application needs to worry about the size of the resulting IP datagram. If it exceeds the network's MTU (Section 2.8), the IP datagram is fragmented. This applies to each network that the datagram traverses from the source to the destination, not just the first network connected to the sending host. (We defined this as the *path MTU* in Section 2.9.) We examine IP fragmentation in Section 11.5.

11.2 UDP Header

Figure 11.2 shows the fields in the UDP header.

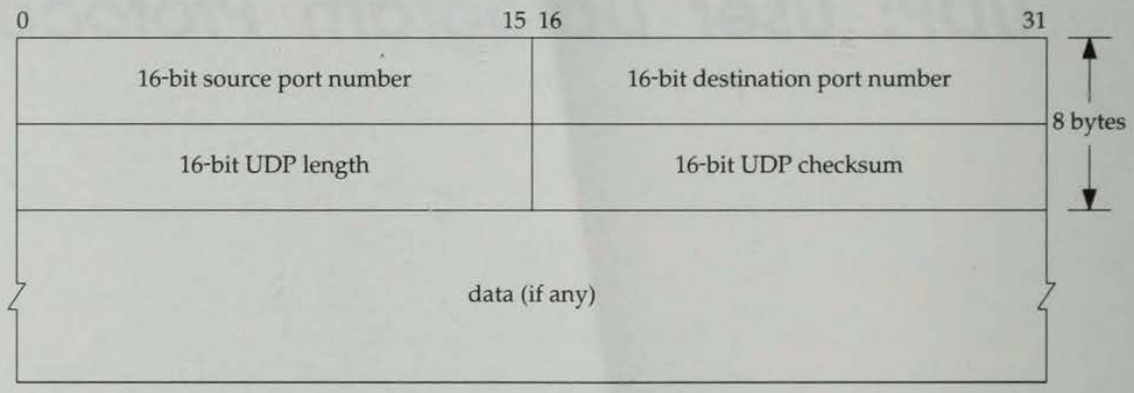


Figure 11.2 UDP header.

The *port numbers* identify the sending process and the receiving process. In Figure 1.8 we showed that TCP and UDP use the destination port number to demultiplex incoming data from IP. Since IP has already demultiplexed the incoming IP datagram to either TCP or UDP (based on the protocol value in the IP header), this means the TCP port numbers are looked at by TCP, and the UDP port numbers by UDP. The TCP port numbers are independent of the UDP port numbers.

> Despite this independence, if a well-known service is provided by both TCP and UDP, the port number is normally chosen to be the same for both transport layers. This is purely for convenience and is not required by the protocols.

The *UDP length* field is the length of the UDP header and the UDP data in bytes. The minimum value for this field is 8 bytes. (Sending a UDP datagram with 0 bytes of data is OK.) This UDP length is redundant. The IP datagram contains its total length in bytes (Figure 3.1), so the length of the UDP datagram is this total length minus the length of the IP header (which is specified by the header length field in Figure 3.1).

11.3 UDP Checksum

The *UDP checksum* covers the UDP header and the UDP data. Recall that the checksum in the IP header only covers the IP header—it does not cover any data in the IP

datagram. Both UDP and TCP have checksums in their headers to cover their header and their data. With UDP the checksum is optional, while with TCP it is mandatory.

Although the basics for calculating the UDP checksum are similar to what we described in Section 3.2 for the IP header checksum (the ones complement sum of 16-bit words), there are differences. First, the length of the UDP datagram can be an odd number of bytes, while the checksum algorithm adds 16-bit words. The solution is to append a pad byte of 0 to the end, if necessary, just for the checksum computation. (That is, this possible pad byte is not transmitted.)

Next, both UDP and TCP include a 12-byte pseudo-header with the UDP datagram (or TCP segment) just for the checksum computation. This pseudo-header includes certain fields from the IP header. The purpose is to let UDP double-check that the data has arrived at the correct destination (i.e., that IP has not accepted a datagram that is not addressed to this host, and that IP has not given UDP a datagram that is for another upper layer). Figure 11.3 shows the pseudo-header along with a UDP datagram.

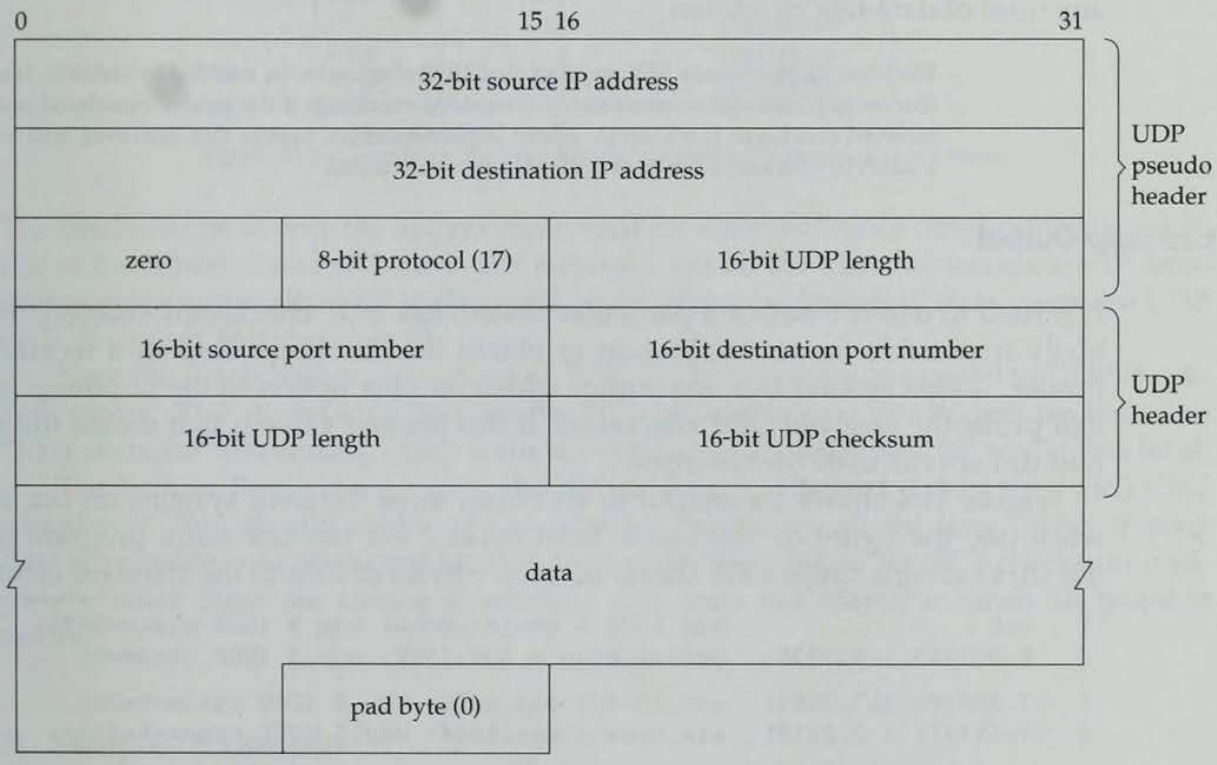


Figure 11.3 Fields used for computation of UDP checksum.

In this figure we explicitly show a datagram with an odd length, requiring a pad byte for the checksum computation. Notice that the length of the UDP datagram appears twice in the checksum computation.

If the calculated checksum is 0, it is stored as all one bits (65535), which is equivalent in ones-complement arithmetic. If the transmitted checksum is 0, it indicates that the sender did not compute the checksum.

If the sender did compute a checksum and the receiver detects a checksum error, the UDP datagram is silently discarded. No error message is generated. (This is what happens if an IP header checksum error is detected by IP.)

This UDP checksum is an end-to-end checksum. It is calculated by the sender, and then verified by the receiver. It is designed to catch any modification of the UDP header or data anywhere between the sender and receiver.

Despite UDP checksums being optional, they should *always* be enabled. During the 1980s some computer vendors turned off UDP checksums by default, to speed up their implementation of Sun's Network File System (NFS), which uses UDP. While this *might* be acceptable on a single LAN, where the cyclic redundancy check on the data-link frame (e.g., Ethernet or token ring frame) can detect most corruption of the frame, when the datagrams pass through routers, all bets are off. Believe it or not, there have been routers with software and hardware bugs that have modified bits in the datagrams being routed. These errors are undetectable in a UDP datagram if the end-to-end UDP checksum is disabled. Also realize that some data-link protocols (e.g., SLIP) don't have any form of data-link checksum.

> The Host Requirements RFC requires that UDP checksums be enabled by default. It also states that an implementation must verify a received checksum if the sender calculated one (i.e., the received checksum is nonzero). Many implementations violate this, however, and only verify a received checksum if outgoing checksums are enabled.

`tcpdump` Output

It is hard to detect whether a particular system has UDP checksums enabled. It is normally impossible for an application to obtain the checksum field in a received UDP header. To get around this, the author added another option to the `tcpdump` program that prints the received UDP checksum. If this printed value is 0, it means the sending host did not calculate the checksum.

Figure 11.4 shows the output to and from three different systems on our test network (see the figure on the inside front cover). We ran our `sock` program (Appendix C), sending a single UDP datagram with 9 bytes of data to the standard echo server.

```
1   0.0                 sun.1900 > gemini.echo: udp 9 (UDP cksum=6e90)
2   0.303755 ( 0.3038)  gemini.echo > sun.1900: udp 9 (UDP cksum=0)

3  17.392480 (17.0887)  sun.1904 > aix.echo: udp 9 (UDP cksum=6e3b)
4  17.614371 ( 0.2219)  aix.echo > sun.1904: udp 9 (UDP cksum=6e3b)

5  32.092454 (14.4781)  sun.1907 > solaris.echo: udp 9 (UDP cksum=6e74)
6  32.314378 ( 0.2219)  solaris.echo > sun.1907: udp 9 (UDP cksum=6e74)
```

Figure 11.4 `tcpdump` output to see whether other hosts enable UDP checksum.

We can see from this that two of the three systems have UDP checksums enabled.

Also notice that for this simple example the outgoing datagram has the same checksum as the incoming datagram (lines 3 and 4, 5 and 6). Looking at Figure 11.3 we see that the two IP addresses are swapped, as are the two port numbers. The other fields in the pseudo-header and the UDP header are the same, as is the data being echoed. This

reiterates that the UDP checksums (indeed, all the checksums in the TCP/IP protocol suite) are simple 16-bit sums. They cannot detect an error that swaps two of the 16-bit values.

> The author also directed a DNS query at each of the eight root name servers described in Section 14.2. The DNS uses UDP primarily, and only two of the eight had UDP checksums enabled!

Some Statistics

[Mogul 1992] provides counts of various checksum errors on a busy NFS (Network File System) server that had been up for 40 days. Figure 11.5 summarizes these numbers.

Layer	Number of checksum errors	Approximate total number of packets
Ethernet	446	170,000,000
IP	14	170,000,000
UDP	5	140,000,000
TCP	350	30,000,000

Figure 11.5 Counts of corrupted packets detected by various checksums.

The final column is only the approximate total for each row, since other protocols are in use at the Ethernet and IP layers. For example, not all the Ethernet frames are IP datagrams, since minimally ARP is also used on an Ethernet. Not all IP datagrams are UDP or TCP, since ICMP also uses IP.

Note the much higher percentage of TCP checksum errors compared to UDP checksum errors. This is probably because the TCP connections on this system tended to be "long distance" (traversing many routers, bridges, etc.) while the UDP traffic was local.

The bottom line is not to trust the data-link (e.g., Ethernet, token ring, etc.) CRC completely. You should enable the end-to-end checksums all the time. Also, if your data is valuable, you might not want to trust either the UDP or the TCP checksum completely, since these are simple checksums and were not meant to catch all possible errors.

11.4 A Simple Example

We'll use our `sock` program to generate some UDP datagrams that we can watch with `tcpdump`:

```
bsdi % sock -v -u -i -n4 svr4 discard
connected on 140.252.13.35.1108 to 140.252.13.34.9

bsdi % sock -v -u -i -n4 -w0 svr4 discard
connected on 140.252.13.35.1110 to 140.252.13.34.9
```

The first time we execute the program we specify the verbose mode (`-v`) to see the ephemeral port numbers, specify UDP (`-u`) instead of the default TCP, and use the

source mode (`-i`) to send data instead of trying to read and write standard input and output. The `-n4` option says to output 4 datagrams (instead of the default 1024) and the destination host is `svr4`. We described the discard service in Section 1.12. We use the default output size of 1024 bytes per write.

The second time we run the program we specify `-w0`, causing 0-length datagrams to be written. Figure 11.6 shows the `tcpdump` output for both commands.

```
1   0.0                 bsdi.1108 > svr4.discard: udp 1024
2   0.002424 ( 0.0024)  bsdi.1108 > svr4.discard: udp 1024
3   0.006210 ( 0.0038)  bsdi.1108 > svr4.discard: udp 1024
4   0.010276 ( 0.0041)  bsdi.1108 > svr4.discard: udp 1024

5  41.720114 (41.7098)  bsdi.1110 > svr4.discard: udp 0
6  41.721072 ( 0.0010)  bsdi.1110 > svr4.discard: udp 0
7  41.722094 ( 0.0010)  bsdi.1110 > svr4.discard: udp 0
8  41.723070 ( 0.0010)  bsdi.1110 > svr4.discard: udp 0
```

Figure 11.6 `tcpdump` output when UDP datagrams are sent in one direction.

This output shows the four 1024-byte datagrams, followed by the four 0-length datagrams. Each datagram followed the previous by a few milliseconds. (It took 41 seconds to type in the second command.)

There is no communication between the sender and receiver before the first datagram is sent. (We'll see in Chapter 17 that TCP must establish a connection with the other end before the first byte of data can be sent.) Also, there are no acknowledgments by the receiver when the data is received. The sender, in this example, has no idea whether the other end receives the datagrams.

Finally note that the source UDP port number changes each time the program is run. First it is 1108 and then it is 1110. We mentioned in Section 1.9 that the ephemeral port numbers used by clients are typically in the range 1024 through 5000, as we see here.

11.5 IP Fragmentation

As we described in Section 2.8, the physical network layer normally imposes an upper limit on the size of the frame that can be transmitted. Whenever the IP layer receives an IP datagram to send, it determines which local interface the datagram is being sent on (routing), and queries that interface to obtain its MTU. IP compares the MTU with the datagram size and performs fragmentation, if necessary. Fragmentation can take place either at the original sending host or at an intermediate router.

When an IP datagram is fragmented, it is not reassembled until it reaches its final destination. (This handling of reassembly differs from some other networking protocols that require reassembly to take place at the next hop, not at the final destination.) The IP layer at the destination performs the reassembly. The goal is to make fragmentation and reassembly transparent to the transport layer (TCP and UDP), which it is, except for possible performance degradation. It is also possible for the fragment of a datagram to

again be fragmented (possibly more than once). The information maintained in the IP header for fragmentation and reassembly provides enough information to do this.

Recalling the IP header (Figure 3.1, p. 34), the following fields are used in fragmentation. The *identification* field contains a unique value for each IP datagram that the sender transmits. This number is copied into each fragment of a particular datagram. (We now see the use for this field.) The *flags* field uses one bit as the "more fragments" bit. This bit is turned on for each fragment comprising a datagram except the final fragment. The *fragment offset* field contains the offset of this fragment from the beginning of the original datagram. Also, when a datagram is fragmented the *total length* field of each fragment is changed to be the size of that fragment.

Finally, one of the bits in the flags field is called the "don't fragment" bit. If this is turned on, IP will not fragment the datagram. Instead the datagram is thrown away and an ICMP error ("fragmentation needed but don't fragment bit set," Figure 6.3) is sent to the originator. We'll see an example of this error in the next section.

When an IP datagram is fragmented, each fragment becomes its own packet, with its own IP header, and is routed independently of any other packets. This makes it possible for the fragments of a datagram to arrive at the final destination out of order, but there is enough information in the IP header to allow the receiver to reassemble the fragments correctly.

Although IP fragmentation looks transparent, there is one feature that makes it less than desirable: if one fragment is lost the entire datagram must be retransmitted. To understand why this happens, realize that IP itself has no timeout and retransmission—that is the responsibility of the higher layers. (TCP performs timeout and retransmission, UDP doesn't. Some UDP applications perform timeout and retransmission themselves.) When a fragment is lost that came from a TCP segment, TCP will time out and retransmit the entire TCP segment, which corresponds to an IP datagram. There is no way to resend only one fragment of a datagram. Indeed, if the fragmentation was done by an intermediate router, and not the originating system, there is no way for the originating system to know how the datagram was fragmented. For this reason alone, fragmentation is often avoided. [Kent and Mogul 1987] provide arguments for avoiding fragmentation.

Using UDP it is easy to generate IP fragmentation. (We'll see later that TCP tries to avoid fragmentation and that it is nearly impossible for an application to force TCP to send segments large enough to require fragmentation.) We can use our `sock` program and increase the size of the datagram until fragmentation occurs. On an Ethernet the maximum amount of data in a frame is 1500 bytes (Figure 2.1), which leaves 1472 bytes for our data, assuming 20 bytes for the IP header and 8 bytes for the UDP header. We'll run our `sock` program, with data sizes of 1471, 1472, 1473, and 1474 bytes. We expect the last two to cause fragmentation:

```
bsdi % sock -u -i -n1 -w1471 svr4 discard
bsdi % sock -u -i -n1 -w1472 svr4 discard
bsdi % sock -u -i -n1 -w1473 svr4 discard
bsdi % sock -u -i -n1 -w1474 svr4 discard
```

Figure 11.7 shows the corresponding `tcpdump` output.

```
1   0.0                  bsdi.1112 > svr4.discard: udp 1471
2  21.008303 (21.0083)   bsdi.1114 > svr4.discard: udp 1472
3  50.449704 (29.4414)   bsdi.1116 > svr4.discard: udp 1473 (frag 26304:1480@0+)
4  50.450040 ( 0.0003)   bsdi > svr4: (frag 26304:1@1480)
5  75.328650 (24.8786)   bsdi.1118 > svr4.discard: udp 1474 (frag 26313:1480@0+)
6  75.328982 ( 0.0003)   bsdi > svr4: (frag 26313:2@1480)
```

Figure 11.7 Watching fragmentation of UDP datagrams.

The first two UDP datagrams (lines 1 and 2) fit into Ethernet frames, and are not fragmented. But the length of the IP datagram corresponding to the write of 1473 bytes is 1501, which must be fragmented (lines 3 and 4). Similarly the datagram generated by the write of 1474 bytes is 1502, and is also fragmented (lines 5 and 6).

When the IP datagram is fragmented, `tcpdump` prints additional information. First, the output `frag 26304` (lines 3 and 4) and `frag 26313` (lines 5 and 6) specify the value of the identification field in the IP header.

The next number in the fragmentation information, the 1480 between the colon and the at sign in line 3, is the size, excluding the IP header. The first fragment of both datagrams contains 1480 bytes of data: 8 bytes for the UDP header and 1472 bytes of user data. (The 20-byte IP header makes the packet exactly 1500 bytes.) The second fragment of the first datagram (line 4) contains 1 byte of data—the remaining byte of user data. The second fragment of the second datagram (line 6) contains the remaining 2 bytes of user data.

Fragmentation requires that the data portion of the generated fragments (that is, everything excluding the IP header) be a multiple of 8 bytes for all fragments other than the final one. In this example, 1480 is a multiple of 8.

The number following the at sign is the offset of the data in the fragment, from the start of the datagram. The first fragment of both datagrams starts at 0 (lines 3 and 5) and the second fragment of both datagrams starts at byte offset 1480 (lines 4 and 6). The plus sign following this offset that is printed for the first fragment of both datagrams means there are more fragments comprising this datagram. This plus sign corresponds to the "more fragments" bit in the 3-bit flags in the IP header. The purpose of this bit is to let the receiver know when it has completed the reassembly of all the fragments for a datagram.

Finally, notice that lines 4 and 6 (fragments other than the first) omit the protocol (UDP) and the source and destination ports. The protocol could be printed, since it's in the IP header that's copied into the fragments. The port numbers, however, are in the UDP header, which only occurs in the first fragment.

Figure 11.8 shows what's happening with the third datagram that is sent (with 1473 bytes of user data). It reiterates that any transport layer header appears only in the first fragment.

Also note the terminology: an *IP datagram* is the unit of end-to-end transmission at the IP layer (before fragmentation and after reassembly), and a *packet* is the unit of data passed between the IP layer and the link layer. A packet can be a complete IP datagram or a fragment of an IP datagram.

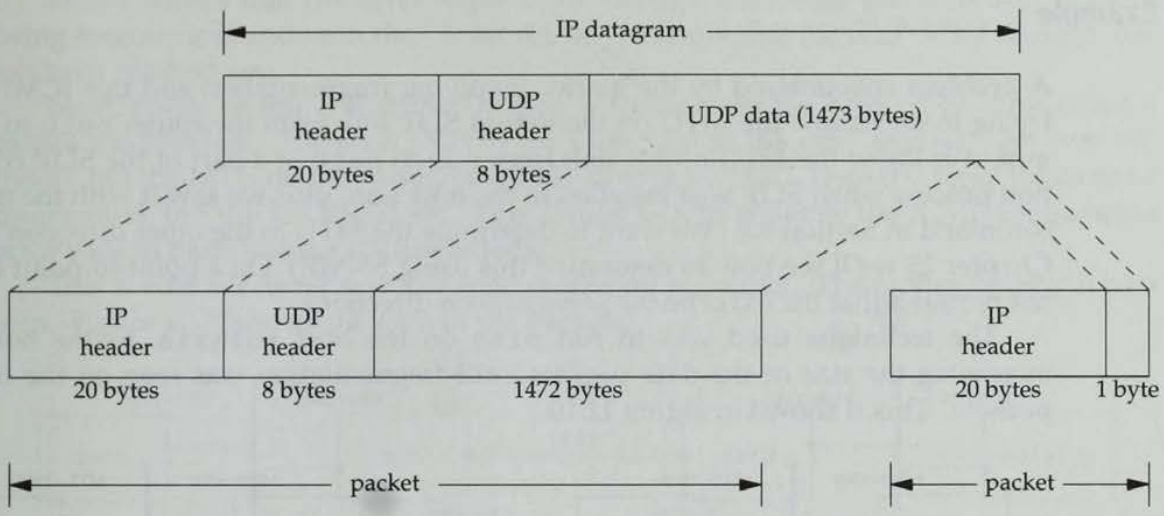


Figure 11.8 Example of UDP fragmentation.

11.6 ICMP Unreachable Error (Fragmentation Required)

Another variation of the ICMP unreachable error occurs when a router receives a datagram that requires fragmentation, but the don't fragment (DF) flag is turned on in the IP header. This error can be used by a program that needs to determine the smallest MTU in the path to a destination—called the *path MTU discovery* mechanism (Section 2.9).

Figure 11.9 shows the format of the ICMP unreachable error for this case. This differs from Figure 6.10 because bits 16–31 of the second 32-bit word can provide the MTU of the next hop, instead of being 0.

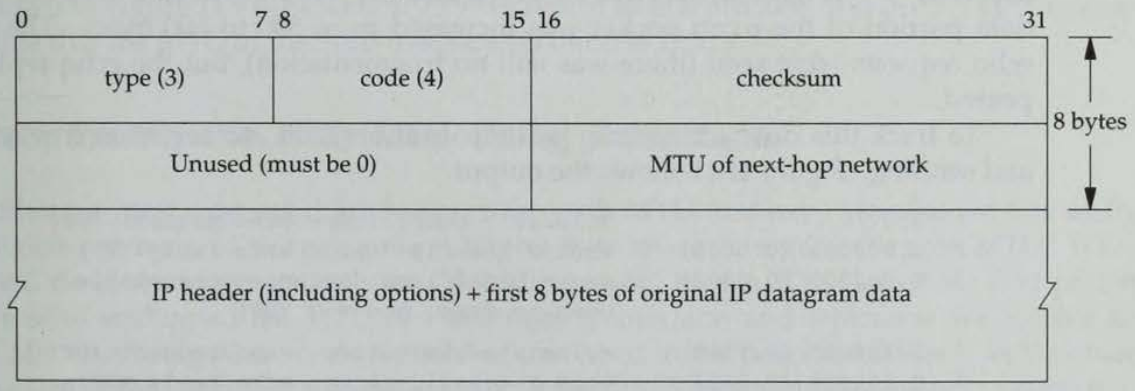


Figure 11.9 ICMP unreachable error when fragmentation required but don't fragment bit set.

If a router doesn't provide this newer format ICMP error, the next-hop MTU is set to 0.

> The new Router Requirements RFC [Almquist 1993] states that a router must generate this newer form when originating this ICMP unreachable error.

Example

A problem encountered by the author involving fragmentation and this ICMP error is trying to determine the MTU on the dialup SLIP link from the router `netb` to the host `sun`. We know the MTU of this link from `sun` to `netb`: it's part of the SLIP configuration process when SLIP was installed in the host `sun`, plus we saw it with the `netstat` command in Section 3.9. We want to determine the MTU in the other direction also. (In Chapter 25 we'll see how to determine this using SNMP.) On a point-to-point link, it is not required that the MTU be the same in both directions.

The technique used was to run `ping` on the host `solaris`, to the host `bsdi`, increasing the size of the data packets until fragmentation was seen on the incoming packets. This is shown in Figure 11.10.

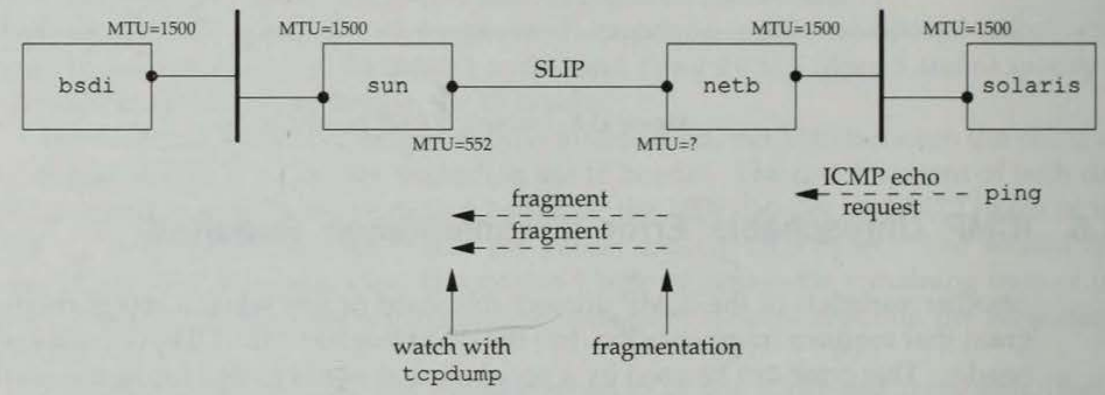


Figure 11.10 Systems being used to determine MTU of SLIP link from `netb` to `sun`.

`tcpdump` was run on the host `sun`, watching the SLIP link, to see when fragmentation occurred. No fragmentation was observed and everything was fine until the size of the data portion of the `ping` packet was increased from 500 to 600 bytes. The incoming echo requests were seen (there was still no fragmentation), but the echo replies disappeared.

To track this down, `tcpdump` was also run on `bsdi`, to see what it was receiving and sending. Figure 11.11 shows the output.

```
1  0.0                  solaris > bsdi: icmp: echo request (DF)
2  0.000000 (0.0000)    bsdi > solaris: icmp: echo reply (DF)
3  0.000000 (0.0000)    sun > bsdi: icmp: solaris unreachable -
                        need to frag, mtu = 0 (DF)
4  0.738400 (0.7384)    solaris > bsdi: icmp: echo request (DF)
5  0.748800 (0.0104)    bsdi > solaris: icmp: echo reply (DF)
6  0.748800 (0.0000)    sun > bsdi: icmp: solaris unreachable -
                        need to frag, mtu = 0 (DF)
```

Figure 11.11 `tcpdump` output for `ping` of `bsdi` from `solaris` with 600-byte IP datagram.

First, the notation `(DF)` in each line means the don't fragment bit is turned on in the IP header. It turns out that Solaris 2.2 normally turns this bit on, as part of its implementation of the path MTU discovery mechanism.

Line 1 shows that the echo request got through the router `netb` to `sun` without being fragmented, and with the DF bit set, so we know that the SLIP MTU of `netb` has not been reached yet.

Next, notice in line 2 that the DF flag is copied into the echo reply. This is what causes the problem. The echo reply is the same size as the echo request (just over 600 bytes), but the MTU on `sun`'s outgoing SLIP interface is 552. The echo reply needs to be fragmented, but the DF flag is set. This causes `sun` to generate the ICMP unreachable error back to `bsdi` (where it's discarded).

This is why we never saw any echo replies on `solaris`. The replies never got past `sun`. Figure 11.12 shows the path of the packets.

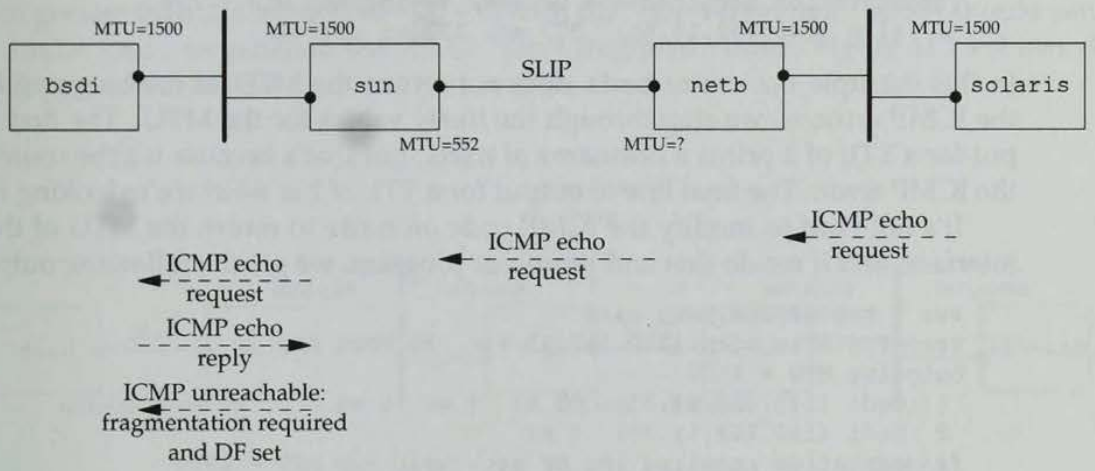


Figure 11.12 Packets exchanged in example.

Finally, the notation `mtu=0` in lines 3 and 6 of Figure 11.11 indicates that `sun` does not return the MTU of the outgoing interface in the ICMP unreachable message, as shown in Figure 11.9. (In Section 25.9 we return to this problem and use SNMP to determine that the MTU of the SLIP interface on `netb` is 1500.)

11.7 Determining the Path MTU Using Traceroute

Although most systems don't support the path MTU discovery feature, we can easily modify a version of `traceroute` (Chapter 8) to let us determine the path MTU. What we'll do is send packets with the "don't fragment" bit set. The size of the first packet we send will equal the MTU of the outgoing interface, and whenever we receive an ICMP "can't fragment" error (which we described in the previous section) we'll reduce the size of the packet. If the router sending the ICMP error sends the newer version that includes the MTU of the outgoing interface, we'll use that value; otherwise we'll try the next smallest MTU. As RFC 1191 [Mogul and Deering 1990] states, there are a limited number of MTUs, so our program has a table of the likely values and moves to the next smallest value.

Let's first try it from our host `sun` to the host `slip`, knowing that the SLIP link has an MTU of 296:

```
sun % traceroute.pmtu slip
traceroute to slip (140.252.13.65), 30 hops max
outgoing MTU = 1500
 1  bsdi (140.252.13.35)  15 ms  6 ms  6 ms
 2  bsdi (140.252.13.35)  6 ms
fragmentation required and DF set, trying new MTU = 1492
fragmentation required and DF set, trying new MTU = 1006
fragmentation required and DF set, trying new MTU = 576
fragmentation required and DF set, trying new MTU = 552
fragmentation required and DF set, trying new MTU = 544
fragmentation required and DF set, trying new MTU = 512
fragmentation required and DF set, trying new MTU = 508
fragmentation required and DF set, trying new MTU = 296
 2  slip (140.252.13.65)  377 ms  377 ms  377 ms
```

In this example the router `bsdi` does not return the MTU of the outgoing interface in the ICMP error, so we step through the likely values for the MTU. The first line of output for a TTL of 2 prints a hostname of `bsdi`, but that's because it's the router returning the ICMP error. The final line of output for a TTL of 2 is what we're looking for.

It's not hard to modify the ICMP code on `bsdi` to return the MTU of the outgoing interface, and if we do that and rerun our program, we get the following output:

```
sun % traceroute.pmtu slip
traceroute to slip (140.252.13.65), 30 hops max
outgoing MTU = 1500
 1  bsdi (140.252.13.35)  53 ms  6 ms  6 ms
 2  bsdi (140.252.13.35)  6 ms
fragmentation required and DF set, next hop MTU = 296
 2  slip (140.252.13.65)  377 ms  378 ms  377 ms
```

Here we don't have to try eight different values for the MTU before finding the right one—the router returns the correct value.

The Worldwide Internet

As an experiment, this modified version of `traceroute` was run numerous times to various hosts around the world. Fifteen countries (including Antartica) were reached and various transatlantic and transpacific links were used. Before doing this, however, the MTU of the dialup SLIP link between the author's subnet and the router `netb` (Figure 11.12) was increased to 1500, the same as an Ethernet.

Out of 18 runs, only 2 had a path MTU of less than 1500. One of the transatlantic links had an MTU of 572 (a value not even listed as a likely value in RFC 1191) and the router did return the newer format ICMP error. Another link, between two routers in Japan, wouldn't handle a 1500-byte frame, and the router did not return the newer format ICMP error. Setting the MTU down to 1006 did work.

The conclusion we can make from this experiment is that many, but not all, WANs today can handle packets larger than 512 bytes. Using the path MTU discovery feature will allow applications to take advantage of these larger MTUs.

11.8 Path MTU Discovery with UDP

Let's examine the interaction between an application using UDP and the path MTU discovery mechanism. We want to see what happens when the application writes datagrams that are too big for some intermediate link.

Example

Since the only system that we've been using that supports the path MTU discovery mechanism is Solaris 2.x, we'll use it as the source host to send 650-byte datagrams to `slip`. Since our host `slip` sits behind a SLIP link with an MTU of 296, any UDP datagram greater than 268 bytes (296 – 20 – 8) with the "don't fragment" bit set should cause the router `bsdi` to generate the ICMP "can't fragment" error. Figure 11.13 shows the topology and the MTUs.

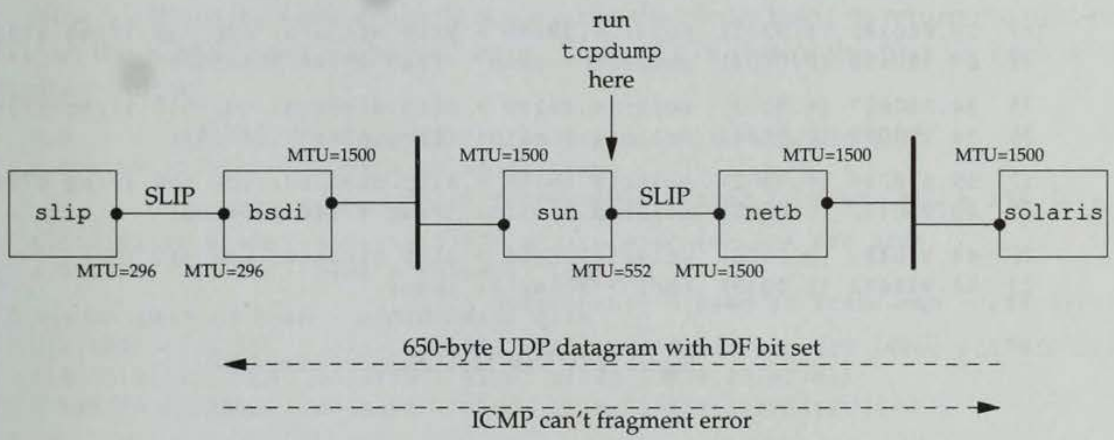


Figure 11.13 Systems used for path MTU discovery using UDP.

The following command generates ten 650-byte UDP datagrams, with a 5-second pause between each datagram:

```
solaris % sock -u -i -n10 -w650 -p5 slip discard
```

Figure 11.14 shows the `tcpdump` output. When this example was run, the router `bsdi` was set to not return the next-hop MTU as part of the ICMP "can't fragment" error.

The first datagram is sent with the DF bit set (line 1) and generates the expected error from the router `bsdi` (line 2). What's puzzling is that the next datagram is also sent with the DF bit set (line 3) and generates the same ICMP error (line 4). We would expect this datagram to be sent with the DF bit off.

On line 5 it appears IP has finally learned that datagrams to this destination should not be sent with the DF bit set, so IP goes ahead and fragments the datagrams at the source host. This is different from earlier examples where IP sends the datagram that is passed to it by UDP and allows the router with the smaller MTU (`bsdi` in this case) to

```
 1   0.0                solaris.38196 > slip.discard: udp 650 (DF)
 2   0.004218 (0.0042)  bsdi > solaris: icmp:
                                slip unreachable - need to frag, mtu = 0 (DF)

 3   4.980528 (4.9763)  solaris.38196 > slip.discard: udp 650 (DF)
 4   4.984503 (0.0040)  bsdi > solaris: icmp:
                                slip unreachable - need to frag, mtu = 0 (DF)

 5   9.870407 (4.8859)  solaris.38196 > slip.discard: udp 650 (frag 47942:552@0+)
 6   9.960056 (0.0896)  solaris > slip: (frag 47942:106@552)

 7  14.940338 (4.9803)  solaris.38196 > slip.discard: udp 650 (DF)
 8  14.944466 (0.0041)  bsdi > solaris: icmp:
                                slip unreachable - need to frag, mtu = 0 (DF)

 9  19.890015 (4.9455)  solaris.38196 > slip.discard: udp 650 (frag 47944:552@0+)
10  19.950463 (0.0604)  solaris > slip: (frag 47944:106@552)

11  24.870401 (4.9199)  solaris.38196 > slip.discard: udp 650 (frag 47945:552@0+)
12  24.960038 (0.0896)  solaris > slip: (frag 47945:106@552)

13  29.880182 (4.9201)  solaris.38196 > slip.discard: udp 650 (frag 47946:552@0+)
14  29.940498 (0.0603)  solaris > slip: (frag 47946:106@552)

15  34.860607 (4.9201)  solaris.38196 > slip.discard: udp 650 (frag 47947:552@0+)
16  34.950051 (0.0894)  solaris > slip: (frag 47947:106@552)

17  39.870216 (4.9202)  solaris.38196 > slip.discard: udp 650 (frag 47948:552@0+)
18  39.930443 (0.0602)  solaris > slip: (frag 47948:106@552)

19  44.940485 (5.0100)  solaris.38196 > slip.discard: udp 650 (DF)
20  44.944432 (0.0039)  bsdi > solaris: icmp:
                                slip unreachable - need to frag, mtu = 0 (DF)
```

Figure 11.14 Path MTU discovery using UDP.

do the fragmentation. Since the ICMP "can't fragment" message didn't specify the next-hop MTU, it appears that IP guesses that an MTU of 576 is OK. The first fragment (line 5) contains 544 bytes of UDP data, the 8-byte UDP header, and the 20-byte IP header, for a total IP datagram size of 572 bytes. The second fragment (line 6) contains the remaining 106 bytes of UDP data and a 20-byte IP header.

Unfortunately the next datagram, line 7, has its DF bit set, so it's discarded by `bsdi` and the ICMP error returned. What has happened here is that an IP timer has expired telling IP to see if the path MTU has increased by setting the DF bit again. We see this happen again on lines 19 and 20. Comparing the times on lines 7 and 19 it appears that IP turns on the DF bit, to see if the path MTU has increased, every 30 seconds.

> This 30-second timer value is way too small. RFC 1191 recommends a value of 10 minutes. It can be changed by modifying the parameter `ip_ire_pathmtu_interval` (Section E.4). Also there is no way in Solaris 2.2 to turn off this path MTU discovery for a single UDP application or for all UDP applications. It can only be enabled or disabled on a systemwide basis by changing the parameter `ip_path_mtu_discovery`. As we can see from this example, enabling path MTU discovery when UDP applications write datagrams that will probably be fragmented can cause datagrams to be discarded.

The maximum datagram size assumed by the IP layer on `solaris` (576 bytes) is not right. In Figure 11.13 we see that the real MTU is 296 bytes. This means the fragments generated by `solaris` will be fragmented again by `bsdi`. Figure 11.15 shows the `tcpdump` output collected on the destination host (`slip`) for the first datagram that arrives (lines 5 and 6 from Figure 11.14).

```
1  0.0                 solaris.38196 > slip.discard: udp 650 (frag 47942:272@0+)
2  0.304513 (0.3045)   solaris > slip: (frag 47942:272@272+)
3  0.334651 (0.0301)   solaris > slip: (frag 47942:8@544+)
4  0.466642 (0.1320)   solaris > slip: (frag 47942:106@552)
```

Figure 11.15 First datagram arriving at host `slip` from `solaris`.

In this example the host `solaris` should not fragment the outgoing datagrams but should turn off the DF bit and let the router with the smaller MTU do the fragmentation.

Now we'll run the same example but modify the router `bsdi` to return the next-hop MTU in the ICMP "can't fragment" error. Figure 11.16 shows the first six lines of the `tcpdump` output.

```
1  0.0                 solaris.37974 > slip.discard: udp 650 (DF)
2  0.004199 (0.0042)   bsdi > solaris: icmp:
                               slip unreachable - need to frag, mtu = 296 (DF)

3  4.950193 (4.9460)   solaris.37974 > slip.discard: udp 650 (DF)
4  4.954325 (0.0041)   bsdi > solaris: icmp:
                               slip unreachable - need to frag, mtu = 296 (DF)

5  9.779855 (4.8255)   solaris.37974 > slip.discard: udp 650 (frag 35278:272@0+)
6  9.930018 (0.1502)   solaris > slip: (frag 35278:272@272+)
7  9.990170 (0.0602)   solaris > slip: (frag 35278:114@544)
```

Figure 11.16 Path MTU discovery using UDP.

Again, the first two datagrams are sent with the DF bit set, and both elicit the ICMP error. The ICMP error now specifies the next-hop MTU of 296.

In lines 5, 6, and 7 we see the source host perform fragmentation, similar to Figure 11.14. But knowing the next-hop MTU, only three fragments are generated, compared to the four fragments generated by the router `bsdi` in Figure 11.15.

11.9 Interaction Between UDP and ARP

Using UDP we can see an interesting (and often unmentioned) interaction with UDP and typical implementations of ARP.

We use our `sock` program to generate a single UDP datagram with 8192 bytes of data. We expect this to generate six fragments on an Ethernet (see Exercise 11.3). We also assure that the ARP cache is empty before running the program, so that an ARP request and reply must be exchanged before the first fragment is sent.

```
bsdi % arp -a                              verify ARP cache is empty
bsdi % sock -u -i -n1 -w8192 svr4 discard
```

We expect the first fragment to cause an ARP request to be sent. Five more fragments are generated by IP and this presents two timing questions that we'll need to use `tcpdump` to answer: are the remaining fragments ready to be sent before the ARP reply is received, and if so, what does ARP do with multiple packets to a given destination when it's waiting for an ARP reply? Figure 11.17 shows the `tcpdump` output.

```
 1  0.0                  arp who-has svr4 tell bsdi
 2  0.001234 (0.0012)    arp who-has svr4 tell bsdi
 3  0.001941 (0.0007)    arp who-has svr4 tell bsdi
 4  0.002775 (0.0008)    arp who-has svr4 tell bsdi
 5  0.003495 (0.0007)    arp who-has svr4 tell bsdi
 6  0.004319 (0.0008)    arp who-has svr4 tell bsdi
 7  0.008772 (0.0045)    arp reply svr4 is-at 0:0:c0:c2:9b:26
 8  0.009911 (0.0011)    arp reply svr4 is-at 0:0:c0:c2:9b:26
 9  0.011127 (0.0012)    bsdi > svr4: (frag 10863:800@7400)
10  0.011255 (0.0001)    arp reply svr4 is-at 0:0:c0:c2:9b:26
11  0.012562 (0.0013)    arp reply svr4 is-at 0:0:c0:c2:9b:26
12  0.013458 (0.0009)    arp reply svr4 is-at 0:0:c0:c2:9b:26
13  0.014526 (0.0011)    arp reply svr4 is-at 0:0:c0:c2:9b:26
14  0.015583 (0.0011)    arp reply svr4 is-at 0:0:c0:c2:9b:26
```

Figure 11.17 Packet exchange when an 8192-byte UDP datagram is sent on an Ethernet.

There are a few surprises in this output. First, six ARP requests are generated before the first ARP reply is returned. What we guess is happening is that IP generates the six fragments rapidly, and each one causes an ARP request.

Next, when the first ARP reply is received (line 7) only the last fragment is sent (line 9)! It appears that the first five fragments have been discarded. Indeed, this is the normal operation of ARP. Most implementations keep only the *last* packet sent to a given destination while waiting for an ARP reply.

> The Host Requirements RFC requires an implementation to prevent this type of *ARP flooding* (repeatedly sending an ARP request for the same IP address at a high rate). The recommended maximum rate is one per second. Here we see six ARP requests in 4.3 ms.
>
> The Host Requirements RFC states that ARP should save at least one packet, and this should be the latest packet. That's what we see here.

Another unexplained anomaly in this output is that `svr4` sends back seven ARP replies, not six.

The final point worth mentioning is that `tcpdump` was left to run for 5 minutes after the final ARP reply was returned, waiting to see if `svr4` sent back an ICMP "time exceeded during reassembly" error. The ICMP error was never sent. (We showed the format of this message in Figure 8.2. A *code* of 1 indicates that the time was exceeded during the reassembly of a datagram.)

The IP layer must start a timer when the first fragment of a datagram appears. Here "first" means the first arrival of any fragment for a given datagram, not the first fragment (with a fragment offset of 0). A normal timeout value is 30 or 60 seconds. If all the

fragments for this datagram have not arrived when the timer expires, all these fragments are discarded. If this were not done, fragments that never arrive (as we see in this example) could eventually cause the receiver to run out of buffers.

There are two reasons we don't see the ICMP message here. First, most Berkeley-derived implementations never generate this error! These implementations do set a timer, and do discard all fragments when the timer expires, but the ICMP error is never generated. Second, the first fragment—the one with an offset of 0 containing the UDP header—was never received. (It was the first of the five packets discarded by ARP.) An implementation is not required to generate the ICMP error unless this first fragment has been received. The reason is that the receiver of the ICMP error couldn't tell which user process sent the datagram that was discarded, because the transport layer header is not available. It's assumed that the upper layer (either TCP or the application using UDP) will eventually time out and retransmit.

In this section we've used IP fragmentation to see this interaction between UDP and ARP. We can also see this interaction if the sender quickly transmits multiple UDP datagrams. We chose to use fragmentation because the packets get generated quickly by IP, faster than multiple datagrams can be generated by a user process.

As unlikely as this example might seem, it occurs regularly. NFS sends UDP datagrams whose length just exceeds 8192 bytes. On an Ethernet these are fragmented as we've indicated, and if the appropriate ARP cache entry times out, you can see what we've shown here. NFS will time out and retransmit, but the first IP datagram can still be discarded because of ARP's limited queue.

11.10 Maximum UDP Datagram Size

Theoretically, the maximum size of an IP datagram is 65535 bytes, imposed by the 16-bit total length field in the IP header (Figure 3.1). With an IP header of 20 bytes and a UDP header of 8 bytes, this leaves a maximum of 65507 bytes of user data in a UDP datagram. Most implementations, however, provide less than this maximum.

There are two limits we can encounter. First the application program may be limited by its programming interface. The sockets API (Section 1.15) provides a function that the application can call to set the size of the receive buffer and the send buffer. For a UDP socket, this size is directly related to the maximum size UDP datagram the application can read or write. Most systems today provide a default of just over 8192 bytes for the maximum size of a UDP datagram that can be read or written. (This default is because 8192 is the amount of user data that NFS reads and writes by default.)

The next limitation comes from the kernel's implementation of TCP/IP. There may be implementation features (or bugs) that limit the size of an IP datagram to less than 65535 bytes.

> The author experimented with various UDP datagram sizes, using the `sock` program. Using the loopback interface under SunOS 4.1.3, the maximum size IP datagram was 32767 bytes. All higher values failed. But going across an Ethernet from BSD/386 to SunOS 4.1.3, the maximum size IP datagram the Sun could accept was 32786 (that is, 32758 bytes of user data). Using the loopback interface under Solaris 2.2, the maximum 65535-byte IP datagram could be sent and received. From Solaris 2.2 to AIX 3.2.2, the maximum 65535-byte IP datagram could be transferred. Obviously this limit depends on the source and destination implementations.

We mentioned in Section 3.2 that a host is required to receive at least a 576-byte IP datagram. Many UDP applications are designed to restrict their application data to 512 bytes or less, to stay below this limit. We saw this in Section 10.4, for example, where the Routing Information Protocol always sent less than 512 bytes of data per datagram. We'll encounter this same limit with other UDP applications: the DNS (Chapter 14), TFTP (Chapter 15), BOOTP (Chapter 16), and SNMP (Chapter 25).

Datagram Truncation

Just because IP is capable of sending and receiving a datagram of a given size doesn't mean the receiving application is prepared to read that size. UDP programming interfaces allow the application to specify the maximum number of bytes to return each time. What happens if the received datagram exceeds the size the application is prepared to deal with?

Unfortunately the answer depends on the programming interface and the implementation.

> The traditional Berkeley version of the sockets API truncates the datagram, discarding any excess data. Whether the application is notified depends on the version. (4.3BSD Reno and later can notify the application that the datagram was truncated.)
>
> The sockets API under SVR4 (including Solaris 2.x) does not truncate the datagram. Any excess data is returned in subsequent reads. The application is not notified that multiple reads are being fulfilled from a single UDP datagram.
>
> The TLI API does not discard the data. Instead a flag is returned indicating that more data is available, and subsequent reads by the application return the rest of the datagram.

When we discuss TCP we'll see that it provides a continuous stream of bytes to the application, without any message boundaries. TCP passes the data to the application in whatever size reads the application asks for—there is never any data loss across this interface.

11.11 ICMP Source Quench Error

Using UDP we are also able to generate the ICMP "source quench" error. This is an error that may be generated by a system (router or host) when it receives datagrams at a rate that is too fast to be processed. Note the qualifier "may." A system is not required to send a source quench, even if it runs out of buffers and throws datagrams away.

Figure 11.18 shows the format of the ICMP source quench error. We have a perfect scenario with our test network for generating this error. We can send datagrams from `bsdi` to the router `sun` across the Ethernet that must be routed across the dialup SLIP link. Since the SLIP link is about 1000 times slower than the Ethernet, we should easily be able to overrun its buffer space. The following command sends 100 1024-byte datagrams from the host `bsdi` through the router `sun` to `solaris`. We send the datagrams to the standard discard service, where they'll be ignored:

```
bsdi % sock -u -i -w1024 -n100 solaris discard
```

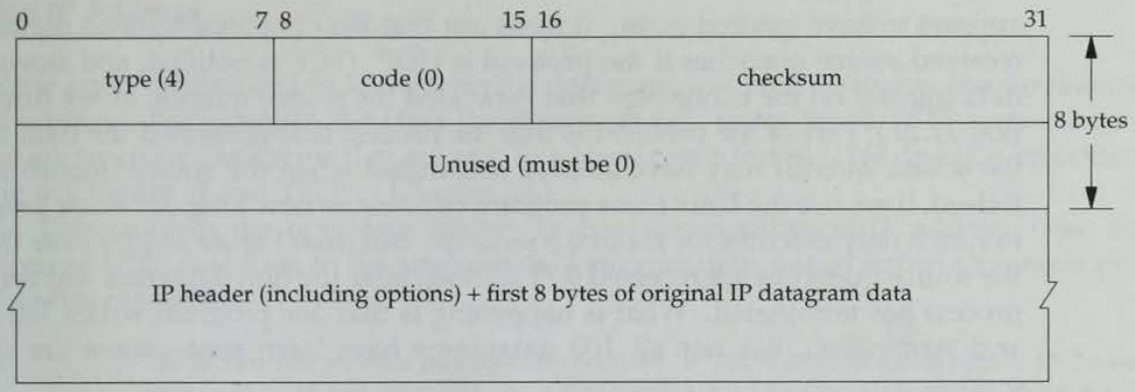


Figure 11.18 ICMP source quench error.

Figure 11.19 shows the `tcpdump` output corresponding to this command.

```
 1  0.0                    bsdi.1403 > solaris.discard: udp 1024
                                  26 lines that we don't show
27  0.10 (0.00)            bsdi.1403 > solaris.discard: udp 1024
28  0.11 (0.01)            sun > bsdi: icmp: source quench
29  0.11 (0.00)            bsdi.1403 > solaris.discard: udp 1024
30  0.11 (0.00)            sun > bsdi: icmp: source quench
                                  142 lines that we don't show
173  0.71 (0.06)           bsdi.1403 > solaris.discard: udp 1024
174  0.71 (0.00)           sun > bsdi: icmp: source quench
```

Figure 11.19 ICMP source quench from the router `sun`.

We have removed lots of lines from this output; there is a pattern. The first 26 datagrams are received without an error; we show the output only for the first. Starting with our 27th datagram, however, every time we send a datagram, we receive a source quench in return. There are a total of $26 + (74 \times 2) = 174$ lines of output.

From our serial line throughput calculations in Section 2.10, it takes just over 1 second to transfer a 1024-byte datagram at 9600 bits/sec. (In our example it should take longer than this since the $20 + 8 + 1024$ byte datagram will be fragmented because the MTU of the SLIP link from `sun` to `netb` is 552 bytes.) But we can see from the timing in Figure 11.19 that the router `sun` receives all 100 datagrams in less than 1 second, before the first one is through the SLIP link. It's not surprising that we used up many of its buffers.

> Although RFC 1009 [Braden and Postel 1987] requires a router to generate source quenches when it runs out of buffers, the new Router Requirements RFC [Almquist 1993] changes this and says that a router must not originate source quench errors. The current feeling is to deprecate the source quench error, since it consumes network bandwidth and is an ineffective and unfair fix for congestion.

Another point to make regarding this example is that our `sock` program either never received a notification that the source quenches were being received, or if it did, it

appears to have ignored them. It turns out that BSD implementations normally ignore received source quenches if the protocol is UDP. (TCP is notified, and slows down the data transfer on the connection that generated the source quench, as we discuss in Section 21.10.) Part of the problem is that the process that generated the data that caused the source quench may have already terminated when the source quench is received. Indeed, if we use the Unix `time` program to measure how long our `sock` program takes to run, it only executes for about 0.5 seconds. But from Figure 11.19 we see that some of the source quenches are received 0.71 seconds after the first datagram was sent, after the process has terminated. What is happening is that our program writes 100 datagrams and terminates. But not all 100 datagrams have been sent—some are queued for output.

This example reiterates that UDP is an unreliable protocol and illustrates the value of end-to-end flow control. Even though our `sock` program successfully wrote 100 datagrams to its network, only 26 were really sent to the destination. The other 74 were probably discarded by the intermediate router. Unless we build some form of acknowledgment into the application, the sender has no idea whether the receiver really got the data.

11.12 UDP Server Design

There are some implications in using UDP that affect the design and implementation of a server. The design and implementation of clients is usually easier than that of servers, which is why we talk about server design and not client design. Servers typically interact with the operating system and most servers need a way to handle multiple clients at the same time.

Normally a client starts, immediately communicates with a single server, and is done. Servers, on the other hand, start and then go to sleep, waiting for a client's request to arrive. In the case of UDP, the server wakes up when a client's datagram arrives, probably containing a request message of some form from the client.

Our interest here is not in the programming aspects of clients and servers ([Stevens 1990] covers all those details), but in the protocol features of UDP that affect the design and implementation of a server using UDP. (We examine the details of TCP server design in Section 18.11.) Although some of the features we describe depend on the implementation of UDP being used, the features are common to most implementations.

Client IP Address and Port Number

What arrives from the client is a UDP datagram. The IP header contains the source and destination IP addresses, and the UDP header contains the source and destination UDP port numbers. When an application receives a UDP datagram, it must be told by the operating system who sent the message—the source IP address and port number.

This feature allows an iterative UDP server to handle multiple clients. Each reply is sent back to the client that sent the request.

Destination IP Address

Some applications need to know who the datagram was sent to, that is, the destination IP address. For example, the Host Requirements RFC states that a TFTP server should ignore received datagrams that are sent to a broadcast address. (We describe broadcasting in Chapter 12 and TFTP in Chapter 15.)

This requires the operating system to pass the destination IP address from the received UDP datagram to the application. Unfortunately, not all implementations provide this capability.

> The sockets API provides this capability with the `IP_RECVDSTADDR` socket option. Of the systems used in the text, only BSD/386, 4.4BSD, and AIX 3.2.2 support this option. SVR4, SunOS 4.x, and Solaris 2.x don't support it.

UDP Input Queue

We said in Section 1.8 that most UDP servers are iterative servers. This means a single server process handles all the client requests on a single UDP port (the server's well-known port).

Normally there is a limited size input queue associated with each UDP port that a application is using. This means that requests that arrive at about the same time from different clients are automatically queued by UDP. The received UDP datagrams are passed to the application (when it asks for the next one) in the order they were received.

It is possible, however, for this queue to overflow, causing the kernel's UDP module to discard incoming datagrams. We can see this with the following experiment. We start our `sock` program on the host `bsdi` running as a UDP server:

```
bsdi % sock -s -u -v -E -R256 -r256 -P30 6666
from 140.252.13.33, to 140.252.13.63: 1111111111       from sun, to broadcast address
from 140.252.13.34, to 140.252.13.35: 4444444444444    from svr4, to unicast address
```

We specify the following flags: `-s` to run as a server, `-u` for UDP, `-v` to print the client's IP address, and `-E` to print the destination IP address (which is supported by this system). Additionally we set the UDP receive buffer for this port to 256 bytes (`-R`), along with the size of each application read (`-r`). The flag `-P30` tells it to pause for 30 seconds after creating the UDP port, before reading the first datagram. This gives us time to start the clients on two other hosts, send some datagrams, and see how the receive queueing works.

Once the server is started, and is in its 30-second pause, we start one client on the host `sun` and send three datagrams:

```
sun % sock -u -v 140.252.13.63 6666          to Ethernet broadcast address
connected on 140.252.13.33.1252 to 140.252.13.63.6666
1111111111                    11 bytes of data (with newline)
222222222                     10 bytes of data (with newline)
33333333333                   12 bytes of data (with newline)
```

The destination address is the broadcast address (140.252.13.63). We also start a second client on the host `svr4` and send another three datagrams:

```
svr4 % sock -u -v bsdi 6666
connected on 0.0.0.0.1042 to 140.252.13.35.6666
4444444444444                    14 bytes of data (with newline)
555555555555555                  16 bytes of data (with newline)
66666666                         9 bytes of data (with newline)
```

The first thing we notice in the interactive output shown earlier on `bsdi` is that only two datagrams were received by the application: the first one from `sun` with all 1s, and the first one from `svr4` with all 4s. The other four datagrams appear to have been thrown away.

The `tcpdump` output in Figure 11.20 shows that all six datagrams were delivered to the destination host. The datagrams were typed on the two clients in alternating order: first from `sun`, then from `svr4`, and so on. We can also see that all six were delivered in about 12 seconds, within the 30-second period while the server was sleeping.

```
1   0.0                 sun.1252 > 140.252.13.63.6666: udp 11
2   2.499184 (2.4992)   svr4.1042 > bsdi.6666: udp 14
3   4.959166 (2.4600)   sun.1252 > 140.252.13.63.6666: udp 10
4   7.607149 (2.6480)   svr4.1042 > bsdi.6666: udp 16
5  10.079059 (2.4719)   sun.1252 > 140.252.13.63.6666: udp 12
6  12.415943 (2.3369)   svr4.1042 > bsdi.6666: udp 9
```

Figure 11.20 `tcpdump` for UDP datagrams sent by two clients.

We can also see the server's `-E` option lets it know the destination IP address of each datagram. If it wanted to, it could choose what to do with the first datagram it receives, which was sent to a broadcast address.

We can see several points in this example. First, the application is not told when its input queue overflows. The excess datagrams are just discarded by UDP. Also, from the `tcpdump` output we see that nothing is sent back to the client to tell it that its datagram was discarded. There is nothing like an ICMP source quench sent back to the sender. Finally, it appears that the UDP input queue is FIFO (first-in, first-out), whereas we saw that the ARP input queue in Section 11.9 was LIFO (last-in, first-out).

Restricting Local IP Address

Most UDP servers *wildcard* their local IP address when they create a UDP end point. This means that an incoming UDP datagram destined for the server's port will be accepted on any local interface. For example, we can start a UDP server on port 7777:

```
sun % sock -u -s 7777
```

We then use the `netstat` command to see the state of the end point:

```
sun % netstat -a -n -f inet
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
udp        0      0  *.7777                 *.*
```

We have deleted many lines of output other than the one in which we're interested. The `-a` flag reports on all network end points. The `-n` flag prints IP addresses as dotted-

decimal numbers, instead of trying to use the DNS to convert the address to a name, and prints numeric port numbers instead of service names. The `-f inet` option reports only TCP and UDP end points.

The local address is printed as `*.7777` where the asterisk means the local IP address has been wildcarded.

When the server creates its end point it can specify one of the host's local IP addresses, including one of its broadcast addresses, as the local IP address for the end point. Incoming UDP datagrams will then be passed to this end point only if the destination IP address matches the specified local address. With our `sock` program, if we specify an IP address before the port number, that IP address becomes the local IP address for the end point. For example,

```
sun % sock -u -s 140.252.1.29 7777
```

restricts the server to datagrams arriving on the SLIP interface (140.252.1.29). The `netstat` output shows this:

```
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
udp        0      0  140.252.1.29.7777      *.*
```

If we try to send this server a datagram from a host on the Ethernet, `bsdi` at address 140.252.13.35, an ICMP port unreachable is returned. The server never sees the datagram. Figure 11.21 shows this scenario.

```
1  0.0                  bsdi.1723 > sun.7777: udp 13
2  0.000822 (0.0008)    sun > bsdi: icmp: sun udp port 7777 unreachable
```

Figure 11.21 Rejection of UDP datagram caused by server's local address binding.

It is possible to start different servers at the same port, each with a different local IP address. Normally, however, the system must be told by the application that it is OK to reuse the same port number.

> With the sockets API the `SO_REUSEADDR` socket option must be specified. This is done by our `sock` program by specifying the `-A` option.

On our host `sun` we can start five different servers on the same UDP port (8888):

```
sun % sock -u -s 140.252.1.29 8888          for SLIP link
sun % sock -u -s -A 140.252.13.33 8888      for Ethernet
sun % sock -u -s -A 127.0.0.1 8888          for loopback interface
sun % sock -u -s -A 140.252.13.63 8888      for Ethernet broadcasts
sun % sock -u -s -A 8888                    everything else (wildcard IP address)
```

All except the first of the servers must be started with the `-A` flag, telling the system that it's OK to reuse the same port number. The `netstat` output shows the five servers:

```
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
udp        0      0  *.8888                 *.*
udp        0      0  140.252.13.63.8888     *.*
udp        0      0  127.0.0.1.8888         *.*
udp        0      0  140.252.13.33.8888     *.*
udp        0      0  140.252.1.29.8888      *.*
```

In this scenario, the only datagrams that will go to the server with the wildcarded local IP address are those destined to 140.252.1.255, because the other four servers cover all other possibilities.

There is a priority implied when an end point with a wildcard address exists. An end point with a specific IP address that matches the destination IP address is always chosen over a wildcard. The wildcard end point is used only when a specific match is not found.

Restricting Foreign IP Address

In all the `netstat` output that we showed earlier, the foreign IP address and foreign port number are shown as `*.*` meaning the end point will accept an incoming UDP datagram from any IP address and any port number. Most implementations allow a UDP end point to restrict the foreign address.

This means the end point will only receive UDP datagrams from that specific IP address and port number. Our `sock` program uses the `-f` option to specify the foreign IP address and port number:

```
sun % sock -u -s -f 140.252.13.35.4444 5555
```

This sets the foreign IP address to 140.252.13.35 (our host `bsdi`) and the foreign port number to 4444. The server's well-known port is 5555. If we run `netstat` we see that the local IP address has also been set, even though we didn't specify it:

```
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
udp        0      0  140.252.13.33.5555    140.252.13.35.4444
```

This is a side effect of specifying the foreign IP address and foreign port on Berkeley-derived systems: if the local address has not been chosen when the foreign address is specified, the local address is chosen automatically. Its value becomes the IP address of the interface chosen by IP routing to reach the specified foreign IP address. Indeed, in this example the IP address on `sun` for the Ethernet that is connected to the foreign address is 140.252.13.33.

Figure 11.22 summarizes the three types of address bindings that a UDP server can establish for itself.

Local Address	Foreign Address	Description
localIP.lport	*foreignIP.fport*	restricted to one client
localIP.lport	*.*	restricted to datagrams arriving on one local interface: *localIP*
*.*lport*	*.*	receives all datagrams sent to *lport*

Figure 11.22 Specification of local and foreign IP addresses and port number for UDP server.

In all cases, *lport* is the server's well-known port and *localIP* must be the IP address of a local interface. The ordering of the three rows in the table is the order that the UDP module applies when trying to determine which local end point receives an incoming datagram. The most specific binding (the first row) is tried first, and the least specific (the last row with both IP addresses wildcarded) is tried last.

Multiple Recipients per Port

Although it's not specified in the RFCs, most implementations allow only one application end point at a time to be associated with any one local IP address and UDP port number. When a UDP datagram arrives at a host destined for that IP address and port number, one copy is delivered to that single end point. The IP address of the end point can be the wildcard, as shown earlier.

For example, under SunOS 4.1.3 we start one server on port 9999 with a wildcarded local IP address:

```
sun % sock -u -s 9999
```

If we then try to start another server with the same wildcarded local address and the same port, it doesn't work, even if we specify the `-A` option:

```
sun % sock -u -s 9999                    we expect this to fail
can't bind local address: Address already in use

sun % sock -u -s -A 9999                 so we try -A flag this time
can't bind local address: Address already in use
```

On systems that support multicasting (Chapter 12), this changes. Multiple end points can use the same local IP address and UDP port number, although the application normally must tell the API that this is OK (i.e., our `-A` flag to specify the `SO_REUSEADDR` socket option).

> 4.4BSD, which supports multicasting, requires the application to set a different socket option (`SO_REUSEPORT`) to allow multiple end points to share the same port. Furthermore each end point must specify this option, including the first one to use the port.

When a UDP datagram arrives whose destination IP address is a broadcast or multicast address, and there are multiple end points at the destination IP address and port number, one copy of the incoming datagram is passed to *each* end point. (The end point's local IP address can be the wildcard, which matches any destination IP address.) But if a UDP datagram arrives whose destination IP address is a unicast address, only a single copy of the datagram is delivered to *one* of the end points. Which end point gets the unicast datagram is implementation dependent.

11.13 Summary

UDP is a simple protocol. Its official specification, RFC 768 [Postel 1980], requires only three pages. The services it provides to a user process, above and beyond IP, are port numbers and an optional checksum. We used UDP to examine this checksum and to see how fragmentation is performed.

We then examined the ICMP unreachable error that is part of the new path MTU discovery feature (Section 2.9). We watched path MTU discovery using Traceroute and UDP. We also looked at the interaction between UDP and ARP whereby most ARP implementations only retain the most recently transmitted datagram to a given destination, while waiting for an ARP reply.

The ICMP source quench error can be sent by a system that is receiving IP datagrams faster than they can be processed. It is easy to generate these ICMP errors using UDP.

Exercises

11.1 In Section 11.5 we caused fragmentation on an Ethernet by writing a UDP datagram with 1473 bytes of user data. What is the smallest amount of user data that causes fragmentation on an Ethernet if IEEE 802 encapsulation (Section 2.2) is used instead?

11.2 Read RFC 791 [Postel 1981a] to determine why all fragments other than the last must have a length that is a multiple of 8 bytes.

11.3 Assume an Ethernet and a UDP datagram with 8192 bytes of user data. How many fragments are transmitted and what is the offset and length of each fragment?

11.4 Continue the previous exercise, assuming these fragments then traverse a SLIP link with an MTU of 552. You also need to remember that the amount of data in each fragment (i.e., everything other than the IP header) must be a multiple of 8 bytes. How many fragments are transmitted and what is the offset and length of each fragment?

11.5 An application using UDP sends a datagram that gets fragmented into four pieces. Assume that fragments 1 and 2 make it to the destination, with fragments 3 and 4 being lost. The application then times out and retransmits the UDP datagram 10 seconds later and this datagram is fragmented identically to the first transmission (i.e., same offsets and lengths). Assume that this time fragments 1 and 2 are lost but fragments 3 and 4 make it to the destination. Also assume that the reassembly timer on the receiving host is 60 seconds, so when fragments 3 and 4 of the retransmission make it to the destination, fragments 1 and 2 from the first transmission have not been discarded. Can the receiver reassemble the IP datagram from the four fragments it now has?

11.6 How do you know that the fragments in Figure 11.15 really correspond to lines 5 and 6 in Figure 11.14?

11.7 After the host `gemini` had been up for 33 days, the `netstat` program showed that 129 IP datagrams out of 48 million had been dropped because of a bad header checksum, and 20 TCP segments out of 30 million had been dropped because of a bad TCP checksum. Not a single UDP datagram was dropped, however, because of a UDP checksum error, out of the approximately 18 million UDP datagrams. Give two reasons why. (*Hint*: See Figure 11.4.)

11.8 In our discussion of fragmentation we never said what happens to IP options in the IP header—are they copied as part of the IP header in each fragment, or left in the first fragment only? We've described the following IP options: record route (Section 7.3), timestamp (Section 7.4), strict and loose source routing (Section 8.5). How would you expect fragmentation to handle these options? Check your answer with RFC 791.

11.9 In Figure 1.8 (p. 11) we said that incoming UDP datagrams are demultiplexed based on the destination UDP port number. Is that correct?

12

Broadcasting and Multicasting

12.1 Introduction

We mentioned in Chapter 1 that there are three kinds of IP addresses: *unicast*, *broadcast*, and *multicast*. In this chapter we discuss broadcasting and multicasting in more detail.

Broadcasting and multicasting only apply to UDP, where it makes sense for an application to send a single message to multiple recipients. TCP is a connection-oriented protocol that implies a connection between two hosts (specified by IP addresses) and one process on each host (specified by port numbers).

Consider a set of hosts on a shared network such as an Ethernet. Each Ethernet frame contains the source and destination Ethernet addresses (48-bit values). Normally each Ethernet frame is destined for a single host. The destination address specifies a single interface—called a *unicast*. In this way communication between any two hosts doesn't bother any of the remaining hosts on the cable (except for possible contention for the shared media).

There are times, however, when a host wants to send a frame to every other host on the cable—called a *broadcast*. We saw this with ARP and RARP. Multicasting fits between unicasting and broadcasting: the frame should be delivered to a set of hosts that belong to a multicast group.

To understand broadcasting and multicasting we need to understand that filtering takes place on each host, each time a frame passes by on the cable. Figure 12.1 shows a picture of this.

First, the interface card sees every frame that passes by on the cable and makes a decision whether to receive the frame and deliver it to the device driver. Normally the interface card receives only those frames whose destination address is either the hardware address of the interface or the broadcast address. Additionally, most interfaces can be placed into a promiscuous mode whereby they receive a copy of every frame. This mode is used by `tcpdump`, for example.

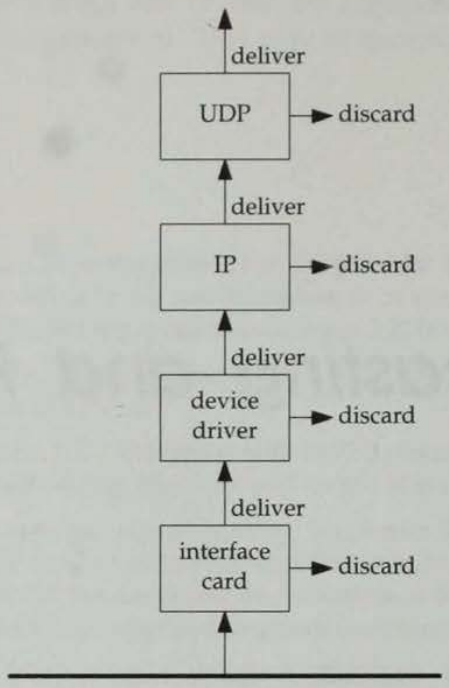


Figure 12.1 Filtering that takes place up the protocol stack when a frame is received.

Today most interfaces can also be configured to receive frames whose destination address is a multicast address, or some subset of multicast addresses. On an Ethernet, a multicast address has the low-order bit of the high-order byte turned on. In hexadecimal this bit looks like `01:00:00:00:00:00`. (We can consider the Ethernet broadcast address, `ff:ff:ff:ff:ff:ff` as a special case of the Ethernet multicast address.)

If the interface card receives the frame, it is passed to the device driver. (One reason the interface card might discard the frame is if the Ethernet checksum is incorrect.) Additional filtering is performed by the device driver. First, the frame type must specify a protocol that is supported (IP, ARP, etc.). Second, additional multicast filtering may be performed, to check whether the host belongs to the addressed multicast group.

The device driver then passes the frame to the next layer, such as IP, if the frame type specifies an IP datagram. IP performs more filtering, based on the source and destination IP addresses, and passes the datagram up to the next layer (such as TCP or UDP) if all is well.

Each time UDP receives a datagram from IP, it performs filtering based on the destination port number, and sometimes the source port number too. If no process is currently using the destination port number, the datagram is discarded and an ICMP port unreachable message is normally generated. (TCP performs similar filtering based on its port numbers.) If the UDP datagram has a checksum error, UDP silently discards it.

The problem with broadcasting is the processing load that it places on hosts that aren't interested in the broadcasts. Consider an application that is designed to use UDP

broadcasts. If there are 50 hosts on the cable, but only 20 are participating in the application, every time one of the 20 sends a UDP broadcast, the other 30 hosts have to process the broadcast, all the way up through the UDP layer, before the UDP datagram is discarded. The UDP datagram is discarded by these 30 hosts because the destination port number is not in use.

The intent of multicasting is to reduce this load on hosts with no interest in the application. With multicasting a host specifically joins one or more multicast groups. If possible, the interface card is told which multicast groups the host belongs to, and only those multicast frames are received.

12.2 Broadcasting

In Figure 3.9 we showed four different forms of IP broadcast addresses. We now describe them in more detail.

Limited Broadcast

The *limited broadcast address* is 255.255.255.255. This can be used as the destination address of an IP datagram during the host configuration process, when the host might not know its subnet mask or even its IP address.

A datagram destined for the limited broadcast address is *never* forwarded by a router under any circumstance. It only appears on the local cable.

An unanswered question is: if a host is multihomed and a process sends a datagram to the limited broadcast address, should the datagram be sent out each connected interface that supports broadcasting? If not, an application that wants to broadcast out all interfaces must determine all the interfaces on the host that support broadcasting, and send a copy out each interface.

Most BSD systems treat 255.255.255.255 as an alias for the broadcast address of the first interface that was configured, and don't provide any way to send a datagram out all attached, broadcast-capable interfaces. Indeed, two applications that send UDP datagrams out every interface are `routed` (Section 10.3) and `rwhod` (the server for the BSD `rwho` client). Both of these applications go through a similar start-up procedure to determine all the interfaces on the host, and which ones are capable of broadcasting. The net-directed broadcast address corresponding to that interface is then used as the destination address for datagrams sent out the interface.

> The Host Requirements RFC takes no stand on the issue of whether a multihomed host should send a limited broadcast out all its interfaces.

Net-directed Broadcast

The *net-directed broadcast address* has a host ID of all one bits. A class A net-directed broadcast address is *netid*.255.255.255, where *netid* is the class A network ID.

A router must forward a net-directed broadcast, but it must also have an option to disable this forwarding.

Subnet-directed Broadcast

The *subnet-directed broadcast address* has a host ID of all one bits but a specific subnet ID. Classification of an IP address as a subnet-directed broadcast address requires knowledge of the subnet mask. For example, if a router receives a datagram destined for 128.1.2.255, this is a subnet-directed broadcast if the class B network 128.1 has a subnet mask of 255.255.255.0, but it is not a broadcast if the subnet mask is 255.255.254.0 (`0xfffffe00`).

All-subnets-directed Broadcast

An *all-subnets-directed broadcast address* also requires knowledge of the destination network's subnet mask, to differentiate this broadcast address from a net-directed broadcast address. Both the subnet ID and the host ID are all one bits. For example, if the destination's subnet mask is 255.255.255.0, then the IP address 128.1.255.255 is an all-subnets-directed broadcast. But if the network is not subnetted, then this is a net-directed broadcast.

Current feeling [Almquist 1993] is that this type of broadcast is obsolete. It is better to use multicasting than an all-subnets-directed broadcast.

> [Almquist 1993] notes that RFC 922 requires that an all-subnets-directed broadcast be sent to all subnets, but no current routers do so. This is fortunate since a host that has been misconfigured without its subnet mask sends all its "local" broadcasts to all subnets. For example, if the host with IP address 128.1.2.3 doesn't set a subnet mask, then its broadcast address normally defaults to 128.1.255.255. But if the subnet mask should have been set to 255.255.255.0, then broadcasts from this misconfigured host appear directed to all subnets.
>
> The first widespread implementation of TCP/IP, the 4.2BSD system in 1983, used a host ID of all zero bits for the broadcast address. One of the earliest references to the broadcast IP address is IEN 212 [Gurwitz and Hinden 1982], and it *proposed* to define the IP broadcast address as a host ID of one bits. (IENs are the *Internet Experiment Notes*, basically predecessors to the RFCs.) RFC 894 [Hornig 1984] commented that 4.2BSD used a nonstandard broadcast address, but RFC 906 [Finlayson 1984] noted that there was no Internet standard for the broadcast address. The RFC editor added a footnote to RFC 906 acknowledging the lack of a standard broadcast address, but strongly recommended that a host ID of all one bits be used as the broadcast address. Although Berkeley adopted the use of all one bits for the broadcast address with 4.3BSD in 1986, some operating systems (notably SunOS 4.x) continued to use the nonstandard broadcast address through the early 1990s.

12.3 Broadcasting Examples

How are broadcasts sent and what do routers and hosts do with broadcasts? Unfortunately this is a hard question to answer because it depends on the type of broadcast address, the application, the TCP/IP implementation, and possible configuration switches.

First, the application must support broadcasting. If we execute

```
sun % ping 255.255.255.255
/usr/etc/ping: unknown host 255.255.255.255
```

intending to send a broadcast on the local cable, it doesn't work. But the problem here is a programming problem in the application (`ping`). Most applications that accept either a dotted-decimal IP address or a hostname call the function `inet_addr`(3) to convert the dotted-decimal character string to its 32-bit binary IP address, and if this fails, assume the character string is a hostname. Unfortunately this library function returns –1 to indicate an error (such as a character other than a digit or decimal point in the string), but the limited broadcast address (255.255.255.255) also converts into –1. Most programs then assume that the character string is a hostname, look it up using the DNS (Chapter 14), and end up printing an error such as "unknown host."

If we fix this programming shortfall in the `ping` program, however, the results are often not what we expect. On six different systems tested by the author, only one handled this as expected and generated a broadcast packet on the local cable. Most looked up the IP address 255.255.255.255 in the routing table, applied the default route, and sent a unicast packet to the default router. Eventually the packet was thrown away.

A subnet-directed broadcast is what we should be using. Indeed, in Section 6.3 we sent datagrams to the IP address 140.252.13.63 for the bottom Ethernet in our test network (inside front cover), and got replies from all the hosts on the Ethernet. The subnet-directed broadcast address associated with each interface is the value used with the `ifconfig` command (Section 3.8). If we ping that address, the result is what we expect:

```
sun % arp -a                                    ARP cache is empty

sun % ping 140.252.13.63
PING 140.252.13.63: 56 data bytes
64 bytes from sun (140.252.13.33): icmp_seq=0. time=4. ms
64 bytes from bsdi (140.252.13.35): icmp_seq=0. time=172. ms
64 bytes from svr4 (140.252.13.34): icmp_seq=0. time=192. ms

64 bytes from sun (140.252.13.33): icmp_seq=1. time=1. ms
64 bytes from bsdi (140.252.13.35): icmp_seq=1. time=52. ms
64 bytes from svr4 (140.252.13.34): icmp_seq=1. time=90. ms

^?                                              type interrupt key to stop
----140.252.13.63 PING Statistics----
2 packets transmitted, 6 packets received, -200% packet loss
round-trip (ms)  min/avg/max = 1/85/192

sun % arp -a                                    check ARP cache again
svr4 (140.252.13.34) at 0:0:c0:c2:9b:26
bsdi (140.252.13.35) at 0:0:c0:6f:2d:40
```

IP looks at the destination address (140.252.13.63), determines that it is the subnet-directed broadcast address, and sends the datagram to the link-layer broadcast address.

We mentioned in Section 6.3 that this type of broadcast means all the hosts on the local network, including the sender. We see here that we do get a reply from the sending host (`sun`) in addition to the other hosts on the cable.

In this example we've also shown the ARP cache before and after the `ping` of the broadcast address. This is to show the interaction between broadcasting and ARP. The ARP cache is empty before we execute `ping`, but full afterward. (That is, there is one entry for every other host on the cable that responded to the echo request.) How did

this happen when we said that the Ethernet frame is sent to the link-layer broadcast address (0xffffffff)? The sending of these frames by sun does not require ARP.

If we watch ping using tcpdump, we see that it is the recipients of the broadcast frames that generate an ARP request to sun, before they can send their reply. This is because the reply is unicast. We said in Section 4.5 that the receiver of an ARP request (sun in this example) normally adds the requestor's IP address and hardware address to its ARP cache, in addition to sending an ARP reply. This is on the assumption that if the requestor is about to send us a packet, we'll probably want to send something back.

Our use of ping is somewhat special because the type of programming interface that it uses (called "raw sockets" on most Unix implementations) always allows a datagram to be sent to the broadcast address. What if we use an application that was not designed to support broadcasting, such as TFTP? (We cover TFTP in more detail in Chapter 15.)

```
bsdi % tftp                              start the client
tftp> connect 140.252.13.63              specify the IP address of the server
tftp> get temp.foo                       and try to fetch a file from the server
tftp: sendto: Permission denied
tftp> quit                               terminate the client
```

Here we get an error immediately, and nothing is sent on the cable. What's happening here is that the sockets API doesn't allow a process to send a UDP datagram to the broadcast address unless the process specifically states that it plans to broadcast. This is intended to prevent users from mistakenly specifying a broadcast address (as we did here) when the application was never intended to broadcast.

> With the sockets API the application must set the SO_BROADCAST socket option before sending a UDP datagram to a broadcast address.
>
> Not all systems enforce this restriction. Some implementations allow any process to broadcast UDP datagrams, without requiring the process to say so. Others are more restrictive and require a process to have superuser privileges to broadcast.

The next question is whether directed broadcasts are forwarded or not. Some kernels and routers have an option to enable or disable this feature. (See Appendix E.)

If we enable this feature on our router bsdi and run ping from the host slip, we can see if the subnet-directed broadcasts are forwarded by bsdi. Forwarding a directed broadcast means the router takes the incoming unicast datagram, determines that the destination address is the directed broadcast for one of its interfaces, and then forwards the datagram onto the appropriate network using a link-layer broadcast.

```
slip % ping 140.252.13.63
PING 140.252.13.63 (140.252.13.63): 56 data bytes
64 bytes from 140.252.13.35: icmp_seq=0 ttl=255 time=190 ms
64 bytes from 140.252.13.33: icmp_seq=0 ttl=254 time=280 ms (DUP!)
64 bytes from 140.252.13.34: icmp_seq=0 ttl=254 time=360 ms (DUP!)

64 bytes from 140.252.13.35: icmp_seq=1 ttl=255 time=180 ms
64 bytes from 140.252.13.33: icmp_seq=1 ttl=254 time=270 ms (DUP!)
64 bytes from 140.252.13.34: icmp_seq=1 ttl=254 time=360 ms (DUP!)

^?                                       type interrupt key to stop
```

```
--- 140.252.13.63 ping statistics ---
3 packets transmitted, 2 packets received, +4 duplicates, 33% packet loss
round-trip min/avg/max = 180/273/360 ms
```

We see that this does indeed work. We also see that the BSD `ping` program checks for duplicate sequence numbers and prints `DUP!` when this occurs. It normally means a packet was duplicated somewhere, but here we expect to see this, since we sent the requests to a broadcast address.

We can also run this test from a host much farther away from the network to which the broadcast is directed. If we run `ping` from the host `vangogh.cs.berkeley.edu` (14 hops away from our network), it still works if the router `sun` is configured to forward directed broadcasts. In this case the IP datagrams (carrying the ICMP echo requests) are forwarded by every router in the path as a normal datagram. None of them knows that it's really a directed broadcast. The next to last router, `netb`, thinks it's for the host with an ID of 63, and forwards it to `sun`. It is the router `sun` that detects that the destination IP address is really the broadcast address of an attached interface, and turns the datagram into a link-layer broadcast on that network.

Broadcasting is a feature that should be used with great care. In many cases IP multicasting will prove to be a better solution.

12.4 Multicasting

IP multicasting provides two services for an application.

1. Delivery to multiple destinations. There are many applications that deliver information to multiple recipients: interactive conferencing and dissemination of mail or news to multiple recipients, for example. Without multicasting these types of services tend to use TCP today (delivering a separate copy to each destination). Even with multicasting, some of these applications might continue to use TCP for its reliability.
2. Solicitation of servers by clients. A diskless workstation, for example, needs to locate a bootstrap server. Today this is provided using a broadcast (as we'll see with BOOTP in Chapter 16), but a multicast solution would impose less overhead on the hosts that don't provide the service.

In this section we'll take a look at multicast addresses, and the next chapter looks at the protocol used by multicasting hosts and routers (IGMP).

Multicast Group Addresses

Figure 12.2 shows the format of a class D IP address.

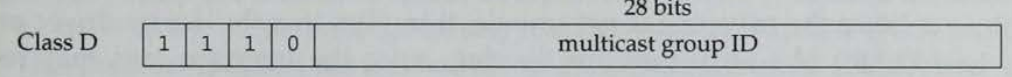


Figure 12.2 Format of a class D IP address.

Unlike the other three classes of IP addresses (A, B, and C), which we showed in Figure 1.5, the 28 bits allocated for the multicast group ID have no further structure.

A *multicast group address* is the combination of the high-order 4 bits of 1110 and the multicast group ID. These are normally written as dotted-decimal numbers and are in the range 224.0.0.0 through 239.255.255.255.

The set of hosts listening to a particular IP multicast address is called a *host group*. A host group can span multiple networks. Membership in a host group is dynamic—hosts may join and leave host groups at will. There is no restriction on the number of hosts in a group, and a host does not have to belong to a group to send a message to that group.

Some multicast group addresses are assigned as well-known addresses by the IANA (Internet Assigned Numbers Authority). These are called *permanent host groups*. This is similar to the well-known TCP and UDP port numbers. Similarly, these well-known multicast addresses are listed in the latest Assigned Numbers RFC. Notice that it is the multicast address of the group that is permanent, not the membership of the group.

For example, 224.0.0.1 means "all systems on this subnet," and 224.0.0.2 means "all routers on this subnet." The multicast address 224.0.1.1 is for NTP, the Network Time Protocol, 224.0.0.9 is for RIP-2 (Section 10.5), and 224.0.1.2 is for SGI's (Silicon Graphics) dogfight application.

Converting Multicast Group Addresses to Ethernet Addresses

The IANA owns an Ethernet address block, which in hexadecimal is `00:00:5e`. This is the high-order 24 bits of the Ethernet address, meaning that this block includes addresses in the range `00:00:5e:00:00:00` through `00:00:5e:ff:ff:ff`. The IANA allocates half of this block for multicast addresses. Given that the first byte of any Ethernet address must be `01` to specify a multicast address, this means the Ethernet addresses corresponding to IP multicasting are in the range `01:00:5e:00:00:00` through `01:00:5e:7f:ff:ff`.

> Our notation here uses the Internet standard bit order, for a CSMA/CD or token bus network, as the bits appear in memory. This is what most programmers and system administrators deal with. The IEEE documentation uses the transmission order of the bits. The Assigned Numbers RFC gives additional details on the differences between these representations.

This allocation allows for 23 bits in the Ethernet address to correspond to the IP multicast group ID. The mapping places the low-order 23 bits of the multicast group ID into these 23 bits of the Ethernet address. This is shown in Figure 12.3.

Since the upper 5 bits of the multicast group ID are ignored in this mapping, it is not unique. Thirty-two different multicast group IDs map to each Ethernet address. For example, the multicast addresses 224.128.64.32 (hex `e0.80.40.20`) and 224.0.64.32 (hex `e0.00.40.20`) both map into the Ethernet address `01:00:5e:00:40:20`.

Since the mapping is not unique, it implies that the device driver or the IP module in Figure 12.1 must perform filtering, since the interface card may receive multicast frames in which the host is really not interested. Also, if the interface card doesn't

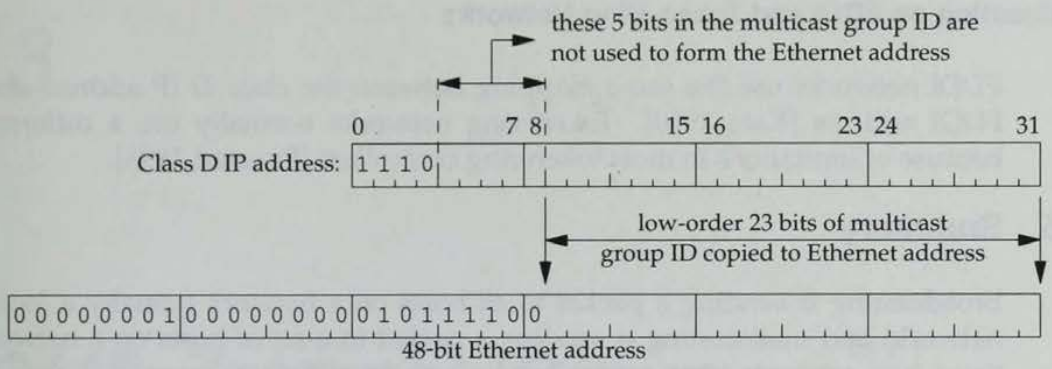


Figure 12.3 Mapping of a class D IP address into Ethernet multicast address.

provide adequate filtering of multicast frames, the device driver may have to receive all multicast frames, and perform the filtering itself.

> LAN interface cards tend to come in two varieties. One type performs multicast filtering based on the hash value of the multicast hardware address, which means some unwanted frames can always get through. The other type has a small, fixed number of multicast addresses to listen for, meaning that when the host needs to receive more multicast addresses than are supported, the interface must be put into a "multicast promiscuous" mode. Hence, both types of interfaces still require that the device driver perform checking that the received frame is really wanted.
>
> Even if the interface performs perfect multicast filtering (based on the 48-bit hardware address), since the mapping from a class D IP address to a 48-bit hardware address is not one-to-one, filtering is still required.
>
> Despite this imperfect address mapping and hardware filtering, multicasting is still better than broadcasting.

Multicasting on a single physical network is simple. The sending process specifies a destination IP address that is a multicast address, the device driver converts this to the corresponding Ethernet address, and sends it. The receiving processes must notify their IP layers that they want to receive datagrams destined for a given multicast address, and the device driver must somehow enable reception of these multicast frames. This is called "joining a multicast group." (The reason we use the plural "receiving processes" is because there are normally multiple receivers for a given multicast message, either on the same host or on multiple hosts, which is why we're using multicasting in the first place.) When a multicast datagram is received by a host, it must deliver a copy to all the processes that belong to that multicast group. This is different from UDP where a single process receives an incoming unicast UDP datagram. With multicasting it is possible for multiple processes on a given host to belong to the same multicast group.

But complications arise when we extend multicasting beyond a single physical network and pass multicast packets through routers. A protocol is needed for multicast routers to know if any hosts on a given physical network belong to a given multicast group. This protocol is called the *Internet Group Management Protocol* (IGMP) and is the topic of the next chapter.

Multicasting on FDDI and Token Ring Networks

FDDI networks use the same mapping between the class D IP address and the 48-bit FDDI address [Katz 1990]. Token ring networks normally use a different mapping, because of limitations in most token ring controllers [Pusateri 1993].

12.5 Summary

Broadcasting is sending a packet to *all* hosts on a network (usually a locally attached network) and multicasting is sending a packet to a *set* of hosts on a network. Basic to these two concepts is an understanding of the different types of filtering that occur when a received frame passes up a protocol stack. Each layer can discard a received packet for different reasons.

There are four types of broadcast addresses: limited, net-directed, subnet-directed, and all-subnets-directed. The most common is subnet-directed. The limited broadcast address is normally seen only when a system is bootstrapping.

Problems occur when trying to broadcast through routers, often because the router may not know the subnet mask of the destination network. The results depend on many factors: which type of broadcast address, configuration parameters, and so on.

A class D IP address is called a multicast group address. It is converted to an Ethernet address by placing its lower 23 bits into a fixed Ethernet address. The mapping is not unique, requiring additional filtering by one of the protocol modules.

Exercises

12.1 Does broadcasting increase the amount of network traffic?

12.2 Consider 50 hosts on an Ethernet: 20 running TCP/IP and 30 running some other protocol suite. How are broadcasts from one protocol suite handled by hosts running the other protocol suite?

12.3 You login to a Unix system that you've never used before and want to find the subnet-directed broadcast address for all attached interfaces that support broadcasting. How can you do this?

12.4 If we `ping` the broadcast address with a large packet size, as in

```
sun % ping 140.252.13.63 1472
PING 140.252.13.63: 1472 data bytes
1480 bytes from sun (140.252.13.33): icmp_seq=0. time=6. ms
1480 bytes from svr4 (140.252.13.34): icmp_seq=0. time=84. ms
1480 bytes from bsdi (140.252.13.35): icmp_seq=0. time=128. ms
```

it works, but increasing the packet size by 1 byte gives us the following error:

```
sun % ping 140.252.13.63 1473
PING 140.252.13.63: 1473 data bytes
sendto: Message too long
```

What's going on?

12.5 Redo Exercise 10.6 assuming the eight RIP messages are multicast instead of broadcast (assume RIP Version 2 is being used). What changes?

13

IGMP: Internet Group Management Protocol

13.1 Introduction

Section 12.4 provided an overview of IP multicasting and described how class D IP addresses are mapped into Ethernet addresses. We briefly mentioned how multicasting occurs on a single physical network, but said complications occur when multiple networks are involved and the multicast datagrams must pass through routers.

In this chapter we'll look at the *Internet Group Management Protocol* (IGMP), which is used by hosts and routers that support multicasting. It lets all the systems on a physical network know which hosts currently belong to which multicast groups. This information is required by the multicast routers, so they know which multicast datagrams to forward onto which interfaces. IGMP is defined in RFC 1112 [Deering 1989].

Like ICMP, IGMP is considered part of the IP layer. Also like ICMP, IGMP messages are transmitted in IP datagrams. Unlike other protocols that we've seen, IGMP has a fixed-size message, with no optional data. Figure 13.1 shows the encapsulation of an IGMP message within an IP datagram.

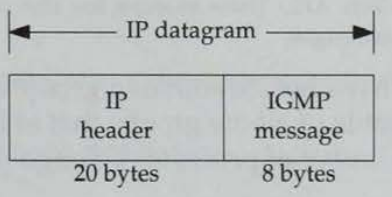


Figure 13.1 Encapsulation of an IGMP message within an IP datagram.

IGMP messages are specified in the IP datagram with a protocol value of 2.

13.2 IGMP Message

Figure 13.2 shows the format of the 8-byte IGMP message.

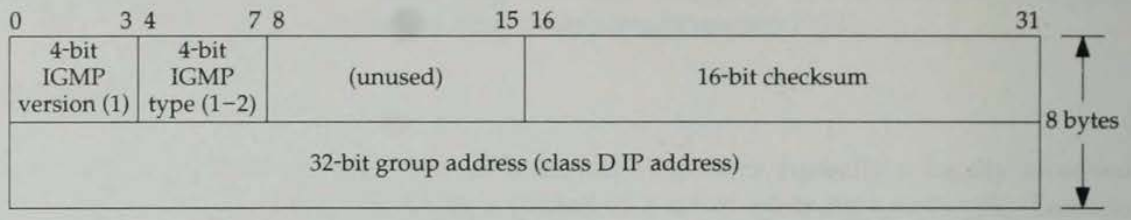


Figure 13.2 Format of fields in IGMP message.

The *IGMP version* is 1. An *IGMP type* of 1 is a query sent by a multicast router, and 2 is a response sent by a host. The *checksum* is calculated in the same manner as the ICMP checksum.

The *group address* is a class D IP address. In a query the group address is set to 0, and in a report it contains the group address being reported. We'll say more about it in the next section when we see how IGMP operates.

13.3 IGMP Protocol

Joining a Multicast Group

Fundamental to multicasting is the concept of a process joining a multicast group on a given interface on a host. (We use the term *process* to mean a program being executed by the operating system.) Membership in a multicast group on a given interface is dynamic—it changes over time as processes join and leave the group.

We imply here that a process must have a way of joining a multicast group on a given interface. A process can also leave a multicast group that it previously joined. These are required parts of any API on a host that supports multicasting. We use the qualifier "interface" because membership in a group is associated with an interface. A process can join the same group on multiple interfaces.

> The release of IP multicasting for Berkeley Unix from Stanford University details these changes for the sockets API. These changes are also provided in Solaris 2.x and documented in the `ip`(7) manual pages.

Implied here is that a host identifies a group by the group address *and* the interface. A host must keep a table of all the groups that at least one process belongs to, and a reference count of the number of processes belonging to the group.

IGMP Reports and Queries

IGMP messages are used by multicast routers to keep track of group membership on each of the router's physically attached networks. The following rules apply.

1. A host sends an IGMP report when the first process joins a group. If multiple processes on a given host join the same group, only one report is sent, the first time a process joins that group. This report is sent out the same interface on which the process joined the group.
2. A host does not send a report when processes leave a group, even when the last process leaves a group. The host knows that there are no members in a given group, so when it receives the next query (next step), it won't report the group.
3. A multicast router sends an IGMP query at regular intervals to see if any hosts still have processes belonging to any groups. The router must send one query out each interface. The group address in the query is 0 since the router expects one response from a host for every group that contains one or more members on that host.
4. A host responds to an IGMP query by sending one IGMP report for each group that still contains at least one process.

Using these queries and reports, a multicast router keeps a table of which of its interfaces have one or more hosts in a multicast group. When the router receives a multicast datagram to forward, it forwards the datagram (using the corresponding multicast link-layer address) only out the interfaces that still have hosts with processes belonging to that group.

Figure 13.3 shows these two IGMP messages, reports sent by hosts, and queries sent by routers. The router is asking each host to identify each group on that interface.

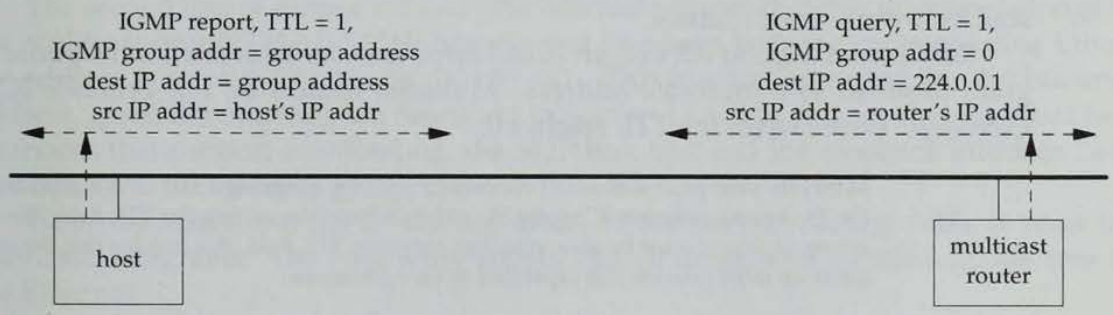


Figure 13.3 IGMP reports and queries.

We talk about the TTL field later in this section.

Implementation Details

There are many implementation details in this protocol that improve its efficiency. First, when a host sends an initial IGMP report (when the first process joins a group), there's no guarantee that the report is delivered (since IP is used as the delivery service). Another report is sent at a later time. This later time is chosen by the host to be a random value between 0 and 10 seconds.

Next, when a host receives a query from a router it doesn't respond immediately, but schedules the responses for later times. (We use the plural "responses" because the host must send one report for each group that contains one or more members.) Since multiple hosts can be sending a report for the same group, each schedules its response using random delays. Also realize that all the hosts on a physical network receive all the reports from other hosts in the same group, because the destination address of the report in Figure 13.3 is the group's address. This means that, if a host is scheduled to send a report, but receives a copy of the same report from another host, the response can be canceled. This is because a multicast router doesn't care how many hosts belong to the group—only whether at least one host belongs to the group. Indeed, a multicast router doesn't even care which host belongs to a group. It only needs to know that at least one host belongs to a group on a given interface.

On a single physical network without any multicast routers, the only IGMP traffic is the reports issued by the hosts that support IP multicasting, when the host joins a new group.

Time-to-Live Field

In Figure 13.3 we noted that the TTL field of the reports and queries is set to 1. This refers to the normal TTL field in the IP header. A multicast datagram with an initial TTL of 0 is restricted to the same host. By default, multicast datagrams are sent with a TTL of 1. This restricts the datagram to the same subnet. Higher TTLs can be forwarded by multicast routers.

Recall from Section 6.2 that an ICMP error is never generated in response to a datagram destined to a multicast address. Multicast routers do not generate ICMP "time exceeded" errors when the TTL reaches 0.

> Normally user processes aren't concerned with the outgoing TTL. One exception, however, is the Traceroute program (Chapter 8), which is based on setting the TTL field. Since multicasting applications must be able to set the outgoing TTL field, this implies that the programming interface must provide this capability to user processes.

By increasing the TTL an application can perform an *expanding ring search* for a particular server. The first multicast datagram is sent with a TTL of 1. If no response is received, a TTL of 2 is tried, then 3, and so on. In this way the application locates the closest server, in terms of hops.

The special range of addresses 224.0.0.0 through 224.0.0.255 is intended for applications that never need to multicast further than one hop. A multicast router should never forward a datagram with one of these addresses as the destination, regardless of the TTL.

All-Hosts Group

In Figure 13.3 we also indicated that the router's IGMP query is sent to the destination IP address of 224.0.0.1. This is called the *all-hosts* group address. It refers to all the multicast-capable hosts and routers on a physical network. Each host automatically joins this multicast group on all multicast-capable interfaces, when the interface is initialized. Membership in this group is never reported.

13.4 An Example

Now that we've gone through some of the details of IP multicasting, let's take a look at the messages involved. We've added IP multicasting support to the host `sun` and will use some test programs provided with the multicasting software to see what happens.

First we'll use a modified version of the `netstat` command that reports multicast group membership for each interface. (We showed the standard `netstat -ni` output for this host in Section 3.9.) In the following output we show the lines corresponding to multicast groups in a bold font:

```
sun % netstat -nia
Name  Mtu   Network     Address            Ipkts Ierrs    Opkts Oerrs  Coll
le0   1500  140.252.13. 140.252.13.33       4370     0     4924     0     0
                        224.0.0.1
                        08:00:20:03:f6:42
                        01:00:5e:00:00:01
sl0   552   140.252.1   140.252.1.29       13587     0    15615     0     0
                        224.0.0.1
lo0   1536  127         127.0.0.1           1351     0     1351     0     0
                        224.0.0.1
```

The `-n` option prints IP addresses in numeric format (instead of trying to print them as names), `-i` prints the interface statistics, and `-a` reports on all configured interfaces.

The second line of output for `le0` (the Ethernet) shows that this interface belongs to the multicast group 224.0.0.1 ("all hosts"), and two lines later the corresponding Ethernet address is shown: `01:00:5e:00:00:01`. This is what we expect for the Ethernet address, given the mapping we described in Section 12.4. We also see that the other two interfaces that support multicasting, the SLIP link `sl0` and the loopback interface `lo0`, also belong to the all-hosts group.

We must also show the IP routing table, as the normal routing table is used for multicast datagrams. The bold entry shows that all datagrams for 224.0.0.0 are sent to the Ethernet:

```
sun % netstat -rn
Routing tables
Destination          Gateway              Flags    Refcnt Use      Interface
140.252.13.65        140.252.13.35        UGH      0      32       le0
127.0.0.1            127.0.0.1            UH       1      381      lo0
140.252.1.183        140.252.1.29         UH       0      6        sl0
default              140.252.1.183        UG       0      328      sl0
224.0.0.0            140.252.13.33        U        0      66       le0
140.252.13.32        140.252.13.33        U        8      5581     le0
```

If you compare this routing table to the one shown in Section 9.2 for the router sun, you'll see that the multicast entry is the only change.

We now run a test program that lets us join a multicast group on an interface. (We don't show any output for our use of this test program.) We join the group 224.1.2.3 on the Ethernet interface (140.252.13.33). Executing `netstat` shows that the kernel has joined the group, and again the Ethernet address is what we expect. We show the changes from the previous `netstat` output in a bold font:

```
sun % netstat -nia
Name  Mtu   Network     Address            Ipkts Ierrs    Opkts Oerrs  Coll
le0   1500  140.252.13. 140.252.13.33       4374     0     4929     0     0
                        224.1.2.3
                        224.0.0.1
                        08:00:20:03:f6:42
                        01:00:5e:01:02:03
                        01:00:5e:00:00:01
sl0   552   140.252.1   140.252.1.29       13862     0    15943     0     0
                        224.0.0.1
lo0   1536  127         127.0.0.1           1360     0     1360     0     0
                        224.0.0.1
```

We have shown the output again for the other two interfaces, `sl0` and `lo0`, to reiterate that the multicast group is joined only on one interface.

Figure 13.4 shows the `tcpdump` output corresponding to the process joining the multicast group.

```
1  0.0                    8:0:20:3:f6:42 1:0:5e:1:2:3 ip 60:
                          sun > 224.1.2.3: igmp report 224.1.2.3 [ttl 1]

2  6.94 (6.94)            8:0:20:3:f6:42 1:0:5e:1:2:3 ip 60:
                          sun > 224.1.2.3: igmp report 224.1.2.3 [ttl 1]
```

Figure 13.4 `tcpdump` output when a host joins a multicast group.

Line 1 occurs when the host joins the group. The next line is the delayed report that we said is sent at some random time up to 10 seconds afterward.

We have shown the hardware addresses in these two lines, to verify that the Ethernet destination address is the correct multicast address. We can also see that the source IP address is the one corresponding to sun, and the destination IP address is the multicast group address. We can also see that the reported address is that same multicast group address.

Finally, we note that the TTL is 1, as specified. `tcpdump` prints the TTL in square brackets when its value is 0 or 1. This is because the TTL is normally greater than this. With multicasting, however, we expect to see lots of IP datagrams with a TTL of 1.

Implied in this output is that a multicast router must receive *all* multicast datagrams on all its interfaces. The router has no idea which multicast groups the hosts might join.

Multicast Router Example

Let's continue the previous example, but we'll also start a multicast routing daemon on the host sun. Our interest here is not the details of multicast routing protocols, but to see the IGMP queries and reports that are exchanged. Even though the multicast routing daemon is running on the only host that supports multicasting (sun), all the queries and reports are multicast on the Ethernet, so we can see them on any other system on the Ethernet.

Before starting the routing daemon we joined another multicast group: 224.9.9.9. Figure 13.5 shows the output.

```
1    0.0                 sun > 224.0.0.4: igmp report 224.0.0.4

2    0.00 (  0.00)       sun > 224.0.0.1: igmp query
3    5.10 (  5.10)       sun > 224.9.9.9: igmp report 224.9.9.9

4    5.22 (  0.12)       sun > 224.0.0.1: igmp query
5    7.90 (  2.68)       sun > 224.1.2.3: igmp report 224.1.2.3
6    8.50 (  0.60)       sun > 224.0.0.4: igmp report 224.0.0.4
7   11.70 (  3.20)       sun > 224.9.9.9: igmp report 224.9.9.9

8  125.51 (113.81)       sun > 224.0.0.1: igmp query
9  125.70 (  0.19)       sun > 224.9.9.9: igmp report 224.9.9.9
10 128.50 (  2.80)       sun > 224.1.2.3: igmp report 224.1.2.3
11 129.10 (  0.60)       sun > 224.0.0.4: igmp report 224.0.0.4

12 247.82 (118.72)       sun > 224.0.0.1: igmp query
13 248.09 (  0.27)       sun > 224.1.2.3: igmp report 224.1.2.3
14 248.69 (  0.60)       sun > 224.0.0.4: igmp report 224.0.0.4
15 255.29 (  6.60)       sun > 224.9.9.9: igmp report 224.9.9.9
```

Figure 13.5 tcpdump output while multicast routing daemon is running.

We have not included the Ethernet addresses in this output, because we've already verified that they are what we expect. We've also deleted the notation that the TTL equals 1, because again that's what we expect.

Line 1 is output when the routing daemon starts. It sends a report that it has joined the group 224.0.0.4. Multicast address 224.0.0.4 is a well-known address used by DVMRP (Distance Vector Multicast Routing Protocol), the protocol currently used for multicast routing. (DVMRP is defined in RFC 1075 [Waitzman, Partridge, and Deering 1988].)

When the daemon starts it also sends out a query (line 2). The destination IP address of the query is 224.0.0.1 (all-hosts), as shown in Figure 13.3.

The first report (line 3) is received about 5 seconds later, for group 224.9.9.9. This is the only report received before another query is sent (line 4). These two queries (lines 2 and 4) occur rapidly when the daemon starts up, as it tries to build its multicast routing table.

Lines 5, 6, and 7 are what we expect: one report from the host sun for each group to which it belongs. Notice that the group 224.0.0.4 is reported, in addition to the two groups that we explicitly joined, because as long as the routing daemon is running, it belongs to this group.

The next query on line 8 occurs about 2 minutes after the previous query. Again it elicits the three reports we expect (lines 9, 10, and 11). The reports are in a different order this time, as expected, since the time between receiving the query and sending the report should be randomized.

The final query that we show occurs about 2 minutes after the previous query, and again we have the expected responses.

13.5 Summary

Multicasting is a way to send a message to multiple recipients. In many applications it is better than broadcasting, since multicasting imposes less overhead on hosts that are not participating in the communication. The simple host membership reporting protocol (IGMP) is the basic building block for multicasting.

Multicasting on a LAN or across closely connected LANs uses the techniques we've described in this chapter. Since broadcasting is often restricted to a single LAN, multicasting could be used instead of broadcasting for many applications that use broadcasting today.

A problem that has not been completely solved, however, is multicasting across wide area networks. [Deering and Cheriton 1990] propose extensions to common routing protocols to support multicasting. Section 9.13 of [Perlman 1992] discusses some of the problems with multicasting across WANs.

[Casner and Deering 1992] describe the delivery of audio for an IETF meeting across the Internet using multicasting and a virtual network called the *MBONE* (multicasting backbone).

Exercises

13.1 We said that hosts schedule IGMP reports with random delays. How can the hosts on a LAN try to ensure that no two hosts generate the same random delay?

13.2 In [Casner and Deering 1992] they mention that UDP lacks two features needed for sending audio samples across the MBONE: detection of packet reordering and detection of duplicate packets. How could you add these capabilities above UDP?

14

DNS: The Domain Name System

14.1 Introduction

The *Domain Name System*, or *DNS*, is a distributed database that is used by TCP/IP applications to map between hostnames and IP addresses, and to provide electronic mail routing information. We use the term *distributed* because no single site on the Internet knows all the information. Each site (university department, campus, company, or department within a company, for example) maintains its own database of information and runs a server program that other systems across the Internet (clients) can query. The DNS provides the protocol that allows clients and servers to communicate with each other.

From an application's point of view, access to the DNS is through a *resolver*. On Unix hosts the resolver is accessed primarily through two library functions, `gethostbyname`(3) and `gethostbyaddr`(3), which are linked with the application when the application is built. The first takes a hostname and returns an IP address, and the second takes an IP address and looks up a hostname. The resolver contacts one or more *name servers* to do the mapping.

In Figure 4.2 (p. 55) we showed that the resolver is normally part of the application. It is not part of the operating system kernel as are the TCP/IP protocols. Another fundamental point from this figure is that an application must convert a hostname to an IP address before it can ask TCP to open a connection or send a datagram using UDP. The TCP/IP protocols within the kernel know nothing about the DNS.

In this chapter we'll take a look at how resolvers communicate with name servers using the TCP/IP protocols (mainly UDP). We do not cover all the administrative details of running a name server or all the options available with resolvers and servers. These details can fill an entire book. (See [Albitz and Liu 1992] for all the details on the care and feeding of the standard Unix resolver and name server.)

RFC 1034 [Mockapetris 1987a] specifies the concepts and facilities provided by the DNS, and RFC 1035 [Mockapetris 1987b] details the implementation and specification. The most commonly used implementation of the DNS, both resolver and name server, is called BIND—the Berkeley Internet Name Domain server. The server is called named. An analysis of the wide-area network traffic generated by the DNS is given in [Danzig, Obraczka, and Kumar 1992].

14.2 DNS Basics

The DNS name space is hierarchical, similar to the Unix filesystem. Figure 14.1 shows this hierarchical organization.

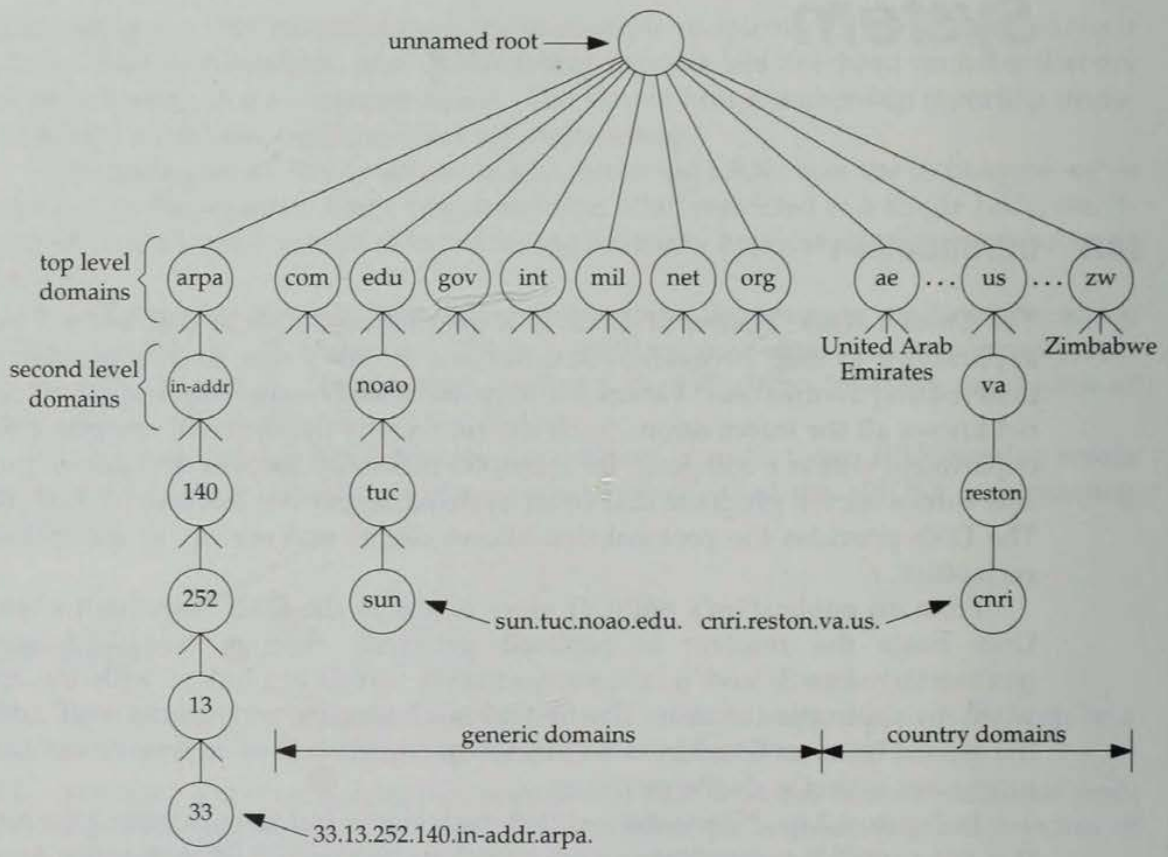


Figure 14.1 Hierarchical organization of the DNS.

Every node (circles in Figure 14.1) has a *label* of up to 63 characters. The root of the tree is a special node with a null label. Any comparison of labels considers uppercase and lowercase characters the same. The *domain name* of any node in the tree is the list of labels, starting at that node, working up to the root, using a period ("dot") to separate

the labels. (Note that this is different from the Unix filesystem, which forms a pathname by starting at the top and going down the tree.) Every node in the tree must have a unique domain name, but the same label can be used at different points in the tree.

A domain name that ends with a period is called an *absolute domain name* or a *fully qualified domain name* (FQDN). An example is `sun.tuc.noao.edu.`. If the domain name does not end with a period, it is assumed that the name needs to be completed. How the name is completed depends on the DNS software being used. If the uncompleted name consists of two or more labels, it might be considered to be complete; otherwise a local addition might be added to the right of the name. For example, the name `sun` might be completed by adding the local suffix `.tuc.noao.edu.`.

The top-level domains are divided into three areas:

1. `arpa` is a special domain used for address-to-name mappings. (We describe this in Section 14.5.)
2. The seven 3-character domains are called the *generic* domains. Some texts call these the *organizational* domains.
3. All the 2-character domains are based on the country codes found in ISO 3166. These are called the *country* domains, or the *geographical* domains.

Figure 14.2 lists the normal classification of the seven generic domains.

Domain	Description
`com`	commercial organizations
`edu`	educational institutions
`gov`	other U.S. governmental organizations
`int`	international organizations
`mil`	U.S. military
`net`	networks
`org`	other organizations

Figure 14.2 The 3-character generic domains.

DNS folklore says that the 3-character generic domains are only for U.S. organizations, and the 2-character country domains for everyone else, but this is false. There are many non-U.S. organizations in the generic domains, and many U.S. organizations in the `.us` country domain. (RFC 1480 [Cooper and Postel 1993] describes the `.us` domain in more detail.) The only generic domains that are restricted to the United States are `.gov` and `.mil`.

Many countries form second-level domains beneath their 2-character country code similar to the generic domains: `.ac.uk`, for example, is for academic institutions in the United Kingdom and `.co.uk` is for commercial organizations in the United Kingdom.

One important feature of the DNS that isn't shown in figures such as Figure 14.1 is the delegation of responsibility within the DNS. No single entity manages every label in the tree. Instead, one entity (the NIC) maintains a portion of the tree (the top-level domains) and delegates responsibility to others for specific zones.

A *zone* is a subtree of the DNS tree that is administered separately. A common zone is a second-level domain, `noao.edu`, for example. Many second-level domains then divide their zone into smaller zones. For example, a university might divide itself into zones based on departments, and a company might divide itself into zones based on branch offices or internal divisions.

> If you are familiar with the Unix filesystem, notice that the division of the DNS tree into zones is similar to the division of a logical Unix filesystem into physical disk partitions. Just as we can't tell from Figure 14.1 where the zones of authority lie, we can't tell from a similar picture of a Unix filesystem which directories are on which disk partitions.

Once the authority for a zone is delegated, it is up to the person responsible for the zone to provide multiple *name servers* for that zone. Whenever a new system is installed in a zone, the DNS administrator for the zone allocates a name and an IP address for the new system and enters these into the name server's database. This is where the need for delegation becomes obvious. At a small university, for example, one person could do this each time a new system was added, but in a large university the responsibility would have to be delegated (probably by departments), since one person couldn't keep up with the work.

A name server is said to have authority for one zone or multiple zones. The person responsible for a zone must provide a *primary name server* for that zone and one or more *secondary name servers*. The primary and secondaries must be independent and redundant servers so that availability of name service for the zone isn't affected by a single point of failure.

The main difference between a primary and secondary is that the primary loads all the information for the zone from disk files, while the secondaries obtain all the information from the primary. When a secondary obtains the information from its primary we call this a *zone transfer*.

When a new host is added to a zone, the administrator adds the appropriate information (name and IP address minimally) to a disk file on the system running the primary. The primary name server is then notified to reread its configuration files. The secondaries query the primary on a regular basis (normally every 3 hours) and if the primary contains newer data, the secondary obtains the new data using a zone transfer.

What does a name server do when it doesn't contain the information requested? It must contact another name server. (This is the *distributed* nature of the DNS.) Not every name server, however, knows how to contact every other name server. Instead every name server must know how to contact the *root name servers*. As of April 1993 there were eight root servers and all the primary servers must know the IP address of each root server. (These IP addresses are contained in the primary's configuration files. The primary servers must know the IP addresses of the root servers, not their DNS names.) The root servers then know the name and location (i.e., the IP address) of each authoritative name server for all the second-level domains. This implies an iterative process: the requesting name server must contact a root server. The root server tells the requesting server to contact another server, and so on. We'll look into this procedure with some examples later in this chapter.

You can fetch the current list of root servers using anonymous FTP. Obtain the file `netinfo/root-servers.txt` from either `ftp.rs.internic.net` or `nic.ddn.mil`.

A fundamental property of the DNS is *caching*. That is, when a name server receives information about a mapping (say, the IP address of a hostname) it caches that information so that a later query for the same mapping can use the cached result and not result in additional queries to other servers. Section 14.7 shows an example of caching.

14.3 DNS Message Format

There is one DNS message defined for both queries and responses. Figure 14.3 shows the overall format of the message.

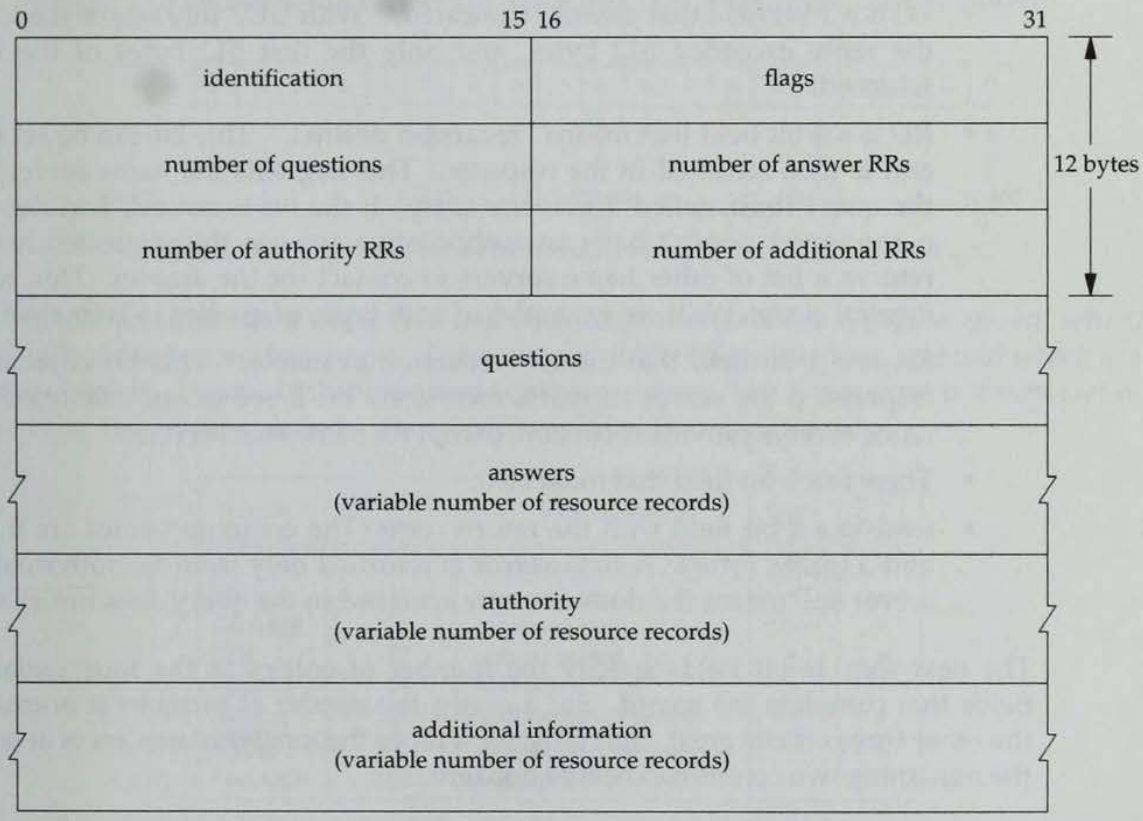


Figure 14.3 General format of DNS queries and responses.

The message has a fixed 12-byte header followed by four variable-length fields.

The *identification* is set by the client and returned by the server. It lets the client match responses to requests.

The 16-bit *flags* field is divided into numerous pieces, as shown in Figure 14.4.

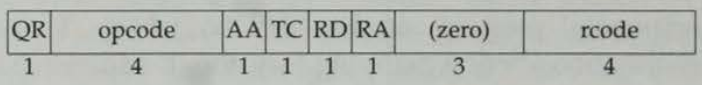


Figure 14.4 *flags* field in the DNS header.

We'll start at the leftmost bit and describe each field.

- *QR* is a 1-bit field: 0 means the message is a query, 1 means it's a response.
- *opcode* is a 4-bit field. The normal value is 0 (a standard query). Other values are 1 (an inverse query) and 2 (server status request).
- *AA* is a 1-bit flag that means "authoritative answer." The name server is authoritative for the domain in the question section.
- *TC* is a 1-bit field that means "truncated." With UDP this means the total size of the reply exceeded 512 bytes, and only the first 512 bytes of the reply was returned.
- *RD* is a 1-bit field that means "recursion desired." This bit can be set in a query and is then returned in the response. This flag tells the name server to handle the query itself, called a *recursive query*. If the bit is not set, and the requested name server doesn't have an authoritative answer, the requested name server returns a list of other name servers to contact for the answer. This is called an *iterative query*. We'll see examples of both types of queries in later examples.
- *RA* is a 1-bit field that means "recursion available." This bit is set to 1 in the response if the server supports recursion. We'll see in our examples that most name servers provide recursion, except for some root servers.
- There is a 3-bit field that must be 0.
- *rcode* is a 4-bit field with the return code. The common values are 0 (no error) and 3 (name error). A name error is returned only from an authoritative name server and means the domain name specified in the query does not exist.

The next four 16-bit fields specify the number of entries in the four variable-length fields that complete the record. For a query, the *number of questions* is normally 1 and the other three counts are 0. Similarly, for a reply the *number of answers* is at least 1, and the remaining two counts can be 0 or nonzero.

Question Portion of DNS Query Message

The format of each question in the *question* section is shown in Figure 14.5. There is normally just one question.

The *query name* is the name being looked up. It is a sequence of one or more labels. Each *label* begins with a 1-byte count that specifies the number of bytes that follow. The name is terminated with a byte of 0, which is a label with a length of 0, which is the label of the root. Each count byte must be in the range of 0 to 63, since labels are limited

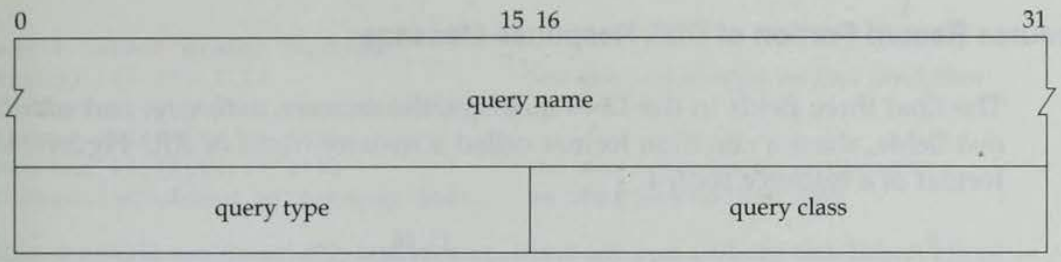


Figure 14.5 Format of *question* portion of DNS query message.

to 63 bytes. (We'll see later in this section that a count byte with the two high-order bits turned on, values 192 to 255, is used with a compression scheme.) Unlike many other message formats that we've encountered, this field is allowed to end on a boundary other than a 32-bit boundary. No padding is used.

Figure 14.6 shows how the domain name `gemini.tuc.noao.edu` is stored.

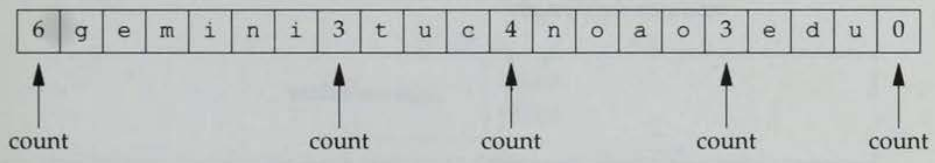


Figure 14.6 Representation of the domain name `gemini.tuc.noao.edu`.

Each question has a *query type* and each response (called a resource record, which we talk about below) has a *type*. There are about 20 different values, some of which are now obsolete. Figure 14.7 shows some of these values. The query type is a superset of the type: two of the values we show can be used only in questions.

Name	Numeric value	Description	*type*?	*query type*?
A	1	IP address	•	•
NS	2	name server	•	•
CNAME	5	canonical name	•	•
PTR	12	pointer record	•	•
HINFO	13	host info	•	•
MX	15	mail exchange record	•	•
AXFR	252	request for zone transfer		•
* or ANY	255	request for all records		•

Figure 14.7 *type* and *query type* values for DNS questions and responses.

The most common query type is an A type, which means an IP address is desired for the *query name*. A PTR query requests the names corresponding to an IP address. This is a pointer query that we describe in Section 14.5. We describe the other query types in Section 14.6.

The *query class* is normally 1, meaning Internet address. (Some other non-IP values are also supported at some locations.)

Resource Record Portion of DNS Response Message

The final three fields in the DNS message, the *answers*, *authority*, and *additional information* fields, share a common format called a *resource record* or *RR*. Figure 14.8 shows the format of a resource record.

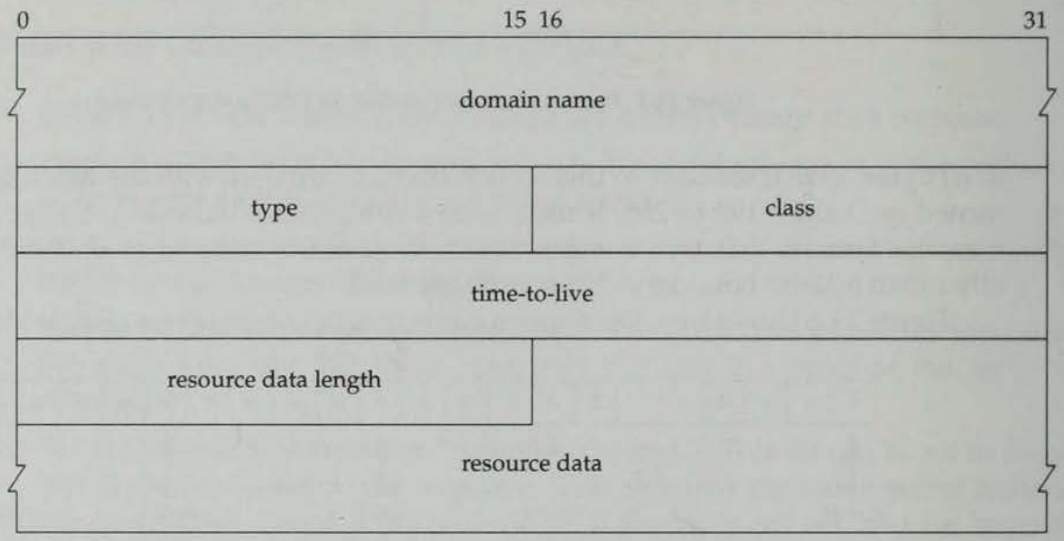


Figure 14.8 Format of DNS resource record.

The *domain name* is the name to which the following resource data corresponds. It is in the same format as we described earlier for the *query name* field (Figure 14.6).

The *type* specifies one of the RR type codes. These are the same as the *query type* values that we described earlier. The *class* is normally 1 for Internet data.

The *time-to-live* field is the number of seconds that the RR can be cached by the client. RRs often have a TTL of 2 days.

The *resource data length* specifies the amount of *resource data*. The format of this data depends on the *type*. For a type of 1 (an A record) the resource data is a 4-byte IP address.

Now that we've described the basic format of the DNS queries and responses, we'll see what is passed in the packets by watching some exchanges using `tcpdump`.

14.4 A Simple Example

Let's start with a simple example to see the communication between a resolver and a name server. We'll run the Telnet client on the host `sun` to the host `gemini`, connecting to the daytime server:

```
sun % telnet gemini daytime
Trying 140.252.1.11 ...                  first three lines of output are from Telnet client
Connected to gemini.tuc.noao.edu.
Escape character is '^]'.
Wed Mar 24 10:44:17 1993                 this is the output from the daytime server
Connection closed by foreign host.       and this is from the Telnet client
```

For this example we direct the resolver on the host `sun` (where the Telnet client is run) to use the name server on the host `noao.edu` (140.252.1.54). Figure 14.9 shows the arrangement of the three systems.

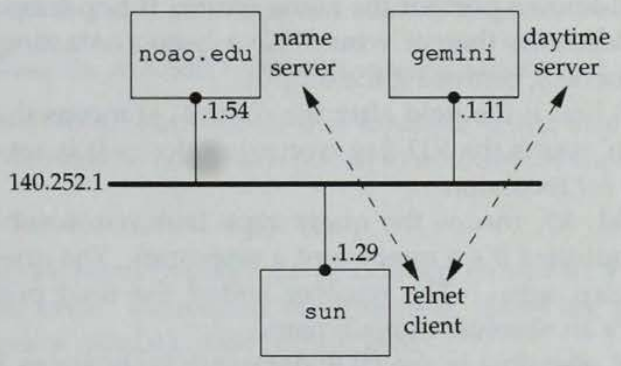


Figure 14.9 Systems being used for simple DNS example.

As we've mentioned before, the resolver is part of the client, and the resolver contacts a name server to obtain the IP address before the TCP connection can be established between Telnet and the daytime server.

In this figure we've omitted the detail that the connection between `sun` and the 140.252.1 Ethernet is really a SLIP link (see the figure on the inside front cover) because that doesn't affect the discussion. We will, however, run `tcpdump` on the SLIP link to see the packets exchanged between the resolver and name server.

The file `/etc/resolv.conf` on the host `sun` tells the resolver what to do:

```
sun % cat /etc/resolv.conf
nameserver 140.252.1.54
domain tuc.noao.edu
```

The first line gives the IP address of the name server—the host `noao.edu`. Up to three `nameserver` lines can be specified, to provide backup in case one is down or unreachable. The `domain` line specifies the default domain. If the name being looked up is not a fully qualified domain name (it doesn't end with a period) then the default domain `.tuc.noao.edu` is appended to the name. This is why we can type `telnet gemini` instead of `telnet gemini.tuc.noao.edu`.

Figure 14.10 shows the packet exchange between the resolver and name server.

```
1  0.0                    140.252.1.29.1447 > 140.252.1.54.53: 1+ A?
                          gemini.tuc.noao.edu. (37)

2  0.290820 (0.2908)      140.252.1.54.53 > 140.252.1.29.1447: 1* 2/0/0 A
                          140.252.1.11 (69)
```

Figure 14.10 `tcpdump` output for name server query of the hostname `gemini.tuc.noao.edu`.

We've instructed `tcpdump` not to print domain names for the source and destination IP addresses of each IP datagram. Instead it prints 140.252.1.29 for the client (the resolver) and 140.252.1.54 for the name server. Port 1447 is the ephemeral port used by the client and 53 is the well-known port for the name server. If `tcpdump` had tried to print names instead of IP addresses, then it would have been contacting the same name server (doing pointer queries), confusing the output.

Starting with line 1, the field after the colon (`1+`) means the identification field is 1, and the plus sign means the RD flag (recursion desired) is set. We see that by default, the resolver asks for recursion.

The next field, `A?`, means the query type is A (we want an IP address), and the question mark indicates it's a query (not a response). The query name is printed next: `gemini.tuc.noao.edu.`. The resolver added the final period to the query name, indicating that it's an absolute domain name.

The length of user data in the UDP datagram is shown as 37 bytes: 12 bytes are the fixed-size header (Figure 14.3); 21 bytes for the query name (Figure 14.6), and 4 bytes for the query type and query class. The odd-length UDP datagram reiterates that there is no padding in the DNS messages.

Line 2 in the `tcpdump` output is the response from the name server and `1*` is the identification field with the asterisk meaning the AA flag (authoritative answer) is set. (We expect this server, the primary server for the `noao.edu` domain, to be authoritative for names within its domain.)

The output `2/0/0` shows the number of resource records in the final three variable-length fields in the response: 2 answer RRs, 0 authority RRs, and 0 additional RRs. `tcpdump` only prints the first answer, which in this case has a type of A (IP address) with a value of 140.252.1.11.

Why do we get two answers to our query? Because the host `gemini` is multi-homed. Two IP addresses are returned. Indeed, another useful tool with the DNS is a publicly available program named `host`. It lets us issue queries to a name server and see what comes back. If we run this program we'll see the two IP addresses for this host:

```
sun % host gemini
gemini.tuc.noao.edu      A       140.252.1.11
gemini.tuc.noao.edu      A       140.252.3.54
```

The first answer in Figure 14.10 and the first line of output from the `host` command are the IP address that shares the same subnet (140.252.1) as the requesting host. This is not an accident. If the name server and the host issuing the query are on the same network (or subnet), then BIND sorts the results so that addresses on common networks appear first.

We can still access the host `gemini` using the other address, but it might be less efficient. Using `traceroute` in this instance shows that the normal route from subnet 140.252.1 to 140.252.3 is not through the host `gemini`, but through another router that's connected to both networks. So in this case if we accessed `gemini` through the other IP address (140.252.3.54) all the packets would require an additional hop. We return to this example and explore the reason for the alternative route in Section 25.9, when we can use SNMP to look at a router's routing table.

There are other programs that provide easy interactive access to the DNS. `nslookup` is supplied with most implementations of the DNS. Chapter 10 of [Albitz and Liu 1992] provides a detailed description of how to use this program. The `dig` program ("Domain Internet Groper") is another publicly available tool that queries DNS servers. `doc` ("Domain Obscenity Control") is a shell script that uses `dig` and diagnoses misbehaving domains by sending queries to the appropriate DNS name servers, and performing simple analysis of the responses. See Appendix F for details on how to obtain these programs.

The final detail to account for in this example is the size of the UDP data in the reply: 69 bytes. We need to know two points to account for these bytes.

1. The question is returned in the reply.
2. There can be many repetitions of domain names in a reply, so a compression scheme is used. Indeed, in our example, there are three occurrences of the domain name `gemini.tuc.noao.edu`.

   The compression scheme is simple. Anywhere the label portion of a domain name can occur, the single count byte (which is between 0 and 63) has its two high-order bits turned on instead. This means it is a 16-bit *pointer* and not an 8-bit count byte. The 14 bits that follow in the pointer specify an offset in the DNS message of a label to continue with. (The offset of the first byte in the identification field is 0.) We purposely said that this pointer can occur wherever a label can occur, not just where a complete domain name can occur, since it's possible for a pointer to form either a complete domain name or just the ending portion of a name. (This is because the ending labels in the names from a given domain tend to be identical.)

Figure 14.11 shows the format of the DNS reply, line 2 from Figure 14.10. We also show the IP and UDP headers to reiterate that DNS messages are normally encapsulated in UDP datagrams. We explicitly show the count bytes in the labels of the domain name in the question. The two answers returned are the same, except for the different IP addresses returned in each answer. In this example the pointer in each answer would have a value of 12, the offset from the start of the DNS header of the complete domain name.

The final point to note from this example is from the second line of output from the Telnet command, which we repeat here:

```
sun % telnet gemini daytime          we only type gemini
Trying 140.252.1.11 ...
Connected to gemini.tuc.noao.edu.    but the Telnet client outputs FQDN
```

We typed just the hostname (`gemini`), not the FQDN, but the Telnet client output the FQDN. What's happening is that the Telnet client looks up the name we type by calling

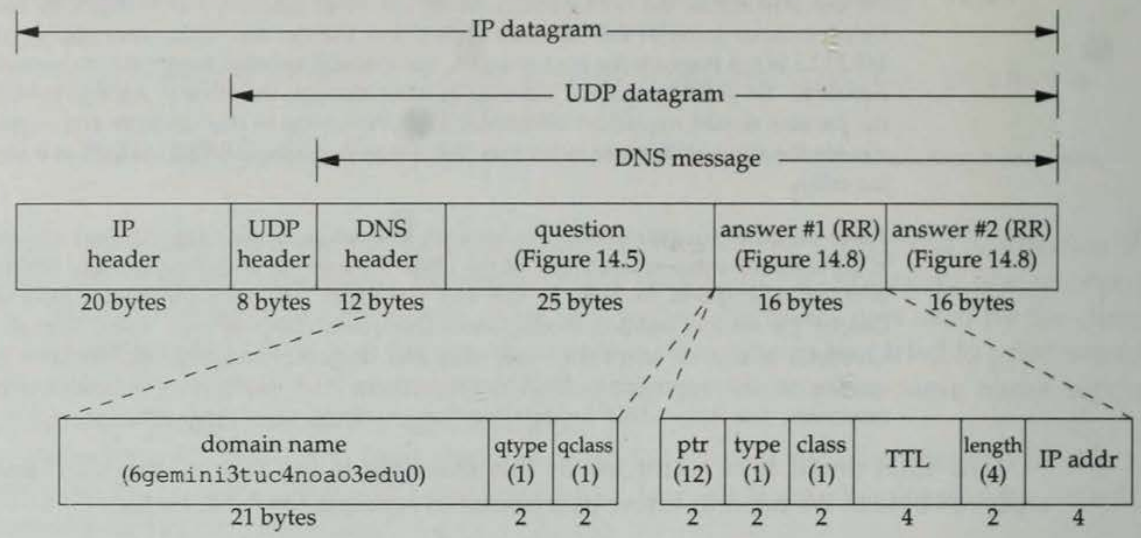


Figure 14.11 Format of DNS reply corresponding to line 2 of Figure 14.10.

the resolver (`gethostbyname`), which returns the IP addresses and the FQDN. Telnet then prints the IP address that it's trying to establish a TCP connection with, and when the connection is established, it outputs the FQDN.

If there is a significant pause between typing the Telnet command and printing the IP address, this delay is caused by the resolver contacting a name server to resolve the name into an IP address. A pause between printing `Trying` and `Connected to`, however, is a delay caused by the establishment of the TCP connection between the client and server, not the DNS.

14.5 Pointer Queries

A perpetual stumbling block in understanding the DNS is how pointer queries are handled—given an IP address, return the name (or names) corresponding to that address.

First return to Figure 14.1 (p. 188) and examine the `arpa` top-level domain, and the `in-addr` domain beneath it. When an organization joins the Internet and obtains authority for a portion of the DNS name space, such as `noao.edu`, they also obtain authority for a portion of the `in-addr.arpa` name space corresponding to their IP address on the Internet. In the case of `noao.edu` it is the class B network ID 140.252. The level of the DNS tree beneath `in-addr.arpa` must be the first byte of the IP address (140 in this example), the next level is the next byte of the IP address (252), and so on. But remember that names are written starting at the bottom of the DNS tree, working upward. This means the DNS name for the host `sun`, with an IP address of 140.252.13.33, is `33.13.252.140.in-addr.arpa`.

We have to write the 4 bytes of the IP address backward because authority is delegated based on network IDs: the first byte of a class A address, the first and second

bytes of a class B address, and the first, second, and third bytes of a class C address. The first byte of the IP address must be immediately below the `in-addr` label, but FQDNs are written from the bottom of the tree up. If FQDNs were written from the top down, then the DNS name for the IP address would be `arpa.in-addr.140.252.13.33`, but the FQDN for the host would be `edu.noao.tuc.sun`.

If there was not a separate branch of the DNS tree for handling this address-to-name translation, there would be no way to do the reverse translation other than starting at the root of the tree and trying *every* top-level domain. This could literally take days or weeks, given the current size of the Internet. The `in-addr.arpa` solution is a clever one, although the reversed bytes of the IP address and the special domain are confusing.

Having to worry about the `in-addr.arpa` domain and reversing the bytes of the IP address affects us only if we're dealing directly with the DNS, using a program such as `host`, or watching the packets with `tcpdump`. From an application's point of view, the normal resolver function (`gethostbyaddr`) takes an IP address and returns information about the host. The reversal of the bytes and appending the domain `in-addr.arpa` are done automatically by this resolver function.

Example

Let's use the `host` program to do a pointer lookup and watch the packets with `tcpdump`. We'll use the same setup as in Figure 14.9, running the `host` program on the host `sun`, and the name server on the host `noao.edu`. We specify the IP address of our host `svr4`:

```
sun % host 140.252.13.34
Name: svr4.tuc.noao.edu
Address: 140.252.13.34
```

Since the only command-line argument is an IP address, the `host` program automatically generates the pointer query. Figure 14.12 shows the `tcpdump` output.

```
1  0.0                  140.252.1.29.1610 > 140.252.1.54.53: 1+ PTR?
                        34.13.252.140.in-addr.arpa. (44)

2  0.332288 (0.3323)    140.252.1.54.53 > 140.252.1.29.1610: 1* 1/0/0 PTR
                        svr4.tuc.noao.edu. (75)
```

Figure 14.12 `tcpdump` output for a pointer query.

Line 1 shows that the identifier is 1, the recursion-desired flag is set (the plus sign), and the query type is PTR. (Recall that the question mark means this is a query and not a response.) The data size of 44 bytes is from the 12-byte DNS header, 28 bytes for the 7 labels in the domain name, and 4 bytes for the query type and query class.

The reply has the authoritative-answer bit set (the asterisk) and contains one answer RR. The RR type is PTR and the resource data contains the domain name.

What is passed from the resolver to the name server for a pointer query is not a 32-bit IP address, but the domain name `34.13.252.140.in-addr.arpa`.

Hostname Spoofing Check

When an IP datagram arrives at a host for a server, be it a UDP datagram or a TCP connection request segment, all that's available to the server process is the client's IP address and port number (UDP or TCP). Some servers require the client's IP address to have a pointer record in the DNS. We'll see an example of this, using anonymous FTP from an unknown IP address, in Section 27.3.

Other servers, such as the Rlogin server (Chapter 26), not only require that the client's IP address have a pointer record, but then ask the DNS for the IP addresses corresponding to the name returned in the PTR response, and require that one of the returned addresses match the source IP address in the received datagram. This check is because entries in the `.rhosts` file (Section 26.2) contain the hostname, not an IP address, so the server wants to verify that the hostname really corresponds to the incoming IP address.

Some vendors automatically put this check into their resolver routines, specifically the function `gethostbyaddr`. This makes the check available to any program using the resolver, instead of manually placing the check in each application.

We can see an example of this using the SunOS 4.1.3 resolver library. We have written a simple program that performs a pointer query by calling the function `gethostbyaddr`. We have also set our `/etc/resolv.conf` file to use the name server on the host `noao.edu`, which is across the SLIP link from the host `sun`. Figure 14.13 shows the `tcpdump` output collected on the SLIP link when the function `gethostbyaddr` is called to fetch the name corresponding to the IP address 140.252.1.29 (our host `sun`).

```
1  0.0                  sun.1812 > noao.edu.domain: 1+ PTR?
                        29.1.252.140.in-addr.arpa. (43)
2  0.339091 (0.3391)    noao.edu.domain > sun.1812: 1* 1/0/0 PTR
                        sun.tuc.noao.edu. (73)

3  0.344348 (0.0053)    sun.1813 > noao.edu.domain: 2+ A?
                        sun.tuc.noao.edu. (33)
4  0.669022 (0.3247)    noao.edu.domain > sun.1813: 2* 2/0/0 A
                        140.252.1.29 (69)
```

Figure 14.13 Calling resolver function to perform pointer query.

Line 1 is the expected pointer query, and line 2 is the expected response. But the resolver function automatically sends an IP address query in line 3 for the name returned in line 2. The response in line 4 contains two answer records, since the host `sun` has two IP addresses. If one of the addresses does not match the argument to `gethostbyaddr`, a message is sent to the system logging facility, and the function returns an error to the application.

14.6 Resource Records

We've seen a few different types of resource records (RRs) so far: an IP address has a type of A, and PTR means a pointer query. We've also seen that RRs are what a name server returns: answer RRs, authority RRs, and additional information RRs. There are about 20 different types of resource records, some of which we'll now describe. Also, more RR types are being added over time.

A An A record defines an IP address. It is stored as a 32-bit binary value.

PTR This is the pointer record used for pointer queries. The IP address is represented as a domain name (a sequence of labels) in the `in-addr.arpa` domain.

CNAME This stands for "canonical name." It is represented as a domain name (a sequence of labels). The domain name that has a canonical name is often called an *alias*. These are used by some FTP sites to provide an easy to remember alias for some other system.

For example, the `gated` server (mentioned in Section 10.3) is available through anonymous FTP from the server `gated.cornell.edu`. But there is no system named `gated`, this is an alias for some other system. That other system is the canonical name for `gated.cornell.edu`:

```
sun % host -t cname gated.cornell.edu
gated.cornell.edu       CNAME   COMET.CIT.CORNELL.EDU
```

Here we use the `-t` option to specify one particular query type.

HINFO Host information: two arbitrary character strings specifying the CPU and operating system. Not all sites provide HINFO records for all their systems, and the information provided may not be up to date.

```
sun % host -t hinfo sun
sun.tuc.noao.edu        HINFO   Sun-4/25        Sun4.1.3
```

MX Mail exchange records, which are used in the following scenarios: (1) A site that is not connected to the Internet can get an Internet-connected site to be its mail exchanger. The two sites then work out an alternative way to exchange any mail that arrives, often using the UUCP protocol. (2) MX records provide a way to deliver mail to an alternative host when the destination host is not available. (3) MX records allow organizations to provide virtual hosts that one can send mail to, such as `cs.university.edu`, even if a host with that name doesn't exist. (4) Organizations with firewall gateways can use MX records to limit connectivity to internal systems.

Many sites that are not connected to the Internet have a UUCP link with an Internet connected site such as UUNET. MX records are then provided so that electronic mail can be sent to the site using the standard `user@host` notation. For example, a fictitious domain `foo.com` might have the following MX records:

```
sun % host -t mx foo.com
foo.com             MX      relay1.UU.NET
foo.com             MX      relay2.UU.NET
```

MX records are used by mailers on hosts connected to the Internet. In this example the other mailers are told "if you have mail to send to `user@foo.com`, send the mail to `relay1.uu.net` or `relay2.uu.net`."

MX records have 16-bit integers assigned to them, called *preference values*. If multiple MX records exist for a destination, they're used in order, starting with the smallest preference value.

Another example of MX records handles the case when a host is down or unavailable. In that case the mailer uses the MX records only if it can't connect to the destination using TCP. In the case of the author's primary system, which is connected to the Internet by a SLIP connection, which is down most of the time, we have:

```
sun % host -tv mx sun
Query about sun for record types MX
Trying sun within tuc.noao.edu ...
Query done, 2 answers, authoritative status: no error
sun.tuc.noao.edu    86400  IN    MX    0  sun.tuc.noao.edu
sun.tuc.noao.edu    86400  IN    MX    10 noao.edu
```

We also specified the `-v` option, to see the preference values. (This option also causes other fields to be output.) The second field, 86400, is the time-to-live value in seconds. This TTL is 24 hours ($24 \times 60 \times 60$). The third column, IN, is the class (Internet). We see that direct delivery to the host itself, the first MX record, has the lowest preference value of 0. If that doesn't work (i.e., the SLIP link is down), the next higher preference is used (10) and delivery is attempted to the host `noao.edu`. If that doesn't work, the sender will time out and retry at a later time.

In Section 28.3 we show examples of SMTP mail delivery using MX records.

NS Name server record. These specify the authoritative name server for a domain. They are represented as domain names (a sequence of labels). We'll see examples of these records in the next section.

These are the common types of RRs. We'll encounter many of them in later examples.

14.7 Caching

To reduce the DNS traffic on the Internet, all name servers employ a cache. With the standard Unix implementation, the cache is maintained in the server, not the resolver. Since the resolver is part of each application, and applications come and go, putting the cache into the program that lives the entire time the system is up (the name server) makes sense. This makes the cache available to any applications that use the server. Any other hosts at the site that use this name server also share the server's cache.

In the scenario that we've used for our examples so far (Figure 14.9), we've run the clients on the host `sun` accessing the name server across the SLIP link on the host `noao.edu`. We'll change that now and run the name server on the host `sun`. In this way if we monitor the DNS traffic on the SLIP link using `tcpdump`, we'll only see queries that can't be handled by the server out of its cache.

By default, the resolver looks for a name server on the local host (UDP port 53 or TCP port 53). We delete the `nameserver` directive from our resolver file, leaving only the `domain` directive:

```
sun % cat /etc/resolv.conf
domain tuc.noao.edu
```

The absence of a `nameserver` directive in this file causes the resolver to use the name server on the local host.

We then use the `host` command to execute the following query:

```
sun % host ftp.uu.net
ftp.uu.net              A       192.48.96.9
```

Figure 14.14 shows the `tcpdump` output for this query.

```
1   0.0                 sun.tuc.noao.edu.domain > NS.NIC.DDN.MIL.domain:
                        2 A? ftp.uu.net. (28)
2   0.559285 ( 0.5593)  NS.NIC.DDN.MIL.domain > sun.tuc.noao.edu.domain:
                        2- 0/5/5 (229)

3   0.564449 ( 0.0052)  sun.tuc.noao.edu.domain > ns.UU.NET.domain:
                        3+ A? ftp.uu.net. (28)
4   1.009476 ( 0.4450)  ns.UU.NET.domain > sun.tuc.noao.edu.domain:
                        3* 1/0/0 A ftp.UU.NET (44)
```

Figure 14.14 `tcpdump` output for: `host ftp.uu.net`.

This time we've used a new option for `tcpdump`. We collected all the data to or from UDP or TCP ports 53 with the `-w` option. This saves the raw output in a file for later processing. This prevents `tcpdump` from trying to call the resolver itself, to print all the names corresponding to the IP addresses. After we ran our queries, we terminated `tcpdump` and reran it with the `-r` option. This causes it to read the raw output file and generate its normal printed output (which we show in Figure 14.14). This takes a few seconds, since `tcpdump` calls the resolver itself.

The first thing to notice in our `tcpdump` output is that the identifiers are small integers (2 and 3). This is because we terminated the name server and then restarted it, to force the cache to be empty. When the name server starts up, it initializes the identifier to 1.

When we type our query, looking for the IP address of the host `ftp.uu.net`, the name server contacts one of the eight root servers, `ns.nic.ddn.mil` (line 1). This is the normal A type query that we've seen before, but notice that the recursion-desired flag is *not* specified. (A plus sign would have been printed after the identifier 2 if the flag was set.) In our earlier examples we always saw the resolver set the recursion-desired flag, but here we see that our name server doesn't set the flag when it's contacting one of the root servers. This is because the root servers shouldn't be asked to recursively answer queries—they should be used only to find the addresses of other, authoritative servers.

Line 2 shows that the response comes back with no answer RRs, five authority RRs, and five additional information RRs. The minus sign following the identifier 2 means the recursion-available (RA) flag was not set—this root server wouldn't answer a recursive query even if we asked it to.

Although `tcpdump` doesn't print the 10 RRs that are returned, we can execute the `host` command to see what's in the cache:

```
sun % host -v ftp.uu.net
Query about ftp.uu.net for record types A
Trying ftp.uu.net ...
Query done, 1 answer, status: no error
The following answer is not authoritative:
ftp.uu.net              19109   IN      A       192.48.96.9
Authoritative nameservers:
UU.NET                  170308  IN      NS      NS.UU.NET
UU.NET                  170308  IN      NS      UUNET.UU.NET
UU.NET                  170308  IN      NS      UUCP-GW-1.PA.DEC.COM
UU.NET                  170308  IN      NS      UUCP-GW-2.PA.DEC.COM
UU.NET                  170308  IN      NS      NS.EU.NET
Additional information:
NS.UU.NET               170347  IN      A       137.39.1.3
UUNET.UU.NET            170347  IN      A       192.48.96.2
UUCP-GW-1.PA.DEC.COM    170347  IN      A       16.1.0.18
UUCP-GW-2.PA.DEC.COM    170347  IN      A       16.1.0.19
NS.EU.NET               170347  IN      A       192.16.202.11
```

This time we specified the `-v` option to see more than just the A record. This shows that there are five authoritative name servers for the domain `uu.net`. The five RRs with additional information that are returned by the root server contain the IP addresses of these five name servers. This saves us from having to contact the root server again, to look up the address of one of the servers. This is another implementation optimization in the DNS.

The `host` command states that the answer is not authoritative. This is because the answer was obtained from our name server's cache, not by contacting an authoritative server.

Returning to line 3 of Figure 14.14, our name server contacts the first of the authoritative servers (`ns.uu.net`) with the same question: What is the IP address of

ftp.uu.net? This time our server sets the recursion-desired flag. The answer is returned on line 4 as a response with one answer RR.

We then execute the host command again, asking for the same name:

```
sun % host ftp.uu.net
ftp.uu.net              A       192.48.96.9
```

This time there is no tcpdump output. This is what we expect, since the answer output by host is returned from the server's cache.

We execute the host command again, looking for the address of ftp.ee.lbl.gov:

```
sun % host ftp.ee.lbl.gov
ftp.ee.lbl.gov          CNAME   ee.lbl.gov
ee.lbl.gov              A       128.3.112.20
```

Figure 14.15 shows the tcpdump output.

```
1  18.664971 (17.6555)   sun.tuc.noao.edu.domain > c.nyser.net.domain:
                         4 A? ftp.ee.lbl.gov. (32)
2  19.429412 ( 0.7644)   c.nyser.net.domain > sun.tuc.noao.edu.domain:
                         4 0/4/4 (188)
3  19.432271 ( 0.0029)   sun.tuc.noao.edu.domain > ns1.lbl.gov.domain:
                         5+ A? ftp.ee.lbl.gov. (32)
4  19.909242 ( 0.4770)   ns1.lbl.gov.domain > sun.tuc.noao.edu.domain:
                         5* 2/0/0 CNAME ee.lbl.gov. (72)
```

Figure 14.15 tcpdump output for: host ftp.ee.lbl.gov.

Line 1 shows that this time our server contacts another of the root servers (c.nyser.net). A name server normally cycles through the various servers for a zone until round-trip estimates are accumulated. The server with the smallest round-trip time is then used.

Since our server is contacting a root server, the recursion-desired flag is not set. This root server does not clear the recursion-available flag, as we saw in line 2 in Figure 14.14. (Even so, a name server still should not ask a root server for a recursive query.)

In line 2 the response comes back with no answers, but four authority RRs and four additional information RRs. As we can guess, the four authority RRs are the names of the name servers for ftp.ee.lbl.gov, and the four other RRs contain the IP addresses of these four servers.

Line 3 is the query of the name server ns1.lbl.gov (the first of the four name servers returned in line 2). The recursion-desired flag is set.

The response in line 4 is different from previous responses. Two answer RRs are returned and tcpdump says that the first one is a CNAME RR. The canonical name of ftp.ee.lbl.gov is ee.lbl.gov.

This is a common usage of CNAME records. The FTP site for LBL always has a name beginning with ftp, but it may move from one host to another over time. Users need only know the name ftp.ee.lbl.gov and the DNS will replace this with its canonical name when referenced.

Remember that when we ran `host`, it printed both the CNAME and the IP address of the canonical name. This is because the response (line 4 in Figure 14.15) contained two answer RRs. The first one is the CNAME and the second is the A record. If the A record had not been returned with the CNAME, our server would have issued another query, asking for the IP address of `ee.lbl.gov`. This is another implementation optimization—both the CNAME and the A record of the canonical name are returned in one response.

14.8 UDP or TCP

We've mentioned that the well-known port numbers for DNS name servers are UDP port 53 and TCP port 53. This implies that the DNS supports both UDP and TCP. But all the examples that we've watched with `tcpdump` have used UDP. When is each protocol used and why?

When the resolver issues a query and the response comes back with the TC bit set ("truncated") it means the size of the response exceeded 512 bytes, so only the first 512 bytes were returned by the server. The resolver normally issues the request again, using TCP. This allows more than 512 bytes to be returned. (Recall our discussion of the maximum UDP datagram size in Section 11.10.) Since TCP breaks up a stream of user data into what it calls *segments*, it can transfer any amount of user data, using multiple segments.

Also, when a secondary name server for a domain starts up it performs a zone transfer from the primary name server for the domain. We also said that the secondary queries the primary on a regular basis (often every 3 hours) to see if the primary has had its tables updated, and if so, a zone transfer is performed. Zone transfers are done using TCP, since there is much more data to transfer than a single query or response.

Since the DNS primarily uses UDP, both the resolver and the name server must perform their own timeout and retransmission. Also, unlike many other Internet applications that used UDP (TFTP, BOOTP, and SNMP), which operate mostly on local area networks, DNS queries and responses often traverse wide area networks. The packet loss rate and variability in round-trip times are normally higher on a WAN than a LAN, increasing the importance of a good retransmission and timeout algorithm for DNS clients.

14.9 Another Example

Let's look at another example that ties together many of the DNS features that we've described. We start an Rlogin client, connecting to an Rlogin server in some other domain. Figure 14.16 shows the exchange of packets that takes place. The following 11 steps take place, assuming none of the information is already cached by the client or server:

1. The client starts and calls its resolver function to convert the hostname that we typed into an IP address. A query of type A is sent to a root server.

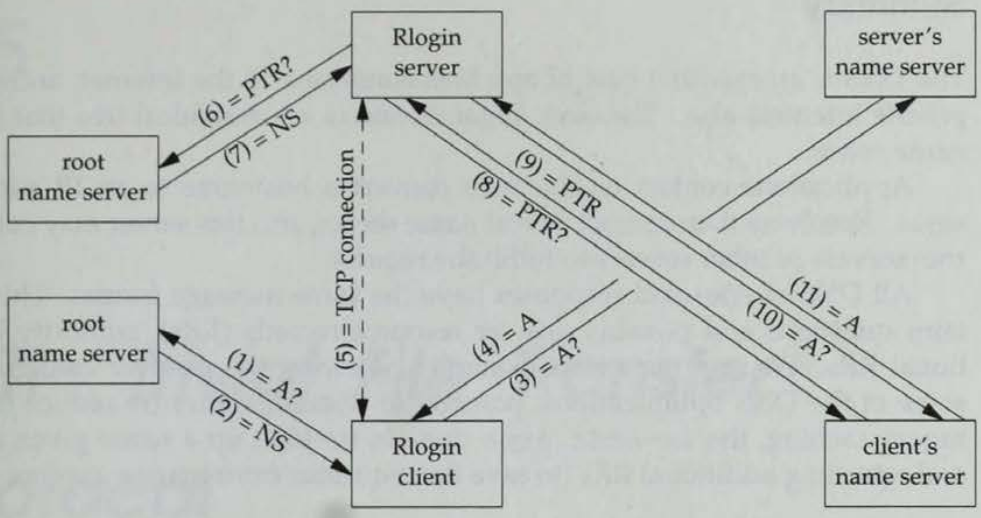


Figure 14.16 Summary of packets exchanged to start up Rlogin client and server.

2. The root server's response contains the name servers for the server's domain.
3. The client's resolver reissues the query of type A to the server's name server. This query normally has the recursion-desired flag set.
4. The response comes back with the IP address of the server host.
5. The Rlogin client establishes a TCP connection with the Rlogin server. (Chapter 18 provides all the details of this step.) Three packets are exchanged between the client and server TCP modules.
6. The Rlogin server receives the connection from the client and calls its resolver to obtain the name of the client host, given the IP address that the server receives from its TCP. This is a PTR query issued to a root name server. This root server can be different from the root server used by the client in step 1.
7. The root server's response contains the name servers for the client's `in-addr.arpa` domain.
8. The server's resolver reissues the PTR query to the client's name server.
9. The PTR response contains the FQDN of the client host.
10. The server's resolver issues a query of type A to the client's name server, asking for the IP addresses corresponding to the name returned in the previous step. This may be done automatically by the server's `gethostbyaddr` function, as we described in Section 14.5, otherwise the Rlogin server does this step explicitly. Also, the client's name server is often the same as the client's `in-addr.arpa` name server, but this isn't required.
11. The response from the client's name server contains the A records for the client host. The Rlogin server compares the A records with the IP address from the client's TCP connection request.

Caching can reduce the number of packets exchanged in this figure.

14.10 Summary

The DNS is an essential part of any host connected to the Internet, and widely used in private internets also. The basic organization is a hierarchical tree that forms the DNS name space.

Applications contact resolvers to convert a hostname to an IP address, and vice versa. Resolvers then contact a local name server, and this server may contact one of the root servers or other servers to fulfill the request.

All DNS queries and responses have the same message format. This message contains questions and possibly answer resource records (RRs), authority RRs, and additional RRs. We saw numerous examples, showing the resolver configuration file and some of the DNS optimizations: pointers to domain names (to reduce the size of messages), caching, the `in-addr.arpa` domain (to look up a name given an IP address), and returning additional RRs (to save the requestor from issuing another query).

Exercises

14.1 Classify a DNS resolver and a DNS name server as either client, server, or both.

14.2 Account for all 75 bytes in the response in Figure 14.12.

14.3 In Section 12.3 we said that an application that accepts either a dotted-decimal IP address or a hostname should assume the former, and if that fails, then assume a hostname. What happens if the order of the tests is reversed?

14.4 Every UDP datagram has an associated length. A process that receives a UDP datagram is told what its length is. When a resolver issues a query using TCP instead of UDP, since TCP is a stream of bytes without any record markers, how does the application know how much data is returned? Notice that there is no length field in the DNS header (Figure 14.3). (*Hint*: Look at RFC 1035.)

14.5 We said that a name server must know the IP addresses of the root servers and that this information is available via anonymous FTP. Unfortunately not all system administrators update their DNS files whenever changes are made to the list of root servers. (Changes do occur to the list of root servers, but not frequently.) How do you think the DNS handles this?

14.6 Fetch the file specified in Exercise 1.8 and determine who is responsible for maintaining the root name servers. How frequently are the root servers updated?

14.7 What is a problem with maintaining the cache in the name server, and having a stateless resolver?

14.8 In the discussion of Figure 14.10 we said that the name server sorts the A records so that addresses on common networks appear first. Who should sort the A records, the name server or the resolver?

15

TFTP: Trivial File Transfer Protocol

15.1 Introduction

TFTP is the Trivial File Transfer Protocol. It is intended to be used when bootstrapping diskless systems (normally workstations or X terminals). Unlike the File Transfer Protocol (FTP), which we describe in Chapter 27 and which uses TCP, TFTP was designed to use UDP, to make it simple and small. Implementations of TFTP (and its required UDP, IP, and a device driver) can fit in read-only memory.

This chapter provides an overview of TFTP because we'll encounter it in the next chapter with the Bootstrap Protocol. We also encountered TFTP in Figure 5.1 when we bootstrapped the host `sun` from the network. It issued a TFTP request after obtaining its IP address using RARP.

RFC 1350 [Sollins 1992] is the official specification of version 2 of TFTP. Chapter 12 of [Stevens 1990] provides a complete source code implementation of a TFTP client and server, and describes some of the programming techniques used with TFTP.

15.2 Protocol

Each exchange between a client and server starts with the client asking the server to either read a file for the client or write a file for the client. In the normal case of bootstrapping a diskless system, the first request is a read request (RRQ). Figure 15.1 shows the format of the five TFTP messages. (Opcodes 1 and 2 share the same format.)

The first 2 bytes of the TFTP message are an *opcode*. For a read request (RRQ) and write request (WRQ) the *filename* specifies the file on the server that the client wants to read from or write to. We specifically show that this filename is terminated by a byte of 0 in Figure 15.1. The *mode* is one of the ASCII strings `netascii` or `octet` (in any

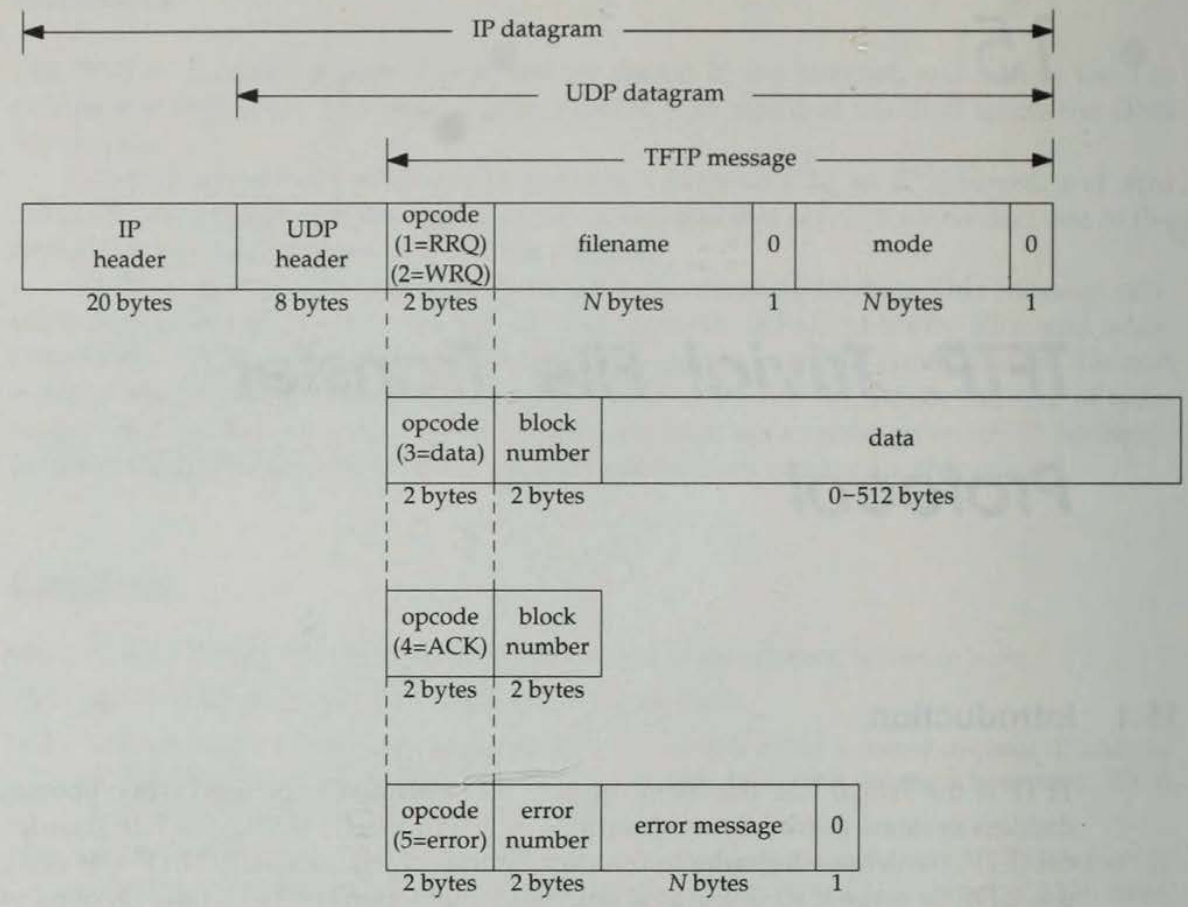


Figure 15.1 Format of the five TFTP messages.

combination of uppercase or lowercase), again terminated by a byte of 0. `netascii` means the data are lines of ASCII text with each line terminated by the 2-character sequence of a carriage return followed by a linefeed (called CR/LF). Both ends must convert between this format and whatever the local host uses as a line delimiter. An `octet` transfer treats the data as 8-bit bytes with no interpretation.

Each data packet contains a *block number* that is later used in an acknowledgment packet. As an example, when reading a file the client sends a read request (RRQ) specifying the filename and mode. If the file can be read by the client, the server responds with a data packet with a block number of 1. The client sends an ACK of block number 1. The server responds with the next data packet, with a block number of 2. The client sends an ACK of block number 2. This continues until the file is transferred. Each data packet contains 512 bytes of data, except for the final packet, which contains 0–511 bytes of data. When the client receives a data packet with less than 512 bytes of data, it knows it has received the final packet.

In the case of a write request (WRQ), the client sends the WRQ specifying the filename and mode. If the file can be written by the client, the server responds with an ACK of block number 0. The client then sends the first 512 bytes of file with a block number of 1. The server responds with an ACK of block number 1.

This type of data transmission is called a *stop-and-wait* protocol. It is found only in simple protocols such as TFTP. We'll see in Section 20.3 that TCP provides a different form of acknowledgment, which can provide higher throughput. TFTP is designed for simplicity of implementation, not high throughput.

The final TFTP message type is the error message, with an *opcode* of 5. This is what the server responds with if a read request or write request can't be processed. Read and write errors during file transmission also cause this message to be sent, and transmission is then terminated. The *error number* gives a numeric error code, followed by an ASCII error message that might contain additional, operating system specific information.

Since TFTP uses the unreliable UDP, it is up to TFTP to handle lost and duplicated packets. Lost packets are detected with a timeout and retransmission implemented by the sender. (Be aware of a potential problem called the "sorcerer's apprentice syndrome" that can occur if both sides time out and retransmit. Section 12.2 of [Stevens 1990] shows how the problem can occur.) As with most UDP applications, there is no checksum in the TFTP messages, which assumes any corruption of the data will be caught by the UDP checksum (Section 11.3).

15.3 An Example

Let's examine TFTP by watching the protocol in action. We'll run the TFTP client on the host `bsdi` and fetch a text file from the host `svr4`:

```
bsdi % tftp svr4                          start the TFTP client
tftp> get test1.c                         fetch a file from the server
Received 962 bytes in 0.3 seconds
tftp> quit                                and terminate

bsdi % ls -l test1.c                      how many bytes in the file we fetched?
-rw-r--r--  1 rstevens  staff  914 Mar 20 11:41 test1.c

bsdi % wc -l test1.c                      and how many lines?
      48 test1.c
```

The first point that catches our eye is that the file contains 914 bytes under Unix, but TFTP transfers 962 bytes. Using the `wc` program we see that there are 48 lines in the file, so the 48 Unix newline characters are expanded into 48 CR/LF pairs, since the TFTP default is a `netascii` transfer.

Figure 15.2 shows the packet exchange that takes place.

```
1  0.0                  bsdi.1106 > svr4.tftp: 19 RRQ "test1.c"

2  0.287080 (0.2871)    svr4.1077 > bsdi.1106: udp 516
3  0.291178 (0.0041)    bsdi.1106 > svr4.1077: udp 4

4  0.299446 (0.0083)    svr4.1077 > bsdi.1106: udp 454
5  0.312320 (0.0129)    bsdi.1106 > svr4.1077: udp 4
```

Figure 15.2 Packet exchange for TFTP of a file.

Line 1 shows the read request from the client to the server. Since the destination UDP port is the TFTP well-known port (69), `tcpdump` interprets the TFTP packet and prints `RRQ` and the name of the file. The length of the UDP data is printed as 19 bytes and is accounted for as follows: 2 bytes for the opcode, 7 bytes for the filename, 1 byte of 0, 8 bytes for `netascii`, and another byte of 0.

The next packet is from the server (line 2) and contains 516 bytes: 2 bytes for the opcode, 2 bytes for the block number, and 512 bytes of data. Line 3 is the acknowledgment for this data: 2 bytes for the opcode and 2 bytes for the block number.

The final data packet (line 4) contains 450 bytes of data. The 512 bytes of data in line 2 and this 450 bytes of data account for the 962 bytes of data output by the client.

Note that `tcpdump` doesn't output any additional TFTP protocol information for lines 2–5, whereas it interpreted the TFTP message in line 1. This is because the server's port number changes between lines 1 and 2. The TFTP protocol requires that the client send the first packet (the RRQ or WRQ) to the server's well-known UDP port (69). The server then allocates some other unused ephemeral port on the server's host (1077 in Figure 15.2), which is then used by the server for all further packet exchange between this client and server. The client's port number (1106 in this example) doesn't change. `tcpdump` has no idea that port 1077 on host `svr4` is really a TFTP server.

The reason the server's port number changes is so the server doesn't tie up the well-known port for the amount of time required to transfer the file (which could be many seconds or even minutes). Instead, the well-known port is left available for other TFTP clients to send their requests to, while the current transfer is under way.

Recall from Figure 10.6 (p. 133) that when the RIP server had more than 512 bytes to send to the client, both UDP datagrams came from the server's well-known port. In that example, even though the server had to write multiple datagrams to send all the data back, the server did one write, followed by the next, both from its well-known port. Here, with TFTP, the protocol is different since there is a longer term relationship between the client and server (which we said could be seconds or minutes). If one server process used the well-known port for the duration of the file transfer, it would either have to refuse any further requests that arrived from other clients, or that one server process would have to multiplex file transfers with multiple clients at the same time, on the same port (69). The simplest solution is to have the server obtain a new port after it receives the RRQ or WRQ. Naturally the client must detect this new port when it receives the first data packet (line 2 in Figure 15.2) and then send all further acknowledgments (lines 3 and 5) to that new port.

In Section 16.3 we'll see TFTP used when an X terminal is bootstrapped.

15.4 Security

Notice in the TFTP packets (Figure 15.1) that there is no provision for a username or password. This is a feature (i.e., "security hole") of TFTP. Since TFTP was designed for use during the bootstrap process it could be impossible to provide a username and password.

This feature of TFTP was used by many crackers to obtain copies of a Unix password file and then try to guess passwords. To prevent this type of access, most TFTP servers nowadays provide an option whereby only files in a specific directory (often `/tftpboot` on Unix systems) can be accessed. This directory then contains only the bootstrap files required by the diskless systems.

For additional security, the TFTP server on a Unix system normally sets its user ID and group ID to values that should not be assigned to any real user. This allows access only to files that have world-read or world-write permissions.

15.5 Summary

TFTP is a simple protocol designed to fit into read-only memory and be used only during the bootstrap process of diskless systems. It uses only a few message formats and a stop-and-wait protocol.

To allow multiple clients to bootstrap at the same time, a TFTP server needs to provide some form of concurrency. Because UDP does not provide a unique connection between a client and server (as does TCP), the TFTP server provides concurrency by creating a new UDP port for each client. This allows different client input datagrams to be demultiplexed by the server's UDP module, based on destination port numbers, instead of doing this in the server itself.

The TFTP protocol provides no security features. Most implementations count on the system administrator of the TFTP server to restrict any client's access to the files necessary for bootstrapping only.

Chapter 27 covers the File Transfer Protocol (FTP), which is designed for general purpose, high-throughput file transfer.

Exercises

15.1 Read the Host Requirements RFC to see what a TFTP server should do if it receives a request and the destination IP address of the request is a broadcast address.

15.2 What do you think happens when the TFTP block number wraps around from 65535 to 0? Does RFC 1350 say anything about this?

15.3 We said that the TFTP sender performs the timeout and retransmission to handle lost packets. How does this affect the use of TFTP when it's being used as part of the bootstrap process?

15.4 What is the limiting factor in the time required to transfer a file using TFTP?

16

BOOTP: Bootstrap Protocol

16.1 Introduction

In Chapter 5 we described how a diskless system, with no knowledge of its IP address, can determine its IP address using RARP when it is bootstrapped. There are two problems with RARP: (1) the only thing returned is the IP address, and (2) since RARP uses a link-layer broadcast, RARP requests are not forwarded by routers (necessitating an RARP server on every physical network). This chapter describes an alternative method for a diskless system to bootstrap itself, called the Bootstrap Protocol, or BOOTP.

BOOTP uses UDP and normally works in conjunction with TFTP (Chapter 15). RFC 951 [Croft and Gilmore 1985] is the official specification for BOOTP with clarifications given in RFC 1542 [Wimer 1993].

16.2 BOOTP Packet Format

BOOTP requests and replies are encapsulated in UDP datagrams, as shown in Figure 16.1.

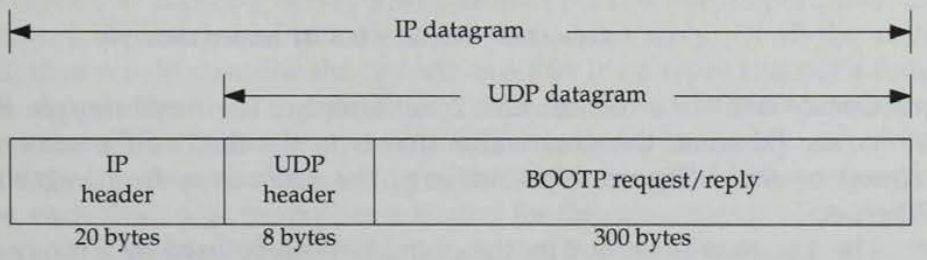


Figure 16.1 Encapsulation of BOOTP requests and replies within a UDP datagram.

Figure 16.2 shows the format of the 300-byte BOOTP request and reply.

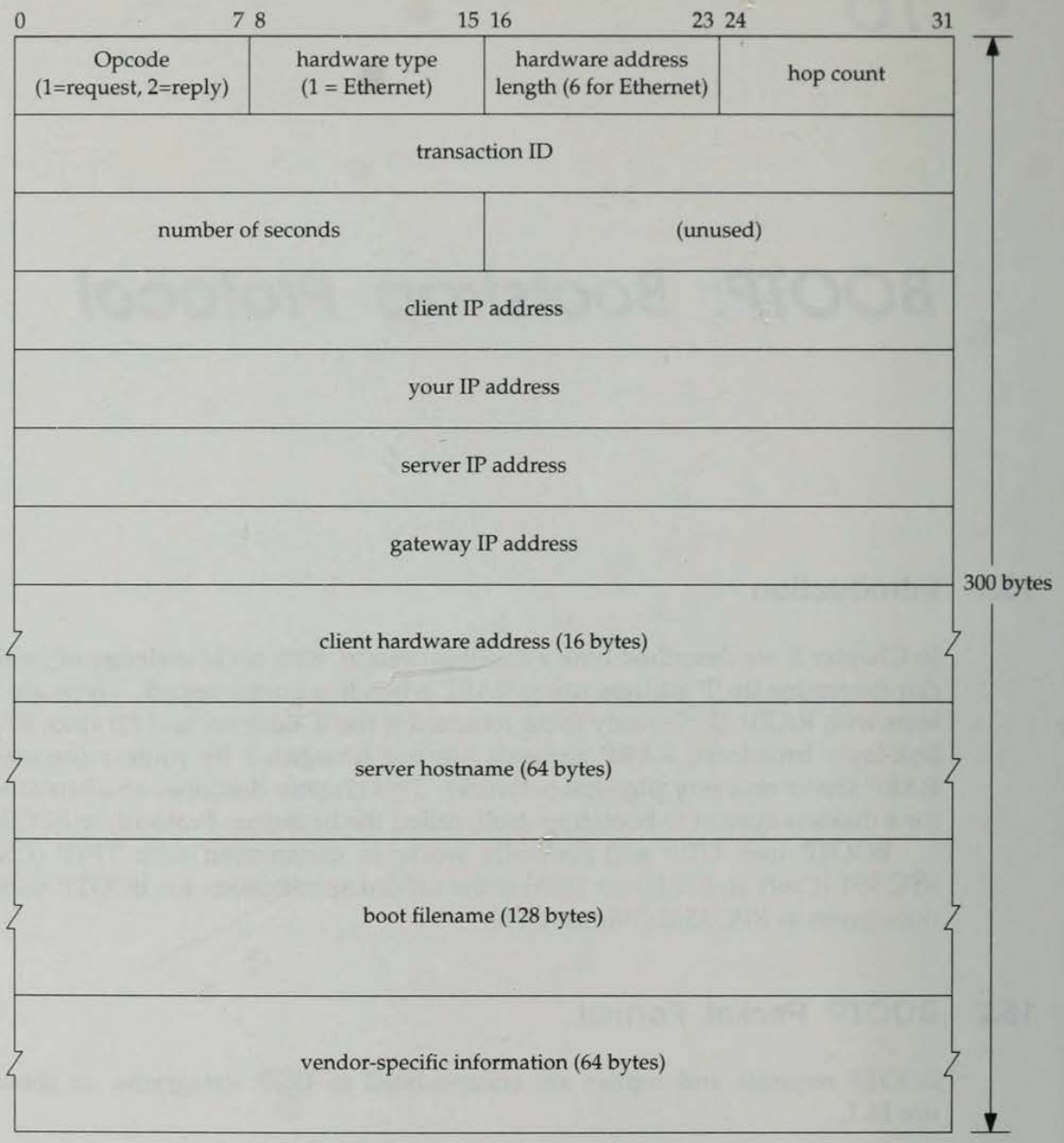


Figure 16.2 Format of BOOTP request and reply.

Opcode is 1 for a request and 2 for a reply. The *hardware type* field is 1 for a 10 Mbits/sec Ethernet, the same value that is in the field of the same name in an ARP request or reply (Figure 4.3). Similarly, the *hardware address length* is 6 bytes for an Ethernet.

The *hop count* is set to 0 by the client, but can be used by a proxy server (described in Section 16.5).

The *transaction ID* is a 32-bit integer set by the client and returned by the server. This lets the client match a response with a request. The client should set this to a random number for each request.

Number of seconds can be set by the client to the time since it started trying to bootstrap. The servers can look at this value, and perhaps a secondary server for a client won't respond until the number of seconds has exceeded some value, implying that the client's primary server is down.

If the client already knows its IP address, it fills in the *client IP address*. Otherwise, the client sets this to 0. In the latter case the server fills in *your IP address* with the client's IP address. The *server IP address* is filled in by the server. If a proxy server is used (Section 16.5), that proxy server fills in its *gateway IP address*.

The client must set its *client hardware address*. Although this is the same value as in the Ethernet header, by placing the field in the UDP datagram also, it is easily available to any user process (e.g., a BOOTP server) that receives the datagram. It is normally much harder (or impossible) for a process reading UDP datagrams to determine the fields in the Ethernet header that carried the UDP datagram.

The *server hostname* is a null terminated string that is optionally filled in by the server. The server can also fill in the *boot filename* with the fully qualified, null terminated pathname of a file to bootstrap from.

The *vendor-specific area* is used for various extensions to BOOTP. Section 16.6 describes some of these extensions.

When a client is bootstrapping using BOOTP (an opcode of 1) the request is normally a link-layer broadcast and the destination IP address in the IP header is normally 255.255.255.255 (the limited broadcast, Section 12.2). The source IP address is often 0.0.0.0 since the client does not know its own IP address yet. Recall from Figure 3.9 that 0.0.0.0 is a valid source IP address when a system is bootstrapping itself.

Port Numbers

There are two well-known ports for BOOTP: 67 for the server and 68 for the client. This means the client does not choose an unused ephemeral port, but uses 68 instead. The reason two port numbers were chosen, instead of just one for the server, is that a server's reply can be (but normally isn't) broadcast.

If the server's reply were broadcast, and if the client were to choose an ephemeral port number, these broadcasts would also be received by other applications on other hosts that happen to be using the same ephemeral port number. Hence, it is considered bad form to broadcast to a random (i.e., ephemeral) port number.

If the client also used the server's well-known port (67) as its port, then all servers on the network are awakened to look at each broadcast reply. (If all the servers were awakened, they would examine the opcode, see that it's a reply and not a request, and go back to sleep.) Therefore the choice was made to have all clients use a single well-known port that differs from the server's well-known port.

If multiple clients are bootstrapping at the same time, and if the server broadcasts the replies, each client sees the replies intended for the other clients. The clients can use the transaction ID field in the BOOTP header to match replies with requests, or the client can examine the returned client hardware address.

16.3 An Example

Let's look at an example of BOOTP when an X terminal is bootstrapped. Figure 16.3 shows the `tcpdump` output. (The client's name is `proteus` and the server's name is `mercury`. This `tcpdump` output was obtained on a different network from the one we've been using for all the other examples in the text.)

```
1   0.0                  0.0.0.0.68 > 255.255.255.255.bootp:
                         secs:100 ether 0:0:a7:0:62:7c
2   0.355446 (0.3554)    mercury.bootp > proteus.68: secs:100 Y:proteus
                         S:mercury G:mercury ether 0:0:a7:0:62:7c
                         file "/local/var/bootfiles/Xncd19r"
3   0.355447 (0.0000)    arp who-has proteus tell 0.0.0.0
4   0.851508 (0.4961)    arp who-has proteus tell 0.0.0.0
5   1.371070 (0.5196)    arp who-has proteus tell proteus
6   1.863226 (0.4922)    proteus.68 > 255.255.255.255.bootp:
                         secs:100 ether 0:0:a7:0:62:7c
7   1.871038 (0.0078)    mercury.bootp > proteus.68: secs:100 Y:proteus
                         S:mercury G:mercury ether 0:0:a7:0:62:7c
                         file "/local/var/bootfiles/Xncd19r"
8   3.871038 (2.0000)    proteus.68 > 255.255.255.255.bootp:
                         secs:100 ether 0:0:a7:0:62:7c
9   3.878850 (0.0078)    mercury.bootp > proteus.68: secs:100 Y:proteus
                         S:mercury G:mercury ether 0:0:a7:0:62:7c
                         file "/local/var/bootfiles/Xncd19r"
10  5.925786 (2.0469)    arp who-has mercury tell proteus
11  5.929692 (0.0039)    arp reply mercury is-at 8:0:2b:28:eb:1d
12  5.929694 (0.0000)    proteus.tftp > mercury.tftp: 37 RRQ
                         "/local/var/bootfiles/Xncd19r"
13  5.996094 (0.0664)    mercury.2352 > proteus.tftp: 516 DATA block 1
14  6.000000 (0.0039)    proteus.tftp > mercury.2352: 4 ACK
                                  many lines deleted here
15  14.980472 (8.9805)   mercury.2352 > proteus.tftp: 516 DATA block 2463
16  14.984376 (0.0039)   proteus.tftp > mercury.2352: 4 ACK
17  14.984377 (0.0000)   mercury.2352 > proteus.tftp: 228 DATA block 2464
18  14.984378 (0.0000)   proteus.tftp > mercury.2352: 4 ACK
```

Figure 16.3 Example of BOOTP being used to bootstrap an X terminal.

In line 1 we see the client's request from 0.0.0.0.68, destined for 255.255.255.255.67. The only fields the client has filled in are the number of seconds and its Ethernet address. We'll see that this client always sets the number of seconds to 100. The hop count and transaction ID are both 0 since they are not output by `tcpdump`. (A transaction ID of 0 means the client ignores the field, since it would set this field to a random number if it was going to verify the returned value in the response.)

Line 2 is the reply from the server. The fields filled in by the server are the client's IP address (which `tcpdump` prints as the name `proteus`), the server's IP address (printed as the name `mercury`), the IP address of a gateway (printed as the name `mercury`), and the name of a boot file.

After receiving the BOOTP reply, the client immediately issues an ARP request to see if anyone else on the network has its IP address. The name `proteus` following `who-has` corresponds to the target IP address (Figure 4.3), and the sender's IP address is set to 0.0.0.0. It sends another identical ARP request 0.5 second later, and another one 0.5 second after that. In the third ARP request (line 5) it changes the sender's IP address to be its own IP address. This is a gratuitous ARP request (Section 4.7).

Line 6 shows that the client waits another 0.5 second and broadcasts another BOOTP request. The only difference between this request and line 1 is that now the client puts its own IP address in the IP header. It receives the same reply from the same server (line 7). The client waits another 2 seconds and broadcasts yet another BOOTP request (line 8) and receives the same reply from the same server.

The client then waits another 2 seconds and sends an ARP request for its server `mercury` (line 10). The ARP reply is received and the client immediately issues a TFTP read request for its boot file (line 12). What follows are 2464 TFTP data packets and acknowledgments. The amount of data transferred is $512 \times 2463 + 224 = 1,261,280$ bytes. This loads the operating system into the X terminal. We have deleted most of the TFTP lines from Figure 16.3.

One thing to notice, when comparing this TFTP exchange with Figure 15.2, is that here the client uses the TFTP well-known port (69) for the entire transfer. Since one of the two partners is using port 69, `tcpdump` knows that the packets are TFTP messages, so it is able to interpret each packet using the TFTP protocol. This is why Figure 16.3 indicates which packets contain data, which contain acknowledgments, and what the block number is for each packet. We didn't get this additional information in Figure 15.2 because neither end was using TFTP's well-known port for the data transfer. Normally the TFTP client cannot use TFTP's well-known port, since that port is used by the server on a multiuser system. But here the system is being bootstrapped, so a TFTP server is not provided, allowing the client to use the port for the duration of the transfer. This also implies that the TFTP server on `mercury` doesn't care what the client's port number is—it sends the data to the client's port, whatever that happens to be.

From Figure 16.3 we see that 1,261,280 bytes are transferred in 9 seconds. This is a rate of about 140,000 bytes per second. While this is slower than most FTP file transfers across an Ethernet, it is not that bad for a simple stop-and-wait protocol such as TFTP.

What follows as this X terminal is bootstrapped are additional TFTP transfers of the terminal's font files, some DNS name server queries, and then the initialization of the X protocol. The total time in Figure 16.3 was almost 15 seconds, and another 6 seconds is taken for the remaining steps. This gives a total of 21 seconds to bootstrap the diskless X terminal.

16.4 BOOTP Server Design

The BOOTP client is normally provided in read-only memory on the diskless system. It is interesting to see how the server is normally implemented.

First, the server reads UDP datagrams from its well-known port (67). Nothing special is required. This differs from an RARP server (Section 5.4), which we said had to read Ethernet frames with a type field of "RARP request." The BOOTP protocol also

made it easy for the server to obtain the client's hardware address, by placing it into the BOOTP packet (Figure 16.2).

An interesting problem arises: how can the server send a response directly back to the client? The response is a UDP datagram, and the server knows the client's IP address (probably read from a configuration file on the server). But if the client sends a UDP datagram to that IP address (the normal way UDP output is handled), the server's host will probably issue an ARP request for that IP address. But the client can't respond to the ARP request since it doesn't know its IP address yet! (This is called the "chicken and egg" issue in RFC 951.)

There are two solutions. The first, commonly used by Unix servers, is for the server to issue an `ioctl`(2) request to the kernel, to place an entry into the ARP cache for this client. (This is what the `arp -s` command does, Section 4.8.) The server can do this since it knows the client's hardware address and IP address. This means that when the server sends the UDP datagram (the BOOTP reply), the server's ARP module will find the client's IP address in the ARP cache.

An alternative solution is for the server to broadcast the BOOTP reply, instead of sending it directly to the client. Since reducing the number of broadcasts on a network is always desirable, this solution should be used only if the server cannot make an entry into its ARP cache. Normally it requires superuser permission to make an entry into the ARP cache, requiring a broadcast reply if the server is nonprivileged.

16.5 BOOTP Through a Router

We said in Section 5.4 that one of the drawbacks of RARP is that it uses a link-layer broadcast, which is normally not forwarded by a router. This required an RARP server on each physical network. BOOTP can be used through a router, if supported by the router. (Most major router vendors do support this feature.)

This is mainly intended for diskless routers, because if a multiuser system with a disk is used as a router, it can probably run a BOOTP server itself. Alternatively, the common Unix BOOTP server (Appendix F) supports this relay mode, but again, if you can run a BOOTP server on the physical network, there's normally no need to forward the requests to yet another server on another network.

What happens is that the router (also called the "BOOTP relay agent") listens for BOOTP requests on the server's well-known port (67). When a request is received, the relay agent places its IP address into the *gateway IP address* field in the BOOTP request, and sends the request to the real BOOTP server. (The address placed by the relay agent into the gateway field is the IP address of the interface on which the request was received.) The relay agent also increments the hops field by one. (This is to prevent infinite loops in case the request is reforwarded. RFC 951 mentions that the request should probably be thrown away if the hop count reaches 3.) Since the outgoing request is a unicast datagram (as opposed to the original client request that was broadcast), it can follow any route to the real BOOTP server, passing through other routers. The real server gets the request, forms the BOOTP reply, and sends it back to the relay agent, not the client. The real server knows that the request has been forwarded, since the gateway field in the request is nonzero. The relay agent receives the reply and sends it to the client.

16.6 Vendor-Specific Information

In Figure 16.2 we showed a 64-byte vendor-specific area. RFC 1533 [Alexander and Droms 1993] defines the format of this area. This area contains optional information for the server to return to the client.

If information is provided, the first 4 bytes of this area are set to the IP address 99.130.83.99. This is called the *magic cookie* and means there is information in the area.

The rest of this area is a list of items. Each item begins with a 1-byte tag field. Two of the items consist of just the tag field: a tag of 0 is a pad byte (to force following items to preferred byte boundaries), and a tag of 255 marks the end of the items. Any bytes remaining in this area after the first end byte should be set to this value (255).

Other than these two 1-byte items, the remaining items consist of a single length byte, followed by the information. Figure 16.4 shows the format of some of the items in the vendor-specific area.

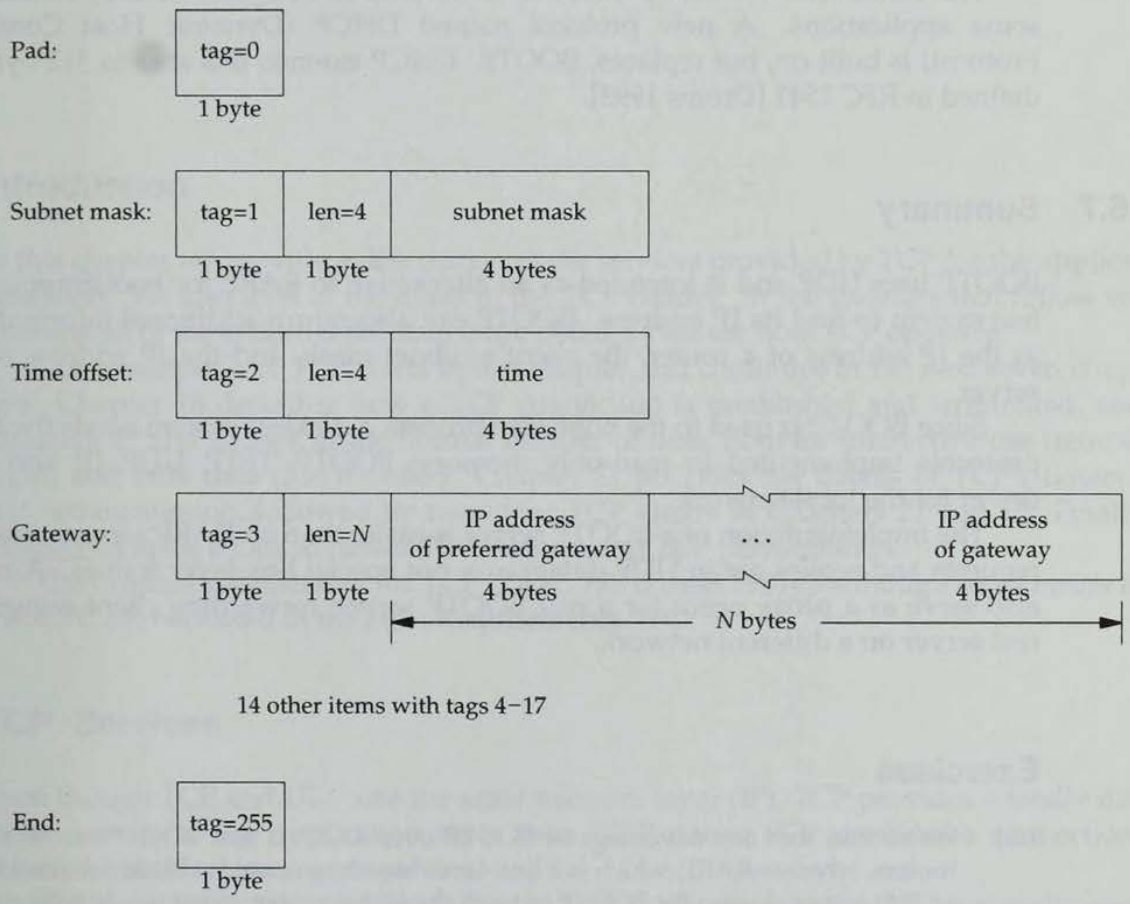


Figure 16.4 Format of some of the items in the vendor-specific area.

The subnet mask and time value are really fixed-length items because their values always occupy 4 bytes. The time offset is the number of seconds since midnight January 1, 1900, UTC.

The gateway item is an example of a variable-length item. The length is always a multiple of 4, and the values are the 32-bit IP addresses of one or more gateways (routers) for the client to use. The first one returned must be the preferred gateway.

There are 14 other items defined in RFC 1533. Probably the most important is the IP address of a DNS name server, with a tag value of 6. Other items return the IP address of a printer server, the IP address of a time server, and so on. Refer to the RFC for all the details.

Returning to our example in Figure 16.3, we never saw an ICMP address mask request (Section 6.3) that would have been broadcast by the client to find its subnet mask. Although it wasn't output by `tcpdump`, we can probably assume that the client's subnet mask was returned in the vendor-specific area of the BOOTP reply.

> The Host Requirements RFC recommends that a system using BOOTP obtain its subnet mask using BOOTP, not ICMP.

The size of the vendor-specific area is limited to 64 bytes. This is a constraint for some applications. A new protocol named DHCP (Dynamic Host Configuration Protocol) is built on, but replaces, BOOTP. DHCP extends this area to 312 bytes and is defined in RFC 1541 [Droms 1993].

16.7 Summary

BOOTP uses UDP and is intended as an alternative to RARP for bootstrapping a diskless system to find its IP address. BOOTP can also return additional information, such as the IP address of a router, the client's subnet mask, and the IP address of a name server.

Since BOOTP is used in the bootstrap process, a diskless system needs the following protocols implemented in read-only memory: BOOTP, TFTP, UDP, IP, and a device driver for the local network.

The implementation of a BOOTP server is easier than an RARP server, since BOOTP requests and replies are in UDP datagrams, not special link-layer frames. A router can also serve as a proxy agent for a real BOOTP server, forwarding client requests to the real server on a different network.

Exercises

16.1 We've said that one advantage of BOOTP over RARP is that BOOTP can work through routers, whereas RARP, which is a link-layer broadcast, cannot. Yet in Section 16.5 we had to define special ways for BOOTP to work through a router. What would happen if a capability were added to routers allowing them to forward RARP requests?

16.2 We said that a BOOTP client must use the transaction ID to match responses with requests, in case there are multiple clients bootstrapping at the same time from a server that broadcasts replies. But in Figure 16.3 the transaction ID is 0, implying that this client ignores the transaction ID. How do you think this client matches the responses with its requests?

17

TCP: Transmission Control Protocol

17.1 Introduction

In this chapter we provide a description of the services provided by TCP for the application layer. We also look at the fields in the TCP header. In the chapters that follow we examine all of these header fields in more detail, as we see how TCP operates.

Our description of TCP starts in this chapter and continues in the next seven chapters. Chapter 18 describes how a TCP connection is established and terminated, and Chapters 19 and 20 look at the normal transfer of data, both for interactive use (remote login) and bulk data (file transfer). Chapter 21 provides the details of TCP's timeout and retransmission, followed by two other TCP timers in Chapters 22 and 23. Finally Chapter 24 takes a look at newer TCP features and TCP performance.

The original specification for TCP is RFC 793 [Postel 1981c], although some errors in that RFC are corrected in the Host Requirements RFC.

17.2 TCP Services

Even though TCP and UDP use the same network layer (IP), TCP provides a totally different service to the application layer than UDP does. TCP provides a connection-oriented, reliable, byte stream service.

The term *connection-oriented* means the two applications using TCP (normally considered a client and a server) must establish a TCP connection with each other before they can exchange data. The typical analogy is dialing a telephone number, waiting for the other party to answer the phone and say "hello," and then saying who's calling. In Chapter 18 we look at how a connection is established, and disconnected some time later when either end is done.

There are exactly two end points communicating with each other on a TCP connection. Concepts that we talked about in Chapter 12, broadcasting and multicasting, aren't applicable to TCP.

TCP provides *reliability* by doing the following:

- The application data is broken into what TCP considers the best sized chunks to send. This is totally different from UDP, where each write by the application generates a UDP datagram of that size. The unit of information passed by TCP to IP is called a *segment*. (See Figure 1.7, p. 10.) In Section 18.4 we'll see how TCP decides what this segment size is.
- When TCP sends a segment it maintains a timer, waiting for the other end to acknowledge reception of the segment. If an acknowledgment isn't received in time, the segment is retransmitted. In Chapter 21 we'll look at TCP's adaptive timeout and retransmission strategy.
- When TCP receives data from the other end of the connection, it sends an acknowledgment. This acknowledgment is not sent immediately, but normally delayed a fraction of a second, as we discuss in Section 19.3.
- TCP maintains a checksum on its header and data. This is an end-to-end checksum whose purpose is to detect any modification of the data in transit. If a segment arrives with an invalid checksum, TCP discards it and doesn't acknowledge receiving it. (It expects the sender to time out and retransmit.)
- Since TCP segments are transmitted as IP datagrams, and since IP datagrams can arrive out of order, TCP segments can arrive out of order. A receiving TCP resequences the data if necessary, passing the received data in the correct order to the application.
- Since IP datagrams can get duplicated, a receiving TCP must discard duplicate data.
- TCP also provides flow control. Each end of a TCP connection has a finite amount of buffer space. A receiving TCP only allows the other end to send as much data as the receiver has buffers for. This prevents a fast host from taking all the buffers on a slower host.

A stream of 8-bit bytes is exchanged across the TCP connection between the two applications. There are no record markers automatically inserted by TCP. This is what we called a *byte stream service*. If the application on one end writes 10 bytes, followed by a write of 20 bytes, followed by a write of 50 bytes, the application at the other end of the connection cannot tell what size the individual writes were. The other end may read the 80 bytes in four reads of 20 bytes at a time. One end puts a stream of bytes into TCP and the same, identical stream of bytes appears at the other end.

Also, TCP does not interpret the contents of the bytes at all. TCP has no idea if the data bytes being exchanged are binary data, ASCII characters, EBCDIC characters, or whatever. The interpretation of this byte stream is up to the applications on each end of the connection.

This treatment of the byte stream by TCP is similar to the treatment of a file by the Unix operating system. The Unix kernel does no interpretation whatsoever of the bytes that an application reads or write—that is up to the applications. There is no distinction to the Unix kernel between a binary file or a file containing lines of text.

17.3 TCP Header

Recall that TCP data is encapsulated in an IP datagram, as shown in Figure 17.1.

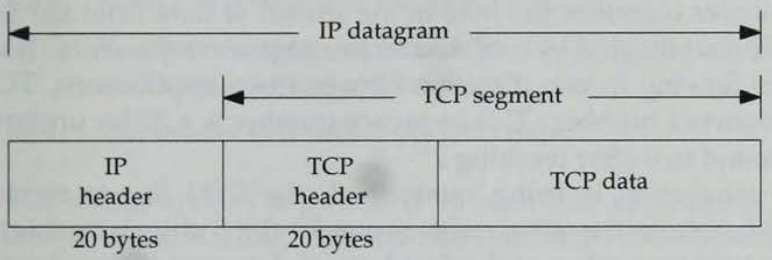


Figure 17.1 Encapsulation of TCP data in an IP datagram.

Figure 17.2 shows the format of the TCP header. Its normal size is 20 bytes, unless options are present.

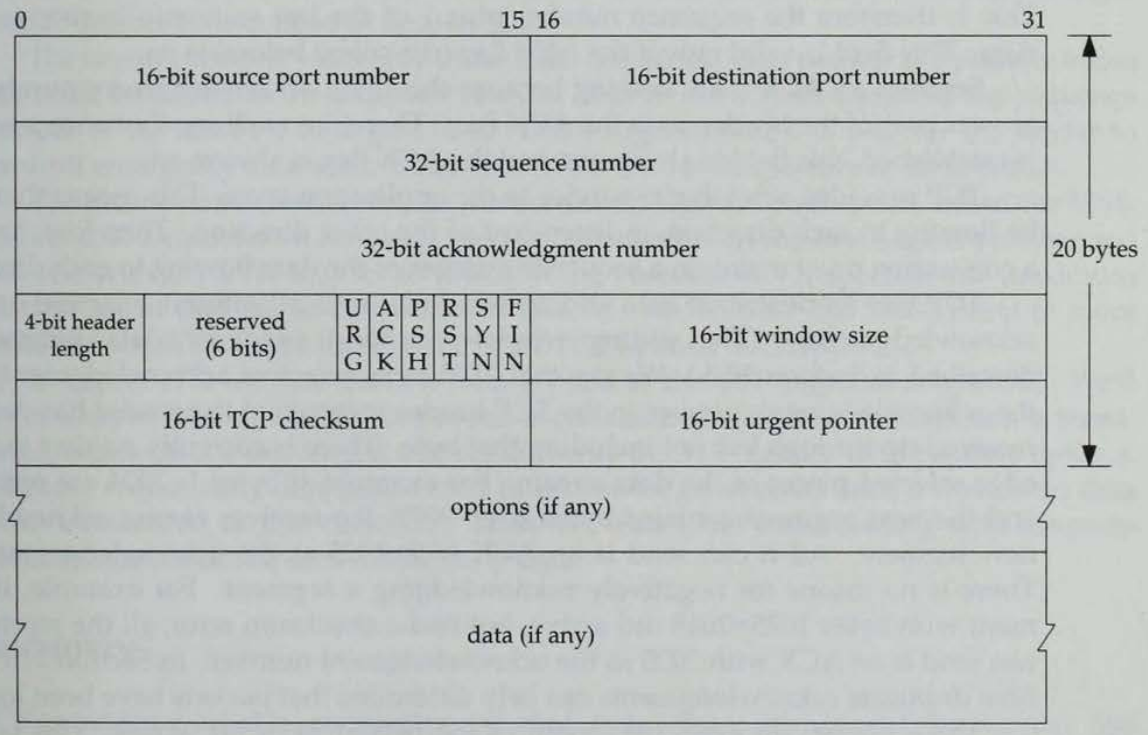


Figure 17.2 TCP header.

Each TCP segment contains the source and destination *port number* to identify the sending and receiving application. These two values, along with the source and destination IP addresses in the IP header, uniquely identify each *connection*.

The combination of an IP address and a port number is sometimes called a *socket*. This term appeared in the original TCP specification (RFC 793), and later it also became used as the name of the Berkeley-derived programming interface (Section 1.15). It is the *socket pair* (the 4-tuple consisting of the client IP address, client port number, server IP address, and server port number) that specifies the two end points that uniquely identifies each TCP connection in an internet.

The *sequence number* identifies the byte in the stream of data from the sending TCP to the receiving TCP that the first byte of data in this segment represents. If we consider the stream of bytes flowing in one direction between two applications, TCP numbers each byte with a sequence number. This sequence number is a 32-bit unsigned number that wraps back around to 0 after reaching $2^{32} - 1$.

When a new connection is being established, the SYN flag is turned on. The *sequence number field* contains the *initial sequence number* (ISN) chosen by this host for this connection. The sequence number of the first byte of data sent by this host will be the ISN plus one because the SYN flag consumes a sequence number. (We describe additional details on exactly how a connection is established and terminated in the next chapter where we'll see that the FIN flag consumes a sequence number also.)

Since every byte that is exchanged is numbered, the *acknowledgment number* contains the next sequence number that the sender of the acknowledgment expects to receive. This is therefore the sequence number plus 1 of the last successfully received byte of data. This field is valid only if the ACK flag (described below) is on.

Sending an ACK costs nothing because the 32-bit acknowledgment number field is always part of the header, as is the ACK flag. Therefore we'll see that once a connection is established, this field is always set and the ACK flag is always on.

TCP provides a *full-duplex* service to the application layer. This means that data can be flowing in each direction, independent of the other direction. Therefore, each end of a connection must maintain a sequence number of the data flowing in each direction.

TCP can be described as a sliding-window protocol without selective or negative acknowledgments. (The sliding window protocol used for data transmission is described in Section 20.3.) We say that TCP lacks selective acknowledgments because the acknowledgment number in the TCP header means that the sender has successfully received up through but not including that byte. There is currently no way to acknowledge selected pieces of the data stream. For example, if bytes 1–1024 are received OK, and the next segment contains bytes 2049–3072, the receiver cannot acknowledge this new segment. All it can send is an ACK with 1025 as the acknowledgment number. There is no means for negatively acknowledging a segment. For example, if the segment with bytes 1025–2048 did arrive, but had a checksum error, all the receiving TCP can send is an ACK with 1025 as the acknowledgment number. In Section 21.7 we'll see how duplicate acknowledgments can help determine that packets have been lost.

The *header length* gives the length of the header in 32-bit words. This is required because the length of the options field is variable. With a 4-bit field, TCP is limited to a 60-byte header. Without options, however, the normal size is 20 bytes.

There are six flag bits in the TCP header. One or more of them can be turned on at the same time. We briefly mention their use here and discuss each flag in more detail in later chapters.

URG The *urgent pointer* is valid (Section 20.8).

ACK The *acknowledgment number* is valid.

PSH The receiver should pass this data to the application as soon as possible (Section 20.5).

RST Reset the connection (Section 18.7).

SYN Synchronize sequence numbers to initiate a connection. This flag and the next are described in Chapter 18.

FIN The sender is finished sending data.

TCP's flow control is provided by each end advertising a *window size*. This is the number of bytes, starting with the one specified by the acknowledgment number field, that the receiver is willing to accept. This is a 16-bit field, limiting the window to 65535 bytes. In Section 24.4 we'll look at the new window scale option that allows this value to be scaled, providing larger windows.

The *checksum* covers the TCP segment: the TCP header and the TCP data. This is a mandatory field that must be calculated and stored by the sender, and then verified by the receiver. The TCP checksum is calculated similar to the UDP checksum, using a pseudo-header as described in Section 11.3.

The *urgent pointer* is valid only if the URG flag is set. This pointer is a positive offset that must be added to the sequence number field of the segment to yield the sequence number of the last byte of urgent data. TCP's urgent mode is a way for the sender to transmit emergency data to the other end. We'll look at this feature in Section 20.8.

The most common *option* field is the maximum segment size option, called the *MSS*. Each end of a connection normally specifies this option on the first segment exchanged (the one with the SYN flag set to establish the connection). It specifies the maximum sized segment that the sender wants to receive. We describe the MSS option in more detail in Section 18.4, and some of the other TCP options in Chapter 24.

In Figure 17.2 we note that the data portion of the TCP segment is optional. We'll see in Chapter 18 that when a connection is established, and when a connection is terminated, segments are exchanged that contain only the TCP header with possible options. A header without any data is also used to acknowledge received data, if there is no data to be transmitted in that direction. There are also some cases dealing with timeouts when a segment can be sent without any data.

17.4 Summary

TCP provides a reliable, connection-oriented, byte stream, transport layer service. We looked briefly at all the fields in the TCP header and will examine them in detail in the following chapters.

TCP packetizes the user data into segments, sets a timeout any time it sends data, acknowledges data received by the other end, reorders out-of-order data, discards duplicate data, provides end-to-end flow control, and calculates and verifies a mandatory end-to-end checksum.

TCP is used by many of the popular applications, such as Telnet, Rlogin, FTP, and electronic mail (SMTP).

Exercises

17.1 We've covered the following packet formats, each of which has a checksum in its corresponding header: IP, ICMP, IGMP, UDP, and TCP. For each one, describe what portion of an IP datagram the checksum covers and whether the checksum is mandatory or optional.

17.2 Why do all the Internet protocols that we've discussed (IP, ICMP, IGMP, UDP, TCP) quietly discard a packet that arrives with a checksum error?

17.3 TCP provides a byte-stream service where record boundaries are not maintained between the sender and receiver. How can applications provide their own record markers?

17.4 Why are the source and destination port numbers at the beginning of the TCP header?

17.5 Why does the TCP header have a header length field while the UDP header (Figure 11.2, p. 144) does not?

18

TCP Connection Establishment and Termination

18.1 Introduction

TCP is a *connection-oriented* protocol. Before either end can send data to the other, a *connection* must be established between them. In this chapter we take a detailed look at how a TCP connection is established and later terminated.

This establishment of a connection between the two ends differs from a *connectionless* protocol such as UDP. We saw in Chapter 11 that with UDP one end just sends a datagram to the other end, without any preliminary handshaking.

18.2 Connection Establishment and Termination

To see what happens when a TCP connection is established and then terminated, we type the following command on the system `svr4`:

```
svr4 % telnet bsdi discard
Trying 192.82.148.3 ...
Connected to bsdi.
Escape character is '^]'.
^]                                  type Control, right bracket to talk to the Telnet client
telnet> quit                        terminate the connection
Connection closed.
```

The `telnet` command establishes a TCP connection with the host `bsdi` on the port corresponding to the discard service (Section 1.12). This is exactly the type of service we need to see what happens when a connection is established and terminated, without having the server initiate any data exchange.

tcpdump Output

Figure 18.1 shows the tcpdump output for the segments generated by this command.

```
1  0.0                    svr4.1037 > bsdi.discard: S 1415531521:1415531521(0)
                                                     win 4096 <mss 1024>
2  0.002402 (0.0024)      bsdi.discard > svr4.1037: S 1823083521:1823083521(0)
                                                     ack 1415531522 win 4096
                                                     <mss 1024>
3  0.007224 (0.0048)      svr4.1037 > bsdi.discard: . ack 1823083522 win 4096
4  4.155441 (4.1482)      svr4.1037 > bsdi.discard: F 1415531522:1415531522(0)
                                                     ack 1823083522 win 4096
5  4.156747 (0.0013)      bsdi.discard > svr4.1037: . ack 1415531523 win 4096
6  4.158144 (0.0014)      bsdi.discard > svr4.1037: F 1823083522:1823083522(0)
                                                     ack 1415531523 win 4096
7  4.180662 (0.0225)      svr4.1037 > bsdi.discard: . ack 1823083523 win 4096
```

Figure 18.1 tcpdump output for TCP connection establishment and termination.

These seven TCP segments contain TCP headers only. No data is exchanged.

For TCP segments, each output line begins with

source > *destination: flags*

where *flags* represents four of the six flag bits in the TCP header (Figure 17.2). Figure 18.2 shows the five different characters that can appear in the *flags* output.

flag	3-character abbreviation	Description
S	SYN	synchronize sequence numbers
F	FIN	sender is finished sending data
R	RST	reset connection
P	PSH	push data to receiving process as soon as possible
.		none of above four flags is on

Figure 18.2 *flag* characters output by tcpdump for flag bits in TCP header.

In this example we see the S, F, and period. We'll see the other two *flags* (R and P) later. The other two TCP header flag bits—ACK and URG—are printed specially by tcpdump.

It's possible for more than one of the four flag bits in Figure 18.2 to be on in a single segment, but we normally see only one on at a time.

> RFC 1025 [Postel 1987], the *TCP and IP Bake Off*, calls a segment with the maximum combination of allowable flag bits turned on at once (SYN, URG, PSH, FIN, and 1 byte of data) a Kamikaze packet. It's also known as a nastygram, Christmas tree packet, and lamp test segment.

In line 1, the field `1415531521:1415531521(0)` means the sequence number of the packet was 1415531521 and the number of data bytes in the segment was 0. `tcpdump` displays this by printing the starting sequence number, a colon, the implied ending sequence number, and the number of data bytes in parentheses. The advantage of displaying both the sequence number and the implied ending sequence number is to see what the implied ending sequence number is, when the number of bytes is greater than 0. This field is output only if (1) the segment contains one or more bytes of data or (2) the SYN, FIN, or RST flag was on. Lines 1, 2, 4, and 6 in Figure 18.1 display this field because of the flag bits—we never exchange any data in this example.

In line 2 the field `ack 1415531522` shows the acknowledgment number. This is printed only if the ACK flag in the header is on.

The field `win 4096` in every line of output shows the window size being advertised by the sender. In these examples, where we are not exchanging any data, the window size never changes from its default of 4096. (We examine TCP's window size in Section 20.4.)

The final field that is output in Figure 18.1, `<mss 1024>` shows the *maximum segment size* (MSS) option specified by the sender. The sender does not want to receive TCP segments larger than this value. This is normally to avoid fragmentation (Section 11.5). We discuss the maximum segment size in Section 18.4, and show the format of the various TCP options in Section 18.10.

Time Line

Figure 18.3 shows the time line for this sequence of packets. (We described some general features of these time lines when we showed the first one in Figure 6.11, p. 80.) This figure shows which end is sending packets. We also expand some of the `tcpdump` output (e.g., printing SYN instead of `S`). In this time line we have also removed the window size values, since they add nothing to the discussion.

Connection Establishment Protocol

Now let's return to the details of the TCP protocol that are shown in Figure 18.3. To establish a TCP connection:

1. The requesting end (normally called the *client*) sends a SYN segment specifying the port number of the *server* that the client wants to connect to, and the client's *initial sequence number* (ISN, 1415531521 in this example). This is segment 1.
2. The server responds with its own SYN segment containing the server's initial sequence number (segment 2). The server also acknowledges the client's SYN by ACKing the client's ISN plus one. A SYN consumes one sequence number.
3. The client must acknowledge this SYN from the server by ACKing the server's ISN plus one (segment 3).

These three segments complete the connection establishment. This is often called the *three-way handshake*.

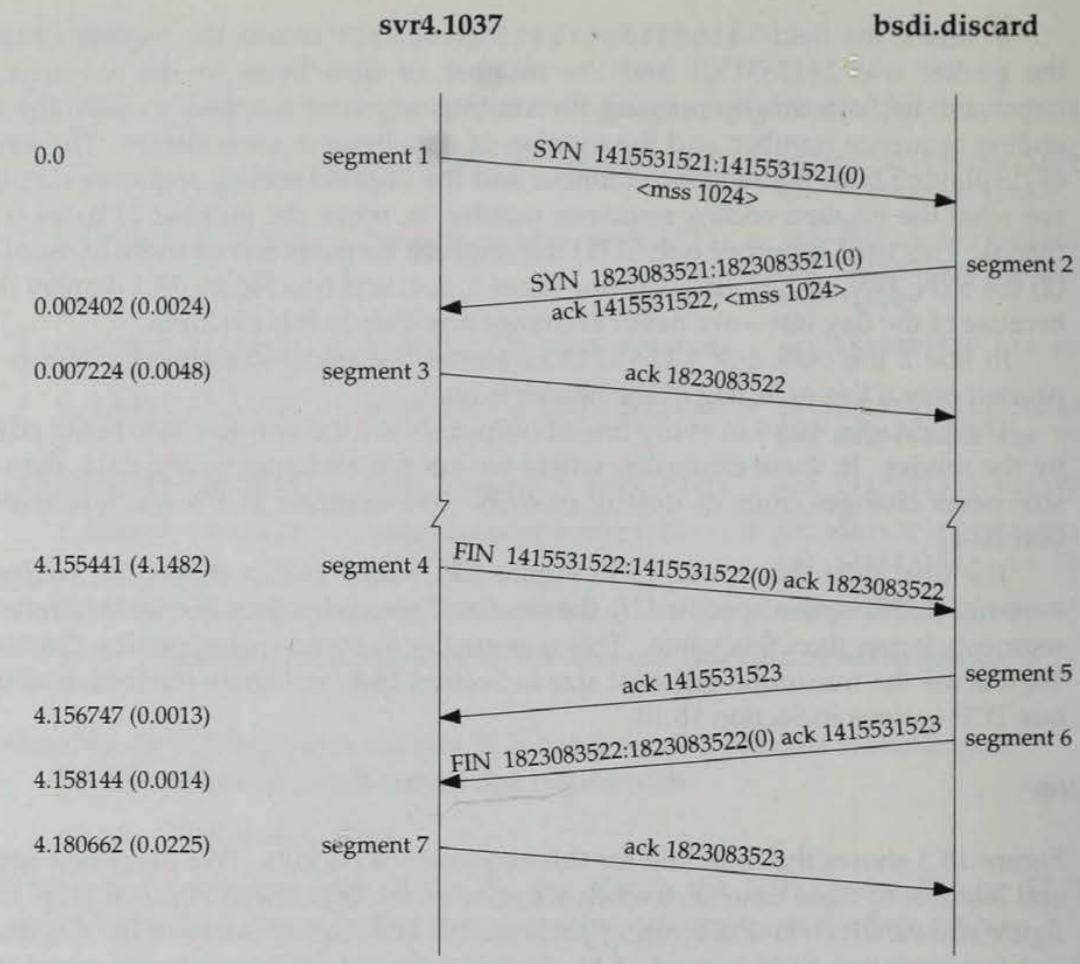


Figure 18.3 Time line of connection establishment and connection termination.

The side that sends the first SYN is said to perform an *active open*. The other side, which receives this SYN and sends the next SYN, performs a *passive open*. (In Section 18.8 we describe a simultaneous open where both sides can do an active open.)

When each end sends its SYN to establish the connection, it chooses an initial sequence number for that connection. The ISN should change over time, so that each connection has a different ISN. RFC 793 [Postel 1981c] specifies that the ISN should be viewed as a 32-bit counter that increments by one every 4 microseconds. The purpose in these sequence numbers is to prevent packets that get delayed in the network from being delivered later and then misinterpreted as part of an existing connection.

> How are the sequence numbers chosen? In 4.4BSD (and most Berkeley-derived implementations) when the system is initialized the initial send sequence number is initialized to 1. This practice violates the Host Requirements RFC. (A comment in the code acknowledges that this is wrong.) This variable is then incremented by 64,000 every half-second, and will cycle back to 0 about every 9.5 hours. (This corresponds to a counter that is incremented every 8

microseconds, not every 4 microseconds.) Additionally, each time a connection is established, this variable is incremented by 64,000.

The 4.1-second gap between segments 3 and 4 is the time between establishing the connection and typing the `quit` command to `telnet` to terminate the connection.

Connection Termination Protocol

While it takes three segments to establish a connection, it takes four to terminate a connection. This is caused by TCP's *half-close*. Since a TCP connection is full-duplex (that is, data can be flowing in each direction independently of the other direction), each direction must be shut down independently. The rule is that either end can send a FIN when it is done sending data. When a TCP receives a FIN, it must notify the application that the other end has terminated that direction of data flow. The sending of a FIN is normally the result of the application issuing a close.

The receipt of a FIN only means there will be no more data flowing in that direction. A TCP can still send data after receiving a FIN. While it's possible for an application to take advantage of this half-close, in practice few TCP applications use it. The normal scenario is what we show in Figure 18.3. We describe the half-close in more detail in Section 18.5.

We say that the end that first issues the close (e.g., sends the first FIN) performs the *active close* and the other end (that receives this FIN) performs the *passive close*. Normally one end does the active close and the other does the passive close, but we'll see in Section 18.9 how both ends can do an active close.

Segment 4 in Figure 18.3 initiates the termination of the connection and is sent when the Telnet client closes its connection. This happens when we type `quit`. This causes the client TCP to send a FIN, closing the flow of data from the client to the server.

When the server receives the FIN it sends back an ACK of the received sequence number plus one (segment 5). A FIN consumes a sequence number, just like a SYN. At this point the server's TCP also delivers an end-of-file to the application (the discard server). The server then closes its connection, causing its TCP to send a FIN (segment 6), which the client TCP must ACK by incrementing the received sequence number by one (segment 7).

Figure 18.4 shows the typical sequence of segments that we've described for the termination of a connection. We omit the sequence numbers. In this figure sending the FINs is caused by the applications closing their end of the connection, whereas the ACKs of these FINs are automatically generated by the TCP software.

Connections are normally initiated by the client, with the first SYN going from the client to the server. Either end can actively close the connection (i.e., send the first FIN). Often, however, it is the client that determines when the connection should be terminated, since client processes are often driven by an interactive user, who enters something like "quit" to terminate. In Figure 18.4 we can switch the labels at the top, calling the left side the server and the right side the client, and everything still works fine as shown. (The first example in Section 14.4, for example, shows the daytime server closing the connection.)

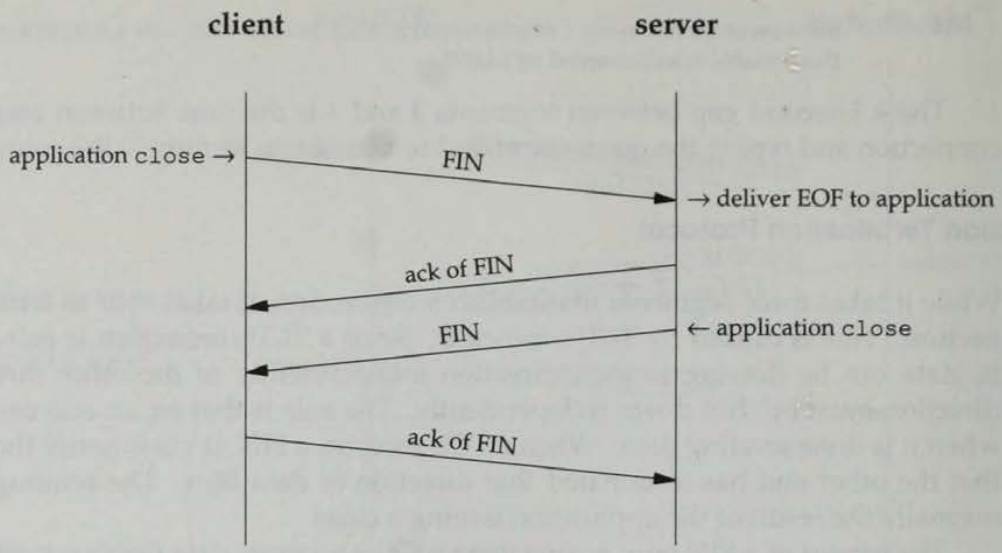


Figure 18.4 Normal exchange of segments during connection termination.

Normal `tcpdump` Output

Having to sort through all the huge sequence numbers is cumbersome, so the default `tcpdump` output shows the complete sequence numbers only on the SYN segments, and shows all following sequence numbers as relative offsets from the original sequence numbers. (To generate the output for Figure 18.1 we had to specify the `-S` option.) The normal `tcpdump` output corresponding to Figure 18.1 is shown in Figure 18.5.

```
1  0.0                  svr4.1037 > bsdi.discard: S 1415531521:1415531521(0)
                                                    win 4096 <mss 1024>
2  0.002402 (0.0024)    bsdi.discard > svr4.1037: S 1823083521:1823083521(0)
                                                    ack 1415531522
                                                    win 4096 <mss 1024>
3  0.007224 (0.0048)    svr4.1037 > bsdi.discard: . ack 1 win 4096
4  4.155441 (4.1482)    svr4.1037 > bsdi.discard: F 1:1(0) ack 1 win 4096
5  4.156747 (0.0013)    bsdi.discard > svr4.1037: . ack 2 win 4096
6  4.158144 (0.0014)    bsdi.discard > svr4.1037: F 1:1(0) ack 2 win 4096
7  4.180662 (0.0225)    svr4.1037 > bsdi.discard: . ack 2 win 4096
```

Figure 18.5 Normal `tcpdump` output for connection establishment and termination.

Unless we need to show the complete sequence numbers, we'll use this form of output in all following examples.

18.3 Timeout of Connection Establishment

There are several instances when the connection cannot be established. In one example the server host is down. To simulate this scenario we issue our `telnet` command after disconnecting the Ethernet cable from the server's host. Figure 18.6 shows the `tcpdump` output.

```
1   0.0                  bsdi.1024 > svr4.discard: S 291008001:291008001(0)
                                                    win 4096 <mss 1024>
                                                    [tos 0x10]
2   5.814797 ( 5.8148)   bsdi.1024 > svr4.discard: S 291008001:291008001(0)
                                                    win 4096 <mss 1024>
                                                    [tos 0x10]
3  29.815436 (24.0006)   bsdi.1024 > svr4.discard: S 291008001:291008001(0)
                                                    win 4096 <mss 1024>
                                                    [tos 0x10]
```

Figure 18.6 `tcpdump` output for connection establishment that times out.

The interesting point in this output is how frequently the client's TCP sends a SYN to try to establish the connection. The second segment is sent 5.8 seconds after the first, and the third is sent 24 seconds after the second.

> As a side note, this example was run about 38 minutes after the client was rebooted. This corresponds with the initial sequence number of 291,008,001 (approximately $38 \times 60 \times 64000 \times 2$). Recall earlier in this chapter we said that typical Berkeley-derived systems initialize the initial sequence number to 1 and then increment it by 64,000 every half-second.
>
> Also, this is the first TCP connection since the system was bootstrapped, which is why the client's port number is 1024.

What isn't shown in Figure 18.6 is how long the client's TCP keeps retransmitting before giving up. To see this we have to time the `telnet` command:

```
bsdi % date ; telnet svr4 discard ; date
Thu Sep 24 16:24:11 MST 1992
Trying 192.82.148.2...
telnet: Unable to connect to remote host: Connection timed out
Thu Sep 24 16:25:27 MST 1992
```

The time difference is 76 seconds. Most Berkeley-derived systems set a time limit of 75 seconds on the establishment of a new connection. We'll see in Section 21.4 that the third packet sent by the client would have timed out around 16:25:29, 48 seconds after it was sent, had the client not given up after 75 seconds.

First Timeout Period

One puzzling item in Figure 18.6 is that the first timeout period, 5.8 seconds, is close to 6 seconds, but not exact, while the second period is almost exactly 24 seconds. Ten more

of these tests were run and the first timeout period took on various values between 5.59 seconds and 5.93 seconds. The second timeout period, however, was always 24.00 (to two decimal places).

What's happening here is that BSD implementations of TCP run a timer that goes off every 500 ms. This 500-ms timer is used for various TCP timeouts, all of which we cover in later chapters. When we type in the `telnet` command, an initial 6-second timer is established (12 clock ticks), but it may expire anywhere between 5.5 and 6 seconds in the future. Figure 18.7 shows what's happening.

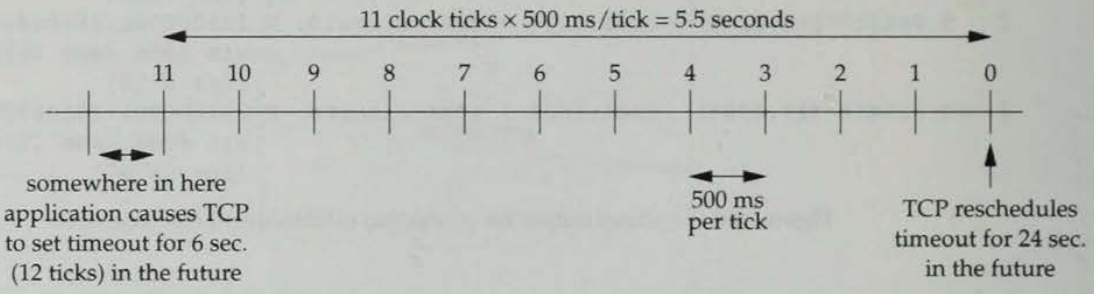


Figure 18.7 TCP 500-ms timer.

Although the timer is initialized to 12 ticks, the first decrement of the timer can occur between 0 and 500 ms after it is set. From that point on the timer is decremented about every 500 ms, but the first period can be variable. (We use the qualifier "about" because the time when TCP gets control every 500 ms can be preempted by other interrupts being handled by the kernel.)

When that 6-second timer expires at the tick labeled 0 in Figure 18.7, the timer is reset for 24 seconds (48 ticks) in the future. This next timer will be close to 24 seconds, since it was set at a time when the TCP's 500-ms timer handler was called by the kernel.

Type-of-Service Field

In Figure 18.6, the notation `[tos 0x10]` appears. This is the type-of-service (TOS) field in the IP datagram (Figure 3.2). The BSD/386 Telnet client sets the field for minimum delay.

18.4 Maximum Segment Size

The maximum segment size (MSS) is the largest "chunk" of data that TCP will send to the other end. When a connection is established, each end can announce its MSS. The values we've seen have all been 1024. The resulting IP datagram is normally 40 bytes larger: 20 bytes for the TCP header and 20 bytes for the IP header.

Some texts refer to this as a "negotiated" option. It is not negotiated in any way. When a connection is established, each end has the option of announcing the MSS it

expects to receive. (An MSS option can only appear in a SYN segment.) If one end does not receive an MSS option from the other end, a default of 536 bytes is assumed. (This default allows for a 20-byte IP header and a 20-byte TCP header to fit into a 576-byte IP datagram.)

In general, the larger the MSS the better, until fragmentation occurs. (This may not always be true. See Figures 24.3 and 24.4 for a counterexample.) A larger segment size allows more data to be sent in each segment, amortizing the cost of the IP and TCP headers. When TCP sends a SYN segment, either because a local application wants to initiate a connection, or when a connection request is received from another host, it can send an MSS value up to the outgoing interface's MTU, minus the size of the fixed TCP and IP headers. For an Ethernet this implies an MSS of up to 1460 bytes. Using IEEE 802.3 encapsulation (Section 2.2), the MSS could go up to 1452 bytes.

The values of 1024 that we've seen in this chapter, for connections involving BSD/386 and SVR4, are because many BSD implementations require the MSS to be a multiple of 512. Other systems, such as SunOS 4.1.3, Solaris 2.2, and AIX 3.2.2, all announce an MSS of 1460 when both ends are on a local Ethernet. Measurements in [Mogul 1993] show how an MSS of 1460 provides better performance on an Ethernet than an MSS of 1024.

If the destination IP address is "nonlocal," the MSS normally defaults to 536. While it's easy to say that a destination whose IP address has the same network ID and the same subnet ID as ours is local, and a destination whose IP address has a totally different network ID from ours is nonlocal, a destination with the same network ID but a different subnet ID could be either local or nonlocal. Most implementations provide a configuration option (Appendix E and Figure E.1) that lets the system administrator specify whether different subnets are local or nonlocal. The setting of this option determines whether the announced MSS is as large as possible (up to the outgoing interface's MTU) or the default of 536.

The MSS lets a host limit the size of datagrams that the other end sends it. When combined with the fact that a host can also limit the size of the datagrams that it sends, this lets a host avoid fragmentation when the host is connected to a network with a small MTU.

Consider our host `slip`, which has a SLIP link with an MTU of 296 to the router `bsdi`. Figure 18.8 shows these systems and the host `sun`.

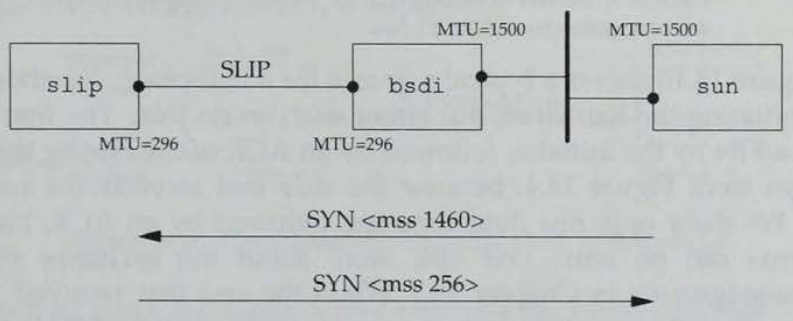


Figure 18.8 TCP connection from `sun` to `slip` showing MSS values.

We initiate a TCP connection from sun to slip and watch the segments using tcpdump. Figure 18.9 shows only the connection establishment (with the window size advertisements removed).

```
1  0.0                    sun.1093 > slip.discard: S 517312000:517312000(0)
                                                    <mss 1460>
2  0.10 (0.00)            slip.discard > sun.1093: S 509556225:509556225(0)
                                                    ack 517312001 <mss 256>
3  0.10 (0.00)            sun.1093 > slip.discard: . ack 1
```

Figure 18.9 tcpdump output for connection establishment from sun to slip.

The important fact here is that sun cannot send a segment with more than 256 bytes of data, since it received an MSS option of 256 (line 2). Furthermore, since slip knows that the outgoing interface's MTU is 296, even though sun announced an MSS of 1460, it will never send more than 256 bytes of data, to avoid fragmentation. It's OK for a system to send *less* than the MSS announced by the other end.

This avoidance of fragmentation works only if either host is directly connected to a network with an MTU of less than 576. If both hosts are connected to Ethernets, and both announce an MSS of 536, but an intermediate network has an MTU of 296, fragmentation will occur. The only way around this is to use the path MTU discovery mechanism (Section 24.2).

18.5 TCP Half-Close

TCP provides the ability for one end of a connection to terminate its output, while still receiving data from the other end. This is called a *half-close*. Few applications take advantage of this capability, as we mentioned earlier.

To use this feature the programming interface must provide a way for the application to say "I am done sending data, so send an end-of-file (FIN) to the other end, but I still want to receive data from the other end, until it sends me an end-of-file (FIN)."

> The sockets API supports the half-close, if the application calls shutdown with a second argument of 1, instead of calling close. Most applications, however, terminate both directions of the connection by calling close.

Figure 18.10 shows a typical scenario for a half-close. We show the client on the left side initiating the half-close, but either end can do this. The first two segments are the same: a FIN by the initiator, followed by an ACK of the FIN by the recipient. But it then changes from Figure 18.4, because the side that receives the half-close can still send data. We show only one data segment, followed by an ACK, but any number of data segments can be sent. (We talk more about the exchange of data segments and acknowledgments in Chapter 19.) When the end that received the half-close is done sending data, it closes its end of the connection, causing a FIN to be sent, and this delivers an end-of-file to the application that initiated the half-close. When this second FIN is acknowledged, the connection is completely closed.

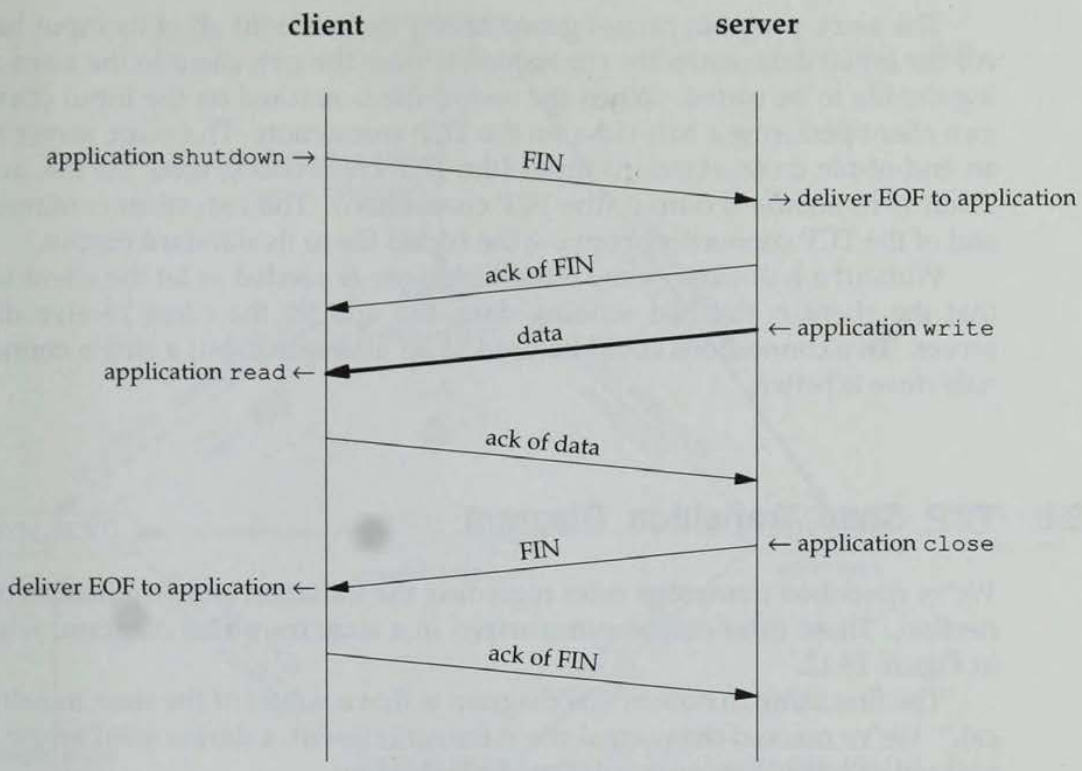


Figure 18.10 Example of TCP's half-close.

Why is there a half-close? One example is the Unix `rsh`(1) command, which executes a command on another system. The command

```
sun % rsh bsdi sort < datafile
```

executes the `sort` command on the host `bsdi` with standard input for the `rsh` command being read from the file named `datafile`. A TCP connection is created by `rsh` between itself and the program being executed on the other host. The operation of `rsh` is then simple: it copies standard input (`datafile`) to the connection, and copies from the connection to standard output (our terminal). Figure 18.11 shows the setup. (Remember that a TCP connection is full-duplex.)

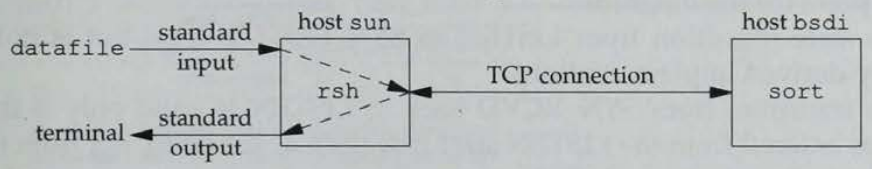


Figure 18.11 The command: `rsh bsdi sort < datafile`.

On the remote host `bsdi` the `rshd` server executes the `sort` program so that its standard input and standard output are both the TCP connection. Chapter 14 of [Stevens 1990] details the Unix process structure involved, but what concerns us here is the use of the TCP connection and the required use of TCP's half-close.

The `sort` program cannot generate any output until all of its input has been read. All the initial data across the connection is from the `rsh` client to the `sort` server, sending the file to be sorted. When the end-of-file is reached on the input (`datafile`), the `rsh` client performs a half-close on the TCP connection. The `sort` server then receives an end-of-file on its standard input (the TCP connection), sorts the file, and writes the result to its standard output (the TCP connection). The `rsh` client continues reading its end of the TCP connection, copying the sorted file to its standard output.

Without a half-close, some other technique is needed to let the client tell the server that the client is finished sending data, but still let the client receive data from the server. Two connections could be used as an alternative, but a single connection with a half-close is better.

18.6 TCP State Transition Diagram

We've described numerous rules regarding the initiation and termination of a TCP connection. These rules can be summarized in a state transition diagram, which we show in Figure 18.12.

The first thing to note in this diagram is that a subset of the state transitions is "typical." We've marked the normal client transitions with a darker solid arrow, and the normal server transitions with a darker dashed arrow.

Next, the two transitions leading to the ESTABLISHED state correspond to opening a connection, and the two transitions leading from the ESTABLISHED state are for the termination of a connection. The ESTABLISHED state is where data transfer can occur between the two ends in both directions. Later chapters describe what happens in this state.

We've collected the four boxes in the lower left of this diagram within a dashed box and labeled it "active close." Two other boxes (CLOSE_WAIT and LAST_ACK) are collected in a dashed box with the label "passive close."

The names of the 11 states (CLOSED, LISTEN, SYN_SENT, etc.) in this figure were purposely chosen to be identical to the states output by the `netstat` command. The `netstat` names, in turn, are almost identical to the names originally described in RFC 793. The state CLOSED is not really a state, but is the imaginary starting point and ending point for the diagram.

The state transition from LISTEN to SYN_SENT is legal but is not supported in Berkeley-derived implementations.

The transition from SYN_RCVD back to LISTEN is valid only if the SYN_RCVD state was entered from the LISTEN state (the normal scenario), not from the SYN_SENT state (a simultaneous open). This means if we perform a passive open (enter LISTEN), receive a SYN, send a SYN with an ACK (enter SYN_RCVD), and then receive a reset instead of an ACK, the end point returns to the LISTEN state and waits for another connection request to arrive.

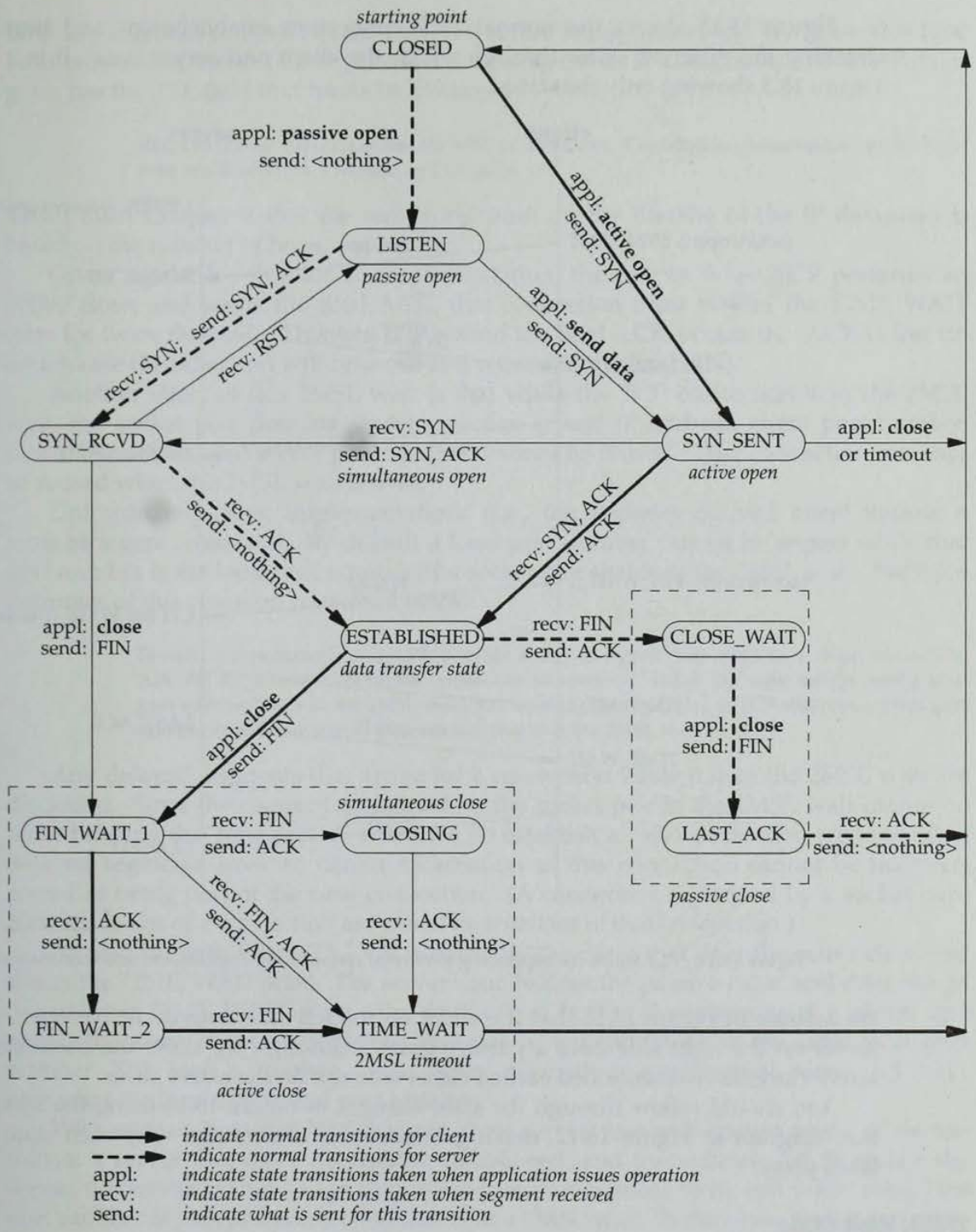


Figure 18.12 TCP state transition diagram.

Figure 18.13 shows the normal TCP connection establishment and termination, detailing the different states through which the client and server pass. It is a redo of Figure 18.3 showing only the states.

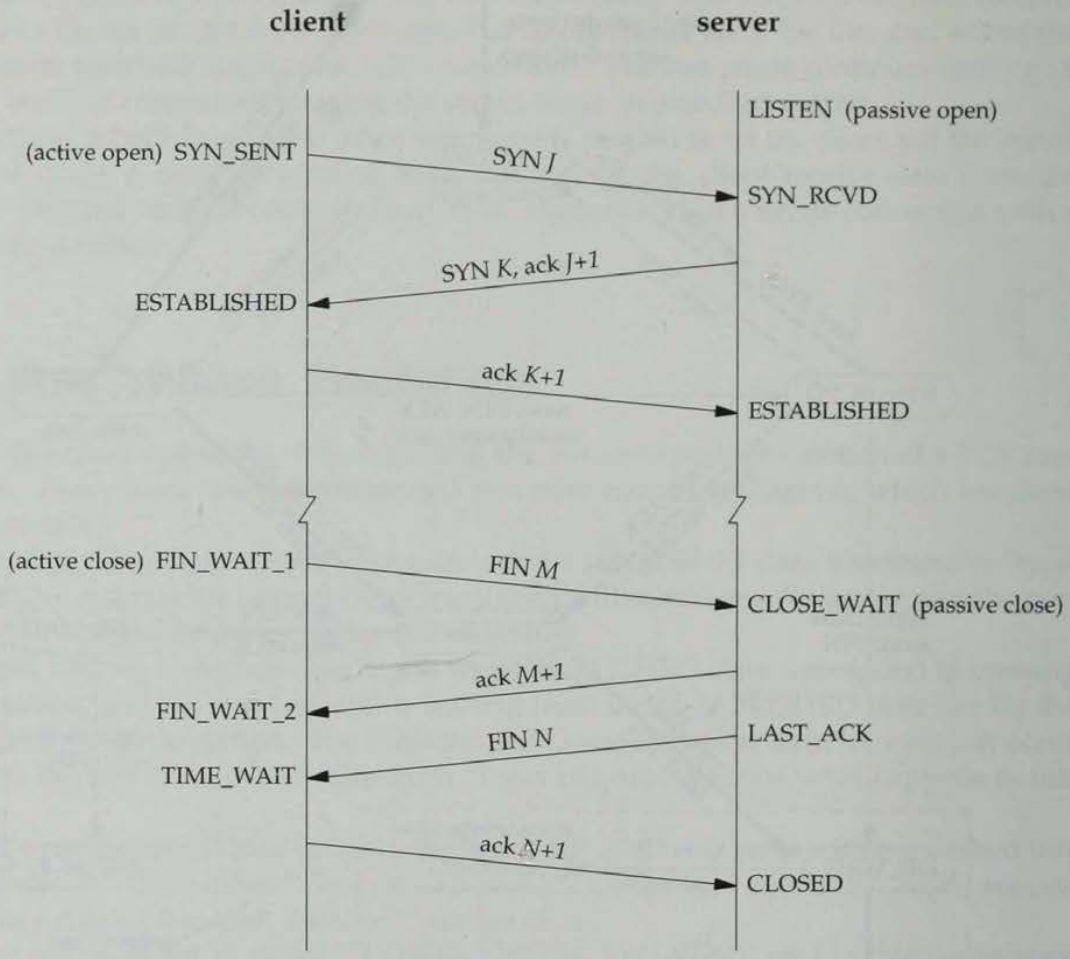


Figure 18.13 TCP states corresponding to normal connection establishment and termination.

We assume in Figure 18.13 that the client on the left side does an active open, and the server on the right side does a passive open. Although we show the client doing the active close, as we mentioned earlier, either side can do the active close.

You should follow through the state changes in Figure 18.13 using the state transition diagram in Figure 18.12, making certain you understand why each state change takes place.

2MSL Wait State

The TIME_WAIT state is also called the 2MSL wait state. Every implementation must choose a value for the *maximum segment lifetime* (MSL). It is the maximum amount of

time any segment can exist in the network before being discarded. We know this time limit is bounded, since TCP segments are transmitted as IP datagrams, and the IP datagram has the TTL field that limits its lifetime.

> RFC 793 [Postel 1981c] specifies the MSL as 2 minutes. Common implementation values, however, are 30 seconds, 1 minute, or 2 minutes.

Recall from Chapter 8 that the real-world limit on the lifetime of the IP datagram is based on the number of hops, not a timer.

Given the MSL value for an implementation, the rule is: when TCP performs an active close, and sends the final ACK, that connection must stay in the TIME_WAIT state for twice the MSL. This lets TCP resend the final ACK in case this ACK is lost (in which case the other end will time out and retransmit its final FIN).

Another effect of this 2MSL wait is that while the TCP connection is in the 2MSL wait, the socket pair defining that connection (client IP address, client port number, server IP address, and server port number) cannot be reused. That connection can only be reused when the 2MSL wait is over.

Unfortunately most implementations (i.e., the Berkeley-derived ones) impose a more stringent constraint. By default a local port number cannot be reused while that port number is the local port number of a socket pair that is in the 2MSL wait. We'll see examples of this common constraint below.

> Some implementations and APIs provide a way to bypass this restriction. With the sockets API, the `SO_REUSEADDR` socket option can be specified. It lets the caller assign itself a local port number that's in the 2MSL wait, but we'll see that the rules of TCP still prevent this port number from being part of a connection that is in the 2MSL wait.

Any delayed segments that arrive for a connection while it is in the 2MSL wait are discarded. Since the connection defined by the socket pair in the 2MSL wait cannot be reused during this time period, when we do establish a valid connection we know that delayed segments from an earlier incarnation of this connection cannot be misinterpreted as being part of the new connection. (A connection is defined by a socket pair. New instances of a connection are called *incarnations* of that connection.)

As we said with Figure 18.13, it is normally the client that does the active close and enters the TIME_WAIT state. The server usually does the passive close, and does not go through the TIME_WAIT state. The implication is that if we terminate a client, and restart the same client immediately, that new client cannot reuse the same local port number. This isn't a problem, since clients normally use ephemeral ports, and don't care what the local ephemeral port number is.

With servers, however, this changes, since servers use well-known ports. If we terminate a server that has a connection established, and immediately try to restart the server, the server cannot assign its well-known port number to its end point, since that port number is part of a connection that is in a 2MSL wait. It may take from 1 to 4 minutes before the server can be restarted.

We can see this scenario using our `sock` program. We start the server, connect to it from a client, and then terminate the server:

```
sun % sock -v -s 6666                    start as server, listening on port 6666
                                         (execute client on bsdi that connects to this port)
connection on 140.252.13.33.6666 from 140.252.13.35.1081
^?                                       then type interrupt key to terminate server
sun % sock -s 6666                       and immediately try to restart server on same port
can't bind local address: Address already in use
sun % netstat                            let's check the state of the connection
Active Internet connections
Proto Recv-Q Send-Q  Local Address   Foreign Address     (state)
tcp        0      0  sun.6666        bsdi.1081          TIME_WAIT
                                         many more lines that are deleted
```

When we try to restart the server, the program outputs an error message indicating it cannot bind its well-known port number, because it's already in use (i.e., it's in a 2MSL wait).

We then immediately execute `netstat` to see the state of the connection, and verify that it is indeed in the TIME_WAIT state.

> If we continually try to restart the server, and measure the time until it succeeds, we can measure the 2MSL value. On SunOS 4.1.3, SVR4, BSD/386, and AIX 3.2.2, it takes 1 minute to restart the server, meaning the MSL is 30 seconds. Under Solaris 2.2 it takes 4 minutes to restart the server, implying an MSL of 2 minutes.

We can see the same error from a client, if the client tries to allocate a port that is part of a connection in the 2MSL wait (something clients normally don't do):

```
sun % sock -v bsdi echo                  start as client, connect to echo server
connected on 140.252.13.33.1162 to 140.252.13.35.7
hello there                              type this line
hello there                              and it's echoed by the server
^D                                       type end-of-file character to terminate client
sun % sock -b1162 bsdi echo
can't bind local address: Address already in use
```

The first time we execute the client we specify the `-v` option to see what the local port number is (1162). The second time we execute the client we specify the `-b` option, telling the client to assign itself 1162 as its local port number. As we expect, the client can't do this, since that port number is part of a connection that is in a 2MSL wait.

We need to reemphasize one effect of the 2MSL wait because we'll encounter it in Chapter 27 with FTP, the File Transfer Protocol. As we said earlier, it is a socket pair (that is, the 4-tuple consisting of a local IP address, local port, remote IP address and remote port) that remains in the 2MSL wait. Although many implementations allow a process to reuse a port number that is part of a connection that is in the 2MSL wait (normally with an option named `SO_REUSEADDR`), TCP *cannot* allow a new connection to be created with the same socket pair. We can see this with the following experiment:

```
sun % sock -v -s 6666                    start as server, listening on port 6666
                                         (execute client on bsdi that connects to this port)
```

```
connection on 140.252.13.33.6666 from 140.252.13.35.1098
^?                                  then type interrupt key to terminate server

sun % sock -b6666 bsdi 1098         try to start as client with local port 6666
can't bind local address: Address already in use

sun % sock -A -b6666 bsdi 1098      try again, this time with -A option
active open error: Address already in use
```

The first time we run our `sock` program, we run it as a server on port 6666 and connect to it from a client on the host `bsdi`. The client's ephemeral port number is 1098. We terminate the server so it does the active close. This causes the 4-tuple of 140.252.13.33 (local IP address), 6666 (local port number), 140.252.13.35 (foreign IP address), and 1098 (foreign port number) to enter the 2MSL wait on the server host.

The second time we run the program, we run it as a client and try to specify the local port number as 6666 and connect to host `bsdi` on port 1098. But the program gets an error when it tries to assign itself the local port number of 6666, because that port number is part of the 4-tuple that is in the 2MSL wait state.

To try and get around this error we run the program again, specifying the `-A` option, which enables the `SO_REUSEADDR` option that we mentioned. This lets the program assign itself the port number 6666, but we then get an error when it tries to issue the active open. Even though it can assign itself the port number 6666, it cannot create a connection to port 1098 on the host `bsdi`, because the socket pair defining that connection is in the 2MSL wait state.

What if we try to establish the connection from the other host? First we must restart the server on `sun` with the `-A` flag, since the local port it needs (6666) is part of a connection that is in the 2MSL wait:

```
sun % sock -A -s 6666               start as server, listening on port 6666
```

Then, before the 2MSL wait is over on `sun`, we start the client on `bsdi`:

```
bsdi % sock -b1098 sun 6666
connected on 140.252.13.35.1098 to 140.252.13.33.6666
```

Unfortunately it works! This is a violation of the TCP specification, but is supported by most Berkeley-derived implementations. These implementations allow a new connection request to arrive for a connection that is in the TIME_WAIT state, if the new sequence number is greater than the final sequence number from the previous incarnation of this connection. In this case the ISN for the new incarnation is set to the final sequence number from the previous incarnation plus 128,000. The appendix of RFC 1185 [Jacobson, Braden, and Zhang 1990] shows the pitfalls still possible with this technique.

This implementation feature lets a client and server continually reuse the same port number at each end for successive incarnations of the same connection, but only if the server does the active close. We'll see another example of this 2MSL wait condition in Figure 27.8, with FTP. See Exercise 18.5 also.

Quiet Time Concept

The 2MSL wait provides protection against delayed segments from an earlier incarnation of a connection from being interpreted as part of a new connection that uses the same local and foreign IP addresses and port numbers. But this works only if a host with connections in the 2MSL wait does not crash.

What if a host with ports in the 2MSL wait crashes, reboots within MSL seconds, and immediately establishes new connections using the same local and foreign IP addresses and port numbers corresponding to the local ports that were in the 2MSL wait before the crash? In this scenario, delayed segments from the connections that existed before the crash can be misinterpreted as belonging to the new connections created after the reboot. This can happen regardless of how the initial sequence number is chosen after the reboot.

To protect against this scenario, RFC 793 states that TCP should not create any connections for MSL seconds after rebooting. This is called the *quiet time*.

> Few implementations abide by this since most hosts take longer than MSL seconds to reboot after a crash.

FIN_WAIT_2 State

In the FIN_WAIT_2 state we have sent our FIN and the other end has acknowledged it. Unless we have done a half-close, we are waiting for the application on the other end to recognize that it has received an end-of-file notification and close its end of the connection, which sends us a FIN. Only when the process at the other end does this close will our end move from the FIN_WAIT_2 to the TIME_WAIT state.

This means our end of the connection can remain in this state forever. The other end is still in the CLOSE_WAIT state, and can remain there forever, until the application decides to issue its close.

> Many Berkeley-derived implementations prevent this infinite wait in the FIN_WAIT_2 state as follows. If the application that does the active close does a complete close, not a half-close indicating that it expects to receive data, then a timer is set. If the connection is idle for 10 minutes plus 75 seconds, TCP moves the connection into the CLOSED state. A comment in the code acknowledges that this implementation feature violates the protocol specification.

18.7 Reset Segments

We've mentioned a bit in the TCP header named RST for "reset." In general, a reset is sent by TCP whenever a segment arrives that doesn't appear correct for the referenced connection. (We use the term "referenced connection" to mean the connection specified by the destination IP address and port number, and the source IP address and port number. This is what RFC 793 calls a socket.)

Connection Request to Nonexistent Port

A common case for generating a reset is when a connection request arrives and no process is listening on the destination port. In the case of UDP, we saw in Section 6.5 that an ICMP port unreachable was generated when a datagram arrived for a destination port that was not in use. TCP uses a reset instead.

This example is trivial to generate—we use the Telnet client and specify a port number that's not in use on the destination:

```
bsdi % telnet svr4 20000              port 20000 should not be in use
Trying 140.252.13.34...
telnet: Unable to connect to remote host: Connection refused
```

This error message is output by the Telnet client immediately. Figure 18.14 shows the packet exchange corresponding to this command.

```
1  0.0                  bsdi.1087 > svr4.20000: S 297416193:297416193(0)
                                                 win 4096 <mss 1024>
                                                 [tos 0x10]

2  0.003771 (0.0038)    svr4.20000 > bsdi.1087: R 0:0(0) ack 297416194 win 0
```

Figure 18.14 Reset generated by attempt to open connection to nonexistent port.

The values we need to examine in this figure are the sequence number field and acknowledgment number field in the reset. Because the ACK bit was not on in the arriving segment, the sequence number of the reset is set to 0 and the acknowledgment number is set to the incoming ISN plus the number of data bytes in the segment. Although there is no real data in the arriving segment, the SYN bit logically occupies 1 byte of sequence number space; therefore, in this example the acknowledgment number in the reset is set to the ISN, plus the data length (0), plus one for the SYN bit.

Aborting a Connection

We saw in Section 18.2 that the normal way to terminate a connection is for one side to send a FIN. This is sometimes called an *orderly release* since the FIN is sent after all previously queued data has been sent, and there is normally no loss of data. But it's also possible to abort a connection by sending a reset instead of a FIN. This is sometimes called an *abortive release*.

Aborting a connection provides two features to the application: (1) any queued data is thrown away and the reset is sent immediately, and (2) the receiver of the RST can tell that the other end did an abort instead of a normal close. The API being used by the application must provide a way to generate the abort instead of a normal close.

We can watch this abort sequence happen using our `sock` program. The sockets API provides this capability by using the "linger on close" socket option (`SO_LINGER`). We specify the `-L` option with a linger time of 0. This causes the abort to be sent when the connection is closed, instead of the normal FIN. We'll connect to a server version of our `sock` program on `svr4` and type one line of input:

```
bsdi % sock -L0 svr4 8888          this is the client; server shown later
hello, world                       type one line of input that's sent to other end
^D                                 type end-of-file character to terminate client
```

Figure 18.15 shows the `tcpdump` output for this example. (We have deleted all the window advertisements in this figure, since they add nothing to the discussion.)

```
1  0.0                  bsdi.1099 > svr4.8888: S 671112193:671112193(0)
                                                 <mss 1024>
2  0.004975 (0.0050)    svr4.8888 > bsdi.1099: S 3224959489:3224959489(0)
                                                 ack 671112194 <mss 1024>
3  0.006656 (0.0017)    bsdi.1099 > svr4.8888: . ack 1

4  4.833073 (4.8264)    bsdi.1099 > svr4.8888: P 1:14(13) ack 1
5  5.026224 (0.1932)    svr4.8888 > bsdi.1099: . ack 14

6  9.527634 (4.5014)    bsdi.1099 > svr4.8888: R 14:14(0) ack 1
```

Figure 18.15 Aborting a connection with a reset (RST) instead of a FIN.

Lines 1–3 show the normal connection establishment. Line 4 sends the data line that we typed (12 characters plus the Unix newline character), and line 5 is the acknowledgment of the received data.

Line 6 corresponds to our typing the end-of-file character (Control-D) to terminate the client. Since we specified an abort instead of a normal close (the `-L0` command-line option), the TCP on `bsdi` sends an RST instead of the normal FIN. The RST segment contains a sequence number and acknowledgment number. Also notice that the RST segment elicits no response from the other end—it is not acknowledged at all. The receiver of the reset aborts the connection and advises the application that the connection was reset.

We get the following error on the server for this exchange:

```
svr4 % sock -s 8888                run as server, listen on port 8888
hello, world                       this is what the client sent over
read error: Connection reset by peer
```

This server reads from the network and copies whatever it receives to standard output. It normally ends by receiving an end-of-file notification from its TCP, but here we see that it receives an error when the RST arrives. The error is what we expect: the connection was reset by the peer.

Detecting Half-Open Connections

A TCP connection is said to be *half-open* if one end has closed or aborted the connection without the knowledge of the other end. This can happen any time one of the two hosts crashes. As long as there is no attempt to transfer data across a half-open connection, the end that's still up won't detect that the other end has crashed.

Another common cause of a half-open connection is when a client host is powered off, instead of terminating the client application and then shutting down the client host. This happens when PCs are being used to run Telnet clients, for example, and the users

power off the PC at the end of the day. If there was no data transfer going on when the PC was powered off, the server will never know that the client disappeared. When the user comes in the next morning, powers on the PC, and starts a new Telnet client, a new occurrence of the server is started on the server host. This can lead to many half-open TCP connections on the server host. (In Chapter 23 we'll see a way for one end of a TCP connection to discover that the other end has disappeared using TCP's keepalive option.)

We can easily create a half-open connection. We'll execute the Telnet client on `bsdi`, connecting to the discard server on `svr4`. We type one line of input, and watch it go across with `tcpdump`, and then disconnect the Ethernet cable on the server's host, and reboot the server host. This simulates the server host crashing. (We disconnect the Ethernet cable before rebooting the server to prevent it from sending a FIN out the open connections, which some TCPs do when they are shut down.) After the server has rebooted, we reconnect the cable, and try to send another line from the client to the server. Since the server's TCP has rebooted, and lost all memory of the connections that existed before it was rebooted, it knows nothing about the connection that the data segment references. The rule of TCP is that the receiver responds with a reset.

```
bsdi % telnet svr4 discard          start the client
Trying 140.252.13.34...
Connected to svr4.
Escape character is '^]'.
hi there                            this line is sent OK
                                    here is where we reboot the server host
another line                        and this one elicits a reset
Connection closed by foreign host.
```

Figure 18.16 shows the `tcpdump` output for this example. (We have removed from this output the window advertisements, the type-of-service information, and the MSS announcements, since they add nothing to the discussion.)

```
1    0.0                  bsdi.1102 > svr4.discard: S 1591752193:1591752193(0)
2    0.004811 (  0.0048)  svr4.discard > bsdi.1102: S 26368001:26368001(0)
                                                      ack 1591752194
3    0.006516 (  0.0017)  bsdi.1102 > svr4.discard: . ack 1

4    5.167679 (  5.1612)  bsdi.1102 > svr4.discard: P 1:11(10) ack 1
5    5.201662 (  0.0340)  svr4.discard > bsdi.1102: . ack 11

6  194.909929 (189.7083)  bsdi.1102 > svr4.discard: P 11:25(14) ack 1
7  194.914957 (  0.0050)  arp who-has bsdi tell svr4
8  194.915678 (  0.0007)  arp reply bsdi is-at 0:0:c0:6f:2d:40
9  194.918225 (  0.0025)  svr4.discard > bsdi.1102: R 26368002:26368002(0)
```

Figure 18.16 Reset in response to data segment on a half-open connection.

Lines 1–3 are the normal connection establishment. Line 4 sends the line "hi there" to the discard server, and line 5 is the acknowledgment.

At this point we disconnect the Ethernet cable from `svr4`, reboot it, and reconnect the cable. This takes almost 190 seconds. We then type the next line of input to the client ("another line") and when we type the return key the line is sent to the server

(line 6 in Figure 18.16). This elicits a response from the server, but note that since the server was rebooted, its ARP cache is empty, so an ARP request and reply are required (lines 7 and 8). Then the reset is sent in line 9. The client receives the reset and outputs that the connection was terminated by the foreign host. (The final message output by the Telnet client is not as informative as it could be.)

18.8 Simultaneous Open

It is possible, although improbable, for two applications to both perform an active open to each other at the same time. Each end must transmit a SYN, and the SYNs must pass each other on the network. It also requires each end to have a local port number that is well known to the other end. This is called a *simultaneous open*.

For example, one application on host A could have a local port of 7777 and perform an active open to port 8888 on host B. The application on host B would have a local port of 8888 and perform an active open to port 7777 on host A.

This is *not* the same as connecting a Telnet client on host A to the Telnet server on host B, at the same time that a Telnet client on host B is connecting to the Telnet server on host A. In this Telnet scenario, both Telnet servers perform passive opens, not active opens, and the Telnet clients assign themselves an ephemeral port number, not a port number that is well known to the other Telnet server.

TCP was purposely designed to handle simultaneous opens and the rule is that only one connection results from this, not two connections. (Other protocol suites, notably the OSI transport layer, create two connections in this scenario, not one.)

When a simultaneous open occurs the state transitions differ from those shown in Figure 18.13. Both ends send a SYN at about the same time, entering the SYN_SENT state. When each end receives the SYN, the state changes to SYN_RCVD (Figure 18.12), and each end resends the SYN and acknowledges the received SYN. When each end receives the SYN plus the ACK, the state changes to ESTABLISHED. These state changes are summarized in Figure 18.17.

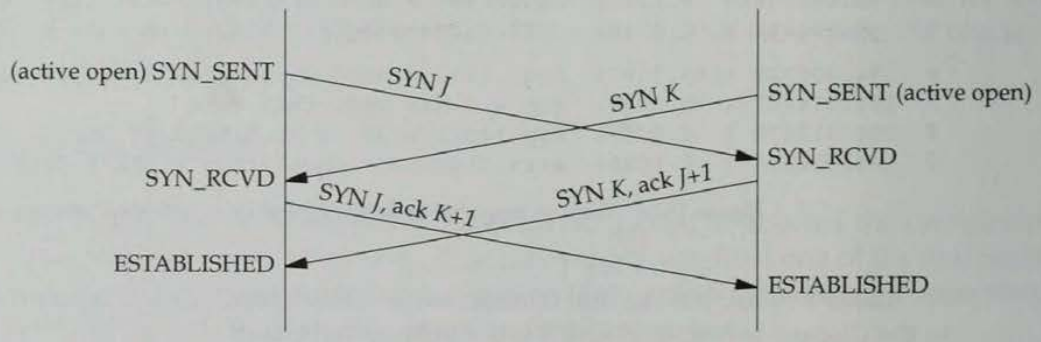


Figure 18.17 Segments exchanged during simultaneous open.

A simultaneous open requires the exchange of four segments, one more than the normal three-way handshake. Also notice that we don't call either end a client or a server, because both ends act as client and server.

An Example

It is possible, though hard, to generate a simultaneous open. The two ends must be started at about the same time, so that the SYNs cross each other. Having a long round-trip time between the two ends helps, to let the SYNs cross. To do this we'll execute one end on our host `bsdi`, and the other end on the host `vangogh.cs.berkeley.edu`. Since there is a dialup SLIP link between them, the round-trip time should be long enough (a few hundred milliseconds) to let the SYNs cross.

One end (`bsdi`) assigns itself a local port of 8888 (the `-b` command-line option) and performs an active open to port 7777 on the other host:

```
bsdi % sock -v -b8888 vangogh.cs.berkeley.edu 7777
connected on 140.252.13.35.8888 to 128.32.130.2.7777
TCP_MAXSEG = 512
hello, world                          we type this line
and hi there                          this line was typed on other end
connection closed by peer             this is output when FIN received
```

The other end is started at about the same time, assigns itself a local port of 7777, and performs an active open to port 8888:

```
vangogh % sock -v -b7777 bsdi.tuc.noao.edu 8888
connected on 128.32.130.2.7777 to 140.252.13.35.8888
TCP_MAXSEG = 512
hello, world                          this is typed on the other end
and hi there                          we type this line
^D                                    and then type our EOF character
```

We specify the `-v` flag to our `sock` program to verify the IP address and port numbers on each end of the connection. This flag also prints the MSS used by each end of the connection. We also type in one line on each end, which is sent to the other end and printed, to verify that both ends are indeed talking to each other.

Figure 18.18 shows the exchange of segments across the connection. (We have deleted some new TCP options that appear in the original SYN from `vangogh`, a 4.4BSD system. We describe these newer options in Section 18.10.) Notice the two SYNs (lines 1 and 2) followed by the two SYNs with ACKs (lines 3 and 4). These perform the simultaneous open.

Line 5 shows the input line "hello, world" going from `bsdi` to `vangogh`, with the acknowledgment in line 6. Lines 7 and 8 correspond to the line "and hi there" going in the other direction. Lines 9–12 show the normal connection termination.

> Many Berkeley-derived implementations do not support the simultaneous open correctly. On these systems, if you can get the SYNs to cross, you end up with an infinite exchange of segments, each with a SYN and an ACK, in each direction. The transition from the SYN_SENT state to the SYN_RCVD state in Figure 18.12 is not always tested in many implementations.

```
 1   0.0                 bsdi.8888 > vangogh.7777: S 91904001:91904001(0)
                                                   win 4096 <mss 512>
 2   0.213782 (0.2138)   vangogh.7777 > bsdi.8888: S 1058199041:1058199041(0)
                                                   win 8192 <mss 512>
 3   0.215399 (0.0016)   bsdi.8888 > vangogh.7777: S 91904001:91904001(0)
                                                   ack 1058199042 win 4096
                                                   <mss 512>
 4   0.340405 (0.1250)   vangogh.7777 > bsdi.8888: S 1058199041:1058199041(0)
                                                   ack 91904002 win 8192
                                                   <mss 512>
 5   5.633142 (5.2927)   bsdi.8888 > vangogh.7777: P 1:14(13) ack 1 win 4096
 6   6.100366 (0.4672)   vangogh.7777 > bsdi.8888: . ack 14 win 8192
 7   9.640214 (3.5398)   vangogh.7777 > bsdi.8888: P 1:14(13) ack 14 win 8192
 8   9.796417 (0.1562)   bsdi.8888 > vangogh.7777: . ack 14 win 4096
 9  13.060395 (3.2640)   vangogh.7777 > bsdi.8888: F 14:14(0) ack 14 win 8192
10  13.061828 (0.0014)   bsdi.8888 > vangogh.7777: . ack 15 win 4096
11  13.079769 (0.0179)   bsdi.8888 > vangogh.7777: F 14:14(0) ack 15 win 4096
12  13.299940 (0.2202)   vangogh.7777 > bsdi.8888: . ack 15 win 8192
```

Figure 18.18 Exchange of segments during simultaneous open.

18.9 Simultaneous Close

We said earlier that one side (often, but not always, the client) performs the active close, causing the first FIN to be sent. It's also possible for both sides to perform an active close, and the TCP protocol allows for this *simultaneous close*.

In terms of Figure 18.12, both ends go from ESTABLISHED to FIN_WAIT_1 when the application issues the close. This causes both FINs to be sent, and they probably pass each other somewhere in the network. When the FIN is received, each end transitions from FIN_WAIT_1 to the CLOSING state, and each state sends its final ACK. When each end receives the final ACK, the state changes to TIME_WAIT. Figure 18.19 summarizes these state changes.

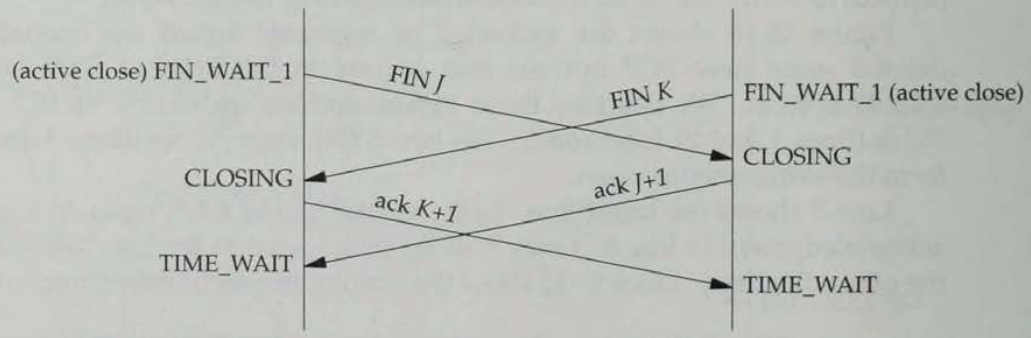


Figure 18.19 Segments exchanged during simultaneous close.

With a simultaneous close the same number of segments are exchanged as in the normal close.

18.10 TCP Options

The TCP header can contain options (Figure 17.2). The only options defined in the original TCP specification are the end of option list, no operation, and the maximum segment size option. We have seen the MSS option in almost every SYN segment in our examples.

Newer RFCs, specifically RFC 1323 [Jacobson, Braden, and Borman 1992], define additional TCP options, most of which are found only in the latest implementations. (We describe these new options in Chapter 24.) Figure 18.20 shows the format of the current TCP options—those from RFC 793 and RFC 1323.

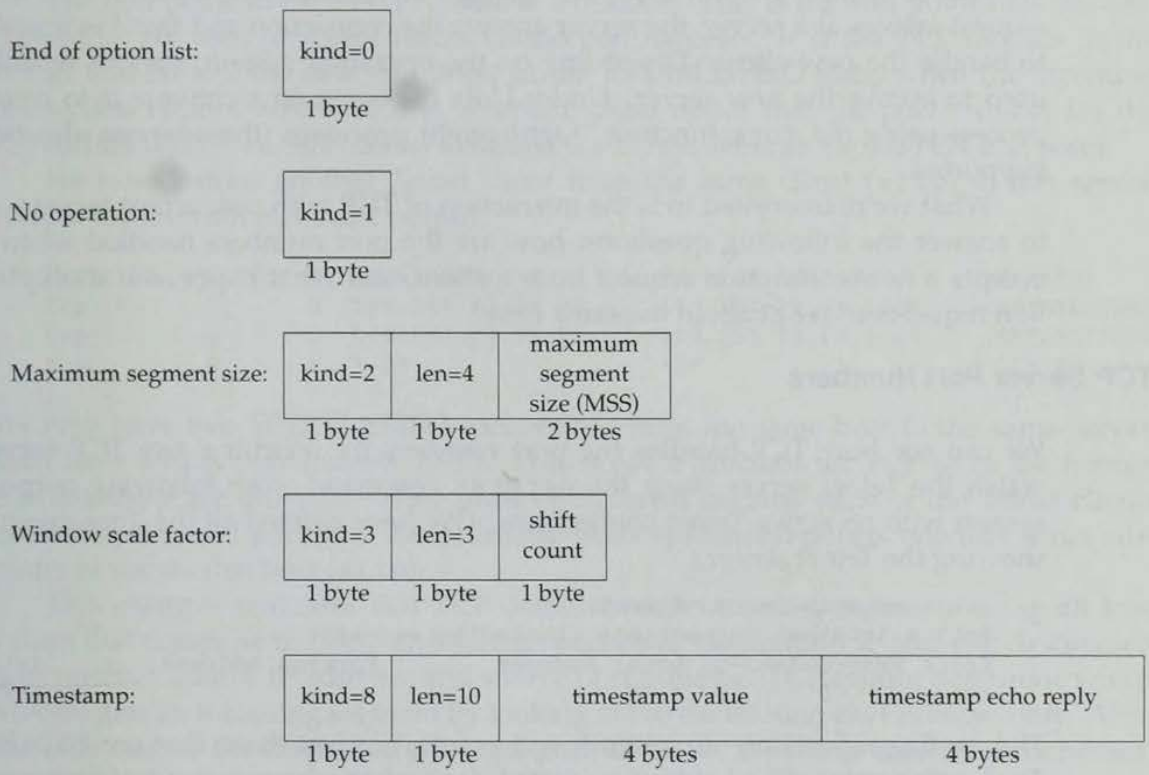


Figure 18.20 TCP options.

Every option begins with a 1-byte *kind* that specifies the type of option. The options with a *kind* of 0 and 1 occupy a single byte. The other options have a *len* byte that follows the *kind* byte. The length is the total length, including the *kind* and *len* bytes.

The reason for the no operation (NOP) option is to allow the sender to pad fields to a multiple of 4 bytes. If we initiate a TCP connection from a 4.4BSD system, the following TCP options are output by `tcpdump` on the initial SYN segment:

```
<mss 512,nop,wscale 0,nop,nop,timestamp 146647 0>
```

The MSS option is set to 512, followed by a NOP, followed by the window scale option. The reason for the first NOP is to pad the 3-byte window scale option to a 4-byte

boundary. Similarly, the 10-byte timestamp option is preceded by two NOPs, to occupy 12 bytes, placing the two 4-byte timestamps onto 4-byte boundaries.

Four other options have been proposed, with *kind*s of 4, 5, 6, and 7 called the selective-ACK and echo options. We don't show them in Figure 18.20 because the echo options have been replaced with the timestamp option, and selective ACKs, as currently defined, are still under discussion and were not included in RFC 1323. Also, the T/TCP proposal for TCP transactions (Section 24.7) specifies three options with *kind*s of 11, 12, and 13.

18.11 TCP Server Design

We said in Section 1.8 that most TCP servers are concurrent. When a new connection request arrives at a server, the server accepts the connection and invokes a new process to handle the new client. Depending on the operating system, various techniques are used to invoke the new server. Under Unix the common technique is to create a new process using the `fork` function. Lightweight processes (threads) can also be used, if supported.

What we're interested in is the interaction of TCP with concurrent servers. We need to answer the following questions: how are the port numbers handled when a server accepts a new connection request from a client, and what happens if multiple connection requests arrive at about the same time?

TCP Server Port Numbers

We can see how TCP handles the port numbers by watching any TCP server. We'll watch the Telnet server using the `netstat` command. The following output is on a system with no active Telnet connections. (We have deleted all the lines except the one showing the Telnet server.)

```
sun % netstat -a -n -f inet
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address        Foreign Address        (state)
tcp        0      0  *.23                 *.*                    LISTEN
```

The `-a` flag reports on all network end points, not just those that are ESTABLISHED. The `-n` flag prints IP addresses as dotted-decimal numbers, instead of trying to use the DNS to convert the address to a name, and prints numeric port numbers (e.g., 23) instead of service names (e.g., Telnet). The `-f inet` option reports only TCP and UDP end points.

The local address is output as `*.23`, where the asterisk is normally called the *wildcard* character. This means that an incoming connection request (i.e., a SYN) will be accepted on any local interface. If the host were multihomed, we could specify a single IP address for the local IP address (one of the host's IP addresses), and only connections received on that interface would be accepted. (We'll see an example of this later in this section.) The local port is 23, the well-known port number for Telnet.

The foreign address is output as `*.*`, which means the foreign IP address and foreign port number are not known yet, because the end point is in the LISTEN state, waiting for a connection to arrive.

We now start a Telnet client on the host `slip` (140.252.13.65) that connects to this server. Here are the relevant lines from the `netstat` output:

```
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp        0      0  140.252.13.33.23      140.252.13.65.1029     ESTABLISHED
tcp        0      0  *.23                  *.*                    LISTEN
```

The first line for port 23 is the ESTABLISHED connection. All four elements of the local and foreign address are filled in for this connection: the local IP address and port number, and the foreign IP address and port number. The local IP address corresponds to the interface on which the connection request arrived (the Ethernet interface, 140.252.13.33).

The end point in the LISTEN state is left alone. This is the end point that the concurrent server uses to accept future connection requests. It is the TCP module in the kernel that creates the new end point in the ESTABLISHED state, when the incoming connection request arrives and is accepted. Also notice that the port number for the ESTABLISHED connection doesn't change: it's 23, the same as the LISTEN end point.

We now initiate another Telnet client from the same client (`slip`) to this server. Here is the relevant `netstat` output:

```
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp        0      0  140.252.13.33.23      140.252.13.65.1030     ESTABLISHED
tcp        0      0  140.252.13.33.23      140.252.13.65.1029     ESTABLISHED
tcp        0      0  *.23                  *.*                    LISTEN
```

We now have two ESTABLISHED connections from the same host to the same server. Both have a local port number of 23. This is not a problem for TCP since the foreign port numbers are different. They must be different because each of the Telnet clients uses an ephemeral port, and the definition of an ephemeral port is one that is not currently in use on that host (`slip`).

This example reiterates that TCP demultiplexes incoming segments using all four values that comprise the local and foreign addresses: destination IP address, destination port number, source IP address, and source port number. TCP cannot determine which process gets an incoming segment by looking at the destination port number only. Also, the only one of the three end points at port 23 that will receive incoming connection requests is the one in the LISTEN state. The end points in the ESTABLISHED state cannot receive SYN segments, and the end point in the LISTEN state cannot receive data segments.

Next we initiate a third Telnet client, from the host `solaris` that is across the SLIP link from `sun`, and not on its Ethernet.

```
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp        0      0  140.252.1.29.23       140.252.1.32.34603     ESTABLISHED
tcp        0      0  140.252.13.33.23      140.252.13.65.1030     ESTABLISHED
tcp        0      0  140.252.13.33.23      140.252.13.65.1029     ESTABLISHED
tcp        0      0  *.23                  *.*                    LISTEN
```

The local IP address of the first ESTABLISHED connection now corresponds to the interface address of SLIP link on the multihomed host `sun` (140.252.1.29).

Restricting Local IP Address

We can see what happens when the server does not wildcard its local IP address, setting it to one particular local interface address instead. If we specify an IP address (or hostname) to our `sock` program when we invoke it as a server, that IP address becomes the local IP address of the listening end point. For example

```
sun % sock -s 140.252.1.29 8888
```

restricts this server to connections arriving on the SLIP interface (140.252.1.29). The `netstat` output reflects this:

```
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp        0      0  140.252.1.29.8888     *.*                    LISTEN
```

If we connect to this server across the SLIP link, from the host `solaris`, it works.

```
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp        0      0  140.252.1.29.8888     140.252.1.32.34614     ESTABLISHED
tcp        0      0  140.252.1.29.8888     *.*                    LISTEN
```

But if we try to connect to this server from a host on the Ethernet (140.252.13), the connection request is not accepted by the TCP module. If we watch it with `tcpdump` the SYN is responded to with an RST, as we show in Figure 18.21.

```
1  0.0                  bsdi.1026 > sun.8888: S 3657920001:3657920001(0)
                                                win 4096 <mss 1024>
2  0.000859 (0.0009)    sun.8888 > bsdi.1026: R 0:0(0) ack 3657920002 win 0
```

Figure 18.21 Rejection of a connection request based on local IP address of server.

The server application never sees the connection request—the rejection is done by the kernel's TCP module, based on the local IP address specified by the application.

Restricting Foreign IP Address

In Section 11.12 we saw that a UDP server can normally specify the foreign IP address and foreign port, in addition to specifying the local IP address and local port. The interface functions shown in RFC 793 allow a server doing a passive open to have either a fully specified foreign socket (to wait for a particular client to issue an active open) or a unspecified foreign socket (to wait for any client).

Unfortunately, most APIs don't provide a way to do this. The server must leave the foreign socket unspecified, wait for the connection to arrive, and then examine the IP address and port number of the client.

Figure 18.22 summarizes the three types of address bindings that a TCP server can establish for itself. In all cases, *lport* is the server's well-known port and *localIP* must be the IP address of a local interface. The ordering of the three rows in the table is the order that the TCP module applies when trying to determine which local end point receives an incoming connection request. The most specific binding (the first row, if supported) is tried first, and the least specific (the last row with both IP addresses wildcarded) is tried last.

Local Address	Foreign Address	Description
localIP.lport	*foreignIP.fport*	restricted to one client (normally not supported)
localIP.lport	*.*	restricted to connections arriving on one local interface: *localIP*
*.*lport*	*.*	receives all connections sent to *lport*

Figure 18.22 Specification of local and foreign IP addresses and port number for TCP server.

Incoming Connection Request Queue

A concurrent server invokes a new process to handle each client, so the listening server should always be ready to handle the next incoming connection request. That's the underlying reason for using concurrent servers. But there is still a chance that multiple connection requests arrive while the listening server is creating a new process, or while the operating system is busy running other higher priority processes. How does TCP handle these incoming connection requests while the listening application is busy?

In Berkeley-derived implementations the following rules apply.

1. Each listening end point has a fixed length queue of connections that have been accepted by TCP (i.e., the three-way handshake is complete), but not yet accepted by the application.

   Be careful to differentiate between TCP accepting a connection and placing it on this queue, and the application taking the accepted connection off this queue.

2. The application specifies a limit to this queue, commonly called the backlog. This backlog must be between 0 and 5, inclusive. (Most applications specify the maximum value of 5.)

3. When a connection request arrives (i.e., the SYN segment), an algorithm is applied by TCP to the current number of connections already queued for this listening end point, to see whether to accept the connection or not. We would expect the backlog value specified by the application to be the maximum number of queued connections allowed for this end point, but it's not that simple. Figure 18.23 shows the relationship between the backlog value and the real maximum number of queued connections allowed by traditional Berkeley systems and Solaris 2.2.

Backlog value	Max # of queued connections	
	Traditional BSD	Solaris 2.2
0	1	0
1	2	1
2	4	2
3	5	3
4	7	4
5	8	5

Figure 18.23 Maximum number of accepted connections allowed for listening end point.

Keep in mind that this backlog value specifies only the maximum number of queued connections for one listening end point, all of which have already been accepted by TCP and are waiting to be accepted by the application. This backlog has no effect whatsoever on the maximum number of established connections allowed by the system, or on the number of clients that a concurrent server can handle concurrently.

> The Solaris values in this figure are what we expect. The traditional BSD values are (for some unknown reason) the backlog value times 3, divided by 2, plus 1.

4. If there is room on this listening end point's queue for this new connection (based on Figure 18.23), the TCP module ACKs the SYN and completes the connection. The server application with the listening end point won't see this new connection until the third segment of the three-way handshake is received. Also, the client may think the server is ready to receive data when the client's active open completes successfully, before the server application has been notified of the new connection. (If this happens, the server's TCP just queues the incoming data.)
5. If there is not room on the queue for the new connection, TCP just ignores the received SYN. Nothing is sent back (i.e., no RST segment). If the listening server doesn't get around to accepting some of the already accepted connections that have filled its queue to the limit, the client's active open will eventually time out.

We can see this scenario take place with our `sock` program. We invoke it with a new option (`-O`) that tells it to pause after creating the listening end point, before accepting any connection requests. If we then invoke multiple clients during this pause period, it should cause the server's queue of accepted connections to fill, and we can see what happens with `tcpdump`.

```
bsdi % sock -s -v -q1 -O30 5555
```

The `-q1` option sets the backlog of the listening end point to 1, which for this traditional BSD system should allow two pending connection requests (Figure 18.23). The `-O30` option causes the program to sleep for 30 seconds before accepting any client connections. This gives us 30 seconds to start some clients, to fill the queue. We'll start four clients on the host `sun`.

Figure 18.24 shows the `tcpdump` output, starting with the first SYN from the first client. (We have removed the window size advertisements and MSS announcements. We have also marked the client port numbers in bold when the TCP connection is established—the three-way handshake.)

The first client's connection request from port 1090 is accepted by TCP (segments 1–3). The second client's connection request from port 1091 is also accepted by TCP (segments 4–6). The server application is still asleep, and has not accepted either connection yet. Everything has been done by the TCP module in the kernel. Also, the two clients have returned successfully from their active opens, since the three-way handshakes are complete.

```
 1   0.0                  sun.1090 > bsdi.7777: S 1617152000:1617152000(0)
 2   0.002310 ( 0.0023)   bsdi.7777 > sun.1090: S 4164096001:4164096001(0)
                                                  ack 1617152001
 3   0.003098 ( 0.0008)   sun.1090 > bsdi.7777: . ack 1

 4   4.291007 ( 4.2879)   sun.1091 > bsdi.7777: S 1617792000:1617792000(0)
 5   4.293349 ( 0.0023)   bsdi.7777 > sun.1091: S 4164672001:4164672001(0)
                                                  ack 1617792001
 6   4.294167 ( 0.0008)   sun.1091 > bsdi.7777: . ack 1

 7   7.131981 ( 2.8378)   sun.1092 > bsdi.7777: S 1618176000:1618176000(0)
 8  10.556787 ( 3.4248)   sun.1093 > bsdi.7777: S 1618688000:1618688000(0)
 9  12.695916 ( 2.1391)   sun.1092 > bsdi.7777: S 1618176000:1618176000(0)
10  16.195772 ( 3.4999)   sun.1093 > bsdi.7777: S 1618688000:1618688000(0)
11  24.695571 ( 8.4998)   sun.1092 > bsdi.7777: S 1618176000:1618176000(0)

12  28.195454 ( 3.4999)   sun.1093 > bsdi.7777: S 1618688000:1618688000(0)
13  28.197810 ( 0.0024)   bsdi.7777 > sun.1093: S 4167808001:4167808001(0)
                                                  ack 1618688001
14  28.198639 ( 0.0008)   sun.1093 > bsdi.7777: . ack 1

15  48.694931 (20.4963)   sun.1092 > bsdi.7777: S 1618176000:1618176000(0)
16  48.697292 ( 0.0024)   bsdi.7777 > sun.1092: S 4170496001:4170496001(0)
                                                  ack 1618176001
17  48.698145 ( 0.0009)   sun.1092 > bsdi.7777: . ack 1
```

Figure 18.24 `tcpdump` output for backlog example.

We try to start a third client in segment 7 (port 1092), and a fourth in segment 8 (port 1093). TCP ignores both SYNs since the queue for this listening end point is full. Both clients retransmit their SYNs in segments 9, 10, 11, 12, and 15. The fourth client's third retransmission is accepted (segments 12–14) because the server's 30-second pause is over, causing the server to remove the two connections that were accepted, emptying its queue. (The reason it appears this connection was accepted by the server at the time 28.19, and not at a time greater than 30, is because it took a few seconds to start the first client [segment 1, the starting time point in the output] after starting the server.) The third client's fourth retransmission is then accepted (segments 15–17). The fourth client connection (port 1093) is accepted by the server before the third client connection (port 1092) because of the timing interactions between the server's 30-second pause and the client's retransmissions.

> We would expect the queue of accepted connections to be passed to the application in FIFO (first-in, first-out) order. That is, after TCP accepts the connections on ports 1090 and 1091, we expect the application to receive the connection on port 1090 first, and then the connection on port 1091. But a bug has existed for years in many Berkeley-derived implementations causing them to be returned in a LIFO (last-in, first-out) order instead. Vendors have recently started fixing this bug, but it still exists in systems such as SunOS 4.1.3.

TCP ignores the incoming SYN when the queue is full, and doesn't respond with an RST, because this is a soft error, not a hard error. Normally the queue is full because the application or the operating system is busy, preventing the application from servicing incoming connections. This condition could change in a short while. But if the server's TCP responded with a reset, the client's active open would abort (which is what we saw

happen if the server wasn't started). By ignoring the SYN, the server forces the client TCP to retransmit the SYN later, hoping that the queue will then have room for the new connection.

A subtle point in this example, which is found in most TCP/IP implementations, is that TCP accepts an incoming connection request (i.e., a SYN) if there is room on the listener's queue, without giving the application a chance to see who it's from (the source IP address and source port number). This is not required by TCP, it's just the common implementation technique (i.e., the way the Berkeley sources have always done it). If an API such as TLI (Section 1.15) gives the application a way to learn when a connection request arrives, and then allows the application to choose whether to accept the connection or not, be aware that with TCP, when the application is supposedly told that the connection has just arrived, TCP's three-way handshake is over! Other transport layers may be implemented to provide this separation to the application between arrival and acceptance (i.e., the OSI transport layer) but not TCP.

> Solaris 2.2 provides an option that prevents TCP from accepting an incoming connection request until the application says so (`tcp_eager_listeners` in Section E.4).

This behavior also means that a TCP server has no way to cause a client's active open to fail. When a new client connection is passed to the server application, TCP's three-way handshake is over, and the client's active open has completed successfully. If the server then looks at the client's IP address and port number, and decides it doesn't want to service this client, all the server can do is either close the connection (causing a FIN to be sent) or reset the connection (causing an RST to be sent). In either case the client thought everything was OK when its active open completed, and may have already sent a request to the server.

18.12 Summary

Before two processes can exchange data using TCP, they must establish a connection between themselves. When they're done they terminate the connection. This chapter has provided a detailed look at how connections are established using a three-way handshake, and terminated using four segments.

We used `tcpdump` to show all the fields in the TCP header. We've also seen how a connection establishment can time out, how resets are sent, what happens with a half-open connection, and how TCP provides a half-close, simultaneous opens, and simultaneous closes.

Fundamental to understanding the operation of TCP is its state transition diagram. We've followed through the steps involved in connection establishment and termination, and the state transitions that take place. We also looked at the implications of TCP's connection establishment on the design of concurrent TCP servers.

A TCP connection is uniquely defined by a 4-tuple: the local IP address, local port number, foreign IP address, and foreign port number. Whenever a connection is terminated, one end must maintain knowledge of the connection, and we saw that the TIME_WAIT state handles this. The rule is that the end that does the active open enters this state for twice the implementation's MSL.

Exercises

18.1 In Section 18.2 we said that the initial sequence number (ISN) normally starts at 1 and is incremented by 64,000 every half-second and every time an active open is performed. This would imply that the low-order three digits of the ISN would always be 001. But in Figure 18.3 these low-order three digits are 521 in each direction. What's going on?

18.2 In Figure 18.15 we typed 12 characters and saw 13 bytes sent by TCP. In Figure 18.16 we typed eight characters but TCP sent 10 bytes. Why was 1 byte added in the first case, but 2 bytes in the second case?

18.3 What's the difference between a half-open connection and a half-closed connection?

18.4 If we start our `sock` program as a server, and then terminate it (without having a client connect to it), we can immediately restart the server. This implies that it doesn't go through the 2MSL wait state. Explain this in terms of the state transition diagram.

18.5 In Section 18.6 we showed that a client cannot reuse the same local port number while that port is part of a connection in the 2MSL wait. But if we run our `sock` program twice in a row as a client, connecting to the daytime server, we can reuse the same local port number. Additionally, we're able to create a new incarnation of a connection that should be in the 2MSL wait. What's going on?

```
sun % sock -v bsdi daytime
connected on 140.252.13.33.1163 to 140.252.13.35.13
Wed Jul  7 07:54:51 1993
connection closed by peer
sun % sock -v -b1163 bsdi daytime      reuse same local port number
connected on 140.252.13.33.1163 to 140.252.13.35.13
Wed Jul  7 07:55:01 1993
connection closed by peer
```

18.6 At the end of Section 18.6 when describing the FIN_WAIT_2 state, we mentioned that many implementations move a connection from this state into the CLOSED state if the application did a complete close (not a half-close) after just over 11 minutes. If the other end (in the CLOSE_WAIT state) waited 12 minutes before issuing its close (i.e., sending its FIN), what would its TCP get in response to the FIN?

18.7 Which end of a telephone conversation does the active open, and which does the passive open? Are simultaneous opens allowed? Are simultaneous closes allowed?

18.8 In Figure 18.6 we don't see an ARP request or an ARP reply. Obviously the hardware address for host `svr4` must be in the ARP cache on `bsdi`. What would change in this figure if this ARP cache entry was not present?

18.9 Explain the following `tcpdump` output. Compare it with Figure 18.13.

```
1  0.0               solaris.32990 > bsdi.discard: S 40140288:40140288(0)
                                                   win 8760 <mss 1460>
2  0.003295 (0.0033) bsdi.discard > solaris.32990: S 4208081409:4208081409(0)
                                                   ack 40140289 win 4096
                                                   <mss 1024>
3  0.419991 (0.4167) solaris.32990 > bsdi.discard: P 1:257(256) ack 1 win 9216
4  0.449852 (0.0299) solaris.32990 > bsdi.discard: F 257:257(0) ack 1 win 9216
5  0.451965 (0.0021) bsdi.discard > solaris.32990: . ack 258 win 3840
6  0.464569 (0.0126) bsdi.discard > solaris.32990: F 1:1(0) ack 258 win 4096
7  0.720031 (0.2555) solaris.32990 > bsdi.discard: . ack 2 win 9216
```

18.10 Why doesn't the server in Figure 18.4 combine the ACK of the client's FIN with its own FIN, reducing the number of segments to three?

18.11 In Figure 18.16 why is the sequence number of the RST 26368002?

18.12 Does TCP's querying the link layer for the MTU violate the spirit of layering?

18.13 Assume in Figure 14.16 that each DNS query is issued using TCP instead of UDP. How many packets are exchanged?

18.14 With an MSL of 120 seconds, what is the maximum at which a system can initiate new connections and then do an active close?

18.15 Read RFC 793 to see what happens when an end point that is in the TIME_WAIT state receives a duplicate of the FIN that placed it into this state.

18.16 Read RFC 793 to see what happens when an end point that is in the TIME_WAIT state receives an RST.

18.17 Read the Host Requirements RFC to obtain the definition of a *half-duplex* TCP close.

18.18 In Figure 1.8 (p. 11) we said that incoming TCP segments are demultiplexed based on the destination TCP port number. Is that correct?

19

TCP Interactive Data Flow

19.1 Introduction

The previous chapter dealt with the establishment and termination of TCP connections. We now examine the transfer of data using TCP.

Studies of TCP traffic, such as [Caceres et al. 1991], usually find that on a packet-count basis about half of all TCP segments contain bulk data (FTP, electronic mail, Usenet news) and the other half contain interactive data (Telnet and Rlogin, for example). On a byte-count basis the ratio is around 90% bulk data and 10% interactive, since bulk data segments tend to be full sized (normally 512 bytes of user data), while interactive data tends to be much smaller. (The above-mentioned study found that 90% of Telnet and Rlogin packets carry less than 10 bytes of user data.)

TCP obviously handles both types of data, but different algorithms come into play for each. In this chapter we'll look at interactive data transfer, using the Rlogin application. We'll see how delayed acknowledgments work and how the Nagle algorithm reduces the number of small packets across wide area networks. The same algorithms apply to Telnet. In the next chapter we'll look at bulk data transfer.

19.2 Interactive Input

Let's look at the flow of data when we type an interactive command on an Rlogin connection. Many newcomers to TCP/IP are surprised to find that each interactive keystroke normally generates a data packet. That is, the keystrokes are sent from the client to the server 1 byte at a time (not one line at a time). Furthermore, Rlogin has the

remote system (the server) echo the characters that we (the client) type. This could generate four segments: (1) the interactive keystroke from the client, (2) an acknowledgment of the keystroke from the server, (3) the echo of the keystroke from the server, and (4) an acknowledgment of the echo from the client. Figure 19.1 shows this flow of data.

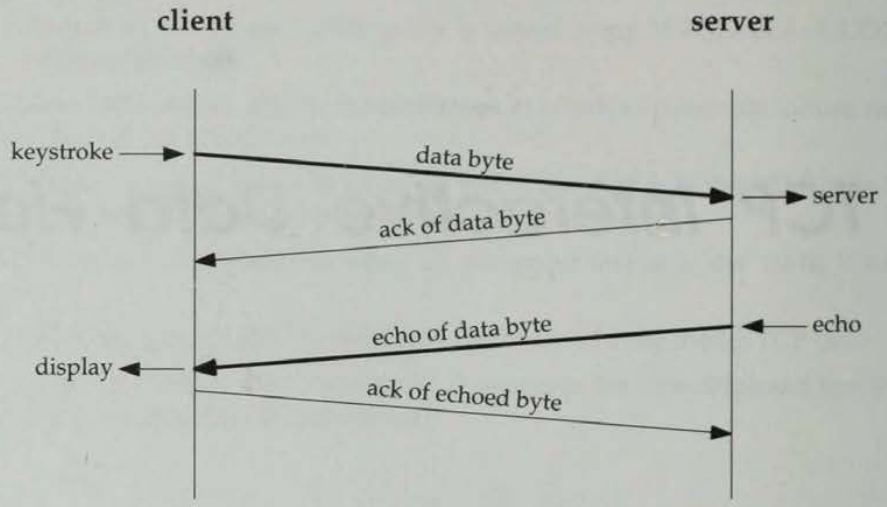


Figure 19.1 One possible way to do remote echo of interactive keystroke.

Normally, however, segments 2 and 3 are combined—the acknowledgment of the keystroke is sent along with the echo. We describe the technique that combines these (called delayed acknowledgments) in the next section.

We purposely use Rlogin for the examples in this chapter because it always sends one character at a time from the client to the server. When we describe Telnet in Chapter 26, we'll see that it has an option that allows lines of input to be sent from the client to the server, which reduces the network load.

Figure 19.2 shows the flow of data when we type the five characters `date\n`. (We do not show the connection establishment and we have removed all the type-of-service output. BSD/386 sets the TOS for an Rlogin connection for minimum delay.) Line 1 sends the character `d` from the client to the server. Line 2 is the acknowledgment of this character and its echo. (This is combining the middle two segments in Figure 19.1.) Line 3 is the acknowledgment of the echoed character. Lines 4–6 correspond to the character `a`, lines 7–9 to the character `t`, and lines 10–12 to the character `e`. The fractional second delays between lines 3–4, 6–7, 9–10, and 12–13 are the human delays between typing each character.

Notice that lines 13–15 are slightly different. One character is sent from the client to the server (the Unix newline character, from our typing the RETURN key) but two characters are echoed. These two characters are a carriage return and linefeed (CR/LF), to move the cursor back to the left and space down one line.

Line 16 is the output of the `date` command from the server. The 30 bytes are composed of the following 28 characters

```
Sat Feb  6 07:52:17 MST 1993
```

```
 1  0.0                bsdi.1023 > svr4.login: P 0:1(1) ack 1 win 4096
 2  0.016497 (0.0165)  svr4.login > bsdi.1023: P 1:2(1) ack 1 win 4096
 3  0.139955 (0.1235)  bsdi.1023 > svr4.login: . ack 2 win 4096

 4  0.458037 (0.3181)  bsdi.1023 > svr4.login: P 1:2(1) ack 2 win 4096
 5  0.474386 (0.0163)  svr4.login > bsdi.1023: P 2:3(1) ack 2 win 4096
 6  0.539943 (0.0656)  bsdi.1023 > svr4.login: . ack 3 win 4096

 7  0.814582 (0.2746)  bsdi.1023 > svr4.login: P 2:3(1) ack 3 win 4096
 8  0.831108 (0.0165)  svr4.login > bsdi.1023: P 3:4(1) ack 3 win 4096
 9  0.940112 (0.1090)  bsdi.1023 > svr4.login: . ack 4 win 4096

10  1.191287 (0.2512)  bsdi.1023 > svr4.login: P 3:4(1) ack 4 win 4096
11  1.207701 (0.0164)  svr4.login > bsdi.1023: P 4:5(1) ack 4 win 4096
12  1.339994 (0.1323)  bsdi.1023 > svr4.login: . ack 5 win 4096

13  1.680646 (0.3407)  bsdi.1023 > svr4.login: P 4:5(1) ack 5 win 4096
14  1.697977 (0.0173)  svr4.login > bsdi.1023: P 5:7(2) ack 5 win 4096
15  1.739974 (0.0420)  bsdi.1023 > svr4.login: . ack 7 win 4096

16  1.799841 (0.0599)  svr4.login > bsdi.1023: P 7:37(30) ack 5 win 4096
17  1.940176 (0.1403)  bsdi.1023 > svr4.login: . ack 37 win 4096
18  1.944338 (0.0042)  svr4.login > bsdi.1023: P 37:44(7) ack 5 win 4096
19  2.140110 (0.1958)  bsdi.1023 > svr4.login: . ack 44 win 4096
```

Figure 19.2 TCP segments when `date` typed on Rlogin connection.

plus a CR/LF pair at the end. The next 7 bytes sent from the server to the client (line 18) are the client's prompt on the server host: `svr4 % `. Line 19 acknowledges these 7 bytes.

Notice how the TCP acknowledgments operate. Line 1 sends the data byte with the sequence number 0. Line 2 ACKs this by setting the acknowledgment sequence number to 1, the sequence number of the last successfully received byte plus one. (This is also called the sequence number of the next expected byte.) Line 2 also sends the data byte with a sequence number of 1 from the server to the client. This is ACKed by the client in line 3 by setting the acknowledged sequence number to 2.

19.3 Delayed Acknowledgments

There are some subtle points in Figure 19.2 dealing with timing that we'll cover in this section. Figure 19.3 shows the time line for the exchange in Figure 19.2. (We have deleted all the window advertisements from this time line, and have added a notation indicating what data is being transferred.)

We have labeled the seven ACKs sent from `bsdi` to `svr4` as *delayed ACKs*. Normally TCP does not send an ACK the instant it receives data. Instead, it delays the ACK, hoping to have data going in the same direction as the ACK, so the ACK can be sent along with the data. (This is sometimes called having the ACK *piggyback* with the data.) Most implementations use a 200-ms delay—that is, TCP will delay an ACK up to 200 ms to see if there is data to send with the ACK.